爱丽丝在可微的仙境历险记

A primer on designing neural networks

第 I 卷 - 探访这片土地

Simone Scardapane

2

“

For, you see, so many out-of-the-way things had happened lately, that Alice had begun to think that very few things indeed were really impossible.”

第一章，兔子洞之旅

前言

这本书是关于（深度）神经网络主题的介绍，这是大型语言模型、生成式人工智能——以及许多其他应用的核心技术。由于术语 *neural* 带有大量历史包袱，并且神经网络仅仅是可微原语的组合，因此当可行时，我用更简单的术语“可微模型”来指代它们。

2009年，我几乎偶然间发现了一篇关于“深度”网络[Ben09]的论文，与此同时，像Theano[ARAA+16]这样的自动微分库也变得流行起来。就像爱丽丝一样，我偶然进入了一个奇异的编程领域——一个*differentiable*奇妙的世界，在这里简单的事情，比如选择一个元素，却异常困难，而其他事情，比如识别猫，却出奇地简单。

我已经花费了十多年时间阅读、实施和教授这些想法。这本书是对我在这个过程中所学知识的一种粗略尝试，重点关注其设计和最常见组件。由于该领域发展迅速，我试图在以下方面取得良好平衡：

理论和方法，历史考虑和近期趋势。我假设读者对机器学习和线性代数有所了解，但在必要时我会尝试涵盖基础知识。

Gather round, friends: it's time for our beloved Alice's Adventures in a differentiable wonderland

内容

1 | 简介

神经网络已成为我们日常世界的组成部分，无论是公开的（以大型语言模型、LLMs的形式），还是隐藏在幕后，通过推动无数技术和科学发现，包括无人机、汽车、搜索引擎、分子设计和推荐系统 [WFD+23]。正如我们将看到的，所有这一切都是通过依赖一个非常小的指导原则和组件集来实现的，构成了本书的核心，而研究重点已经转移到将它们扩展到物理可能性的极限。

缩放的力量体现在相对较新的概念——神经缩放定律中，这反过来又推动了人工智能（AI）领域的巨额投资 [KMH+20，HBE+24]：非正式地说，对于几乎任何任务，同时增加数据、计算能力和模型大小——几乎总是——会导致准确性的显著提升。换句话说，为了实现特定任务的给定准确性所需的计算能力，每经过一个时间段 [HBE+24] 就会以一个常数因子下降。将简单、通用的工具与指数级增加的计算能力相结合的巨大力量在于

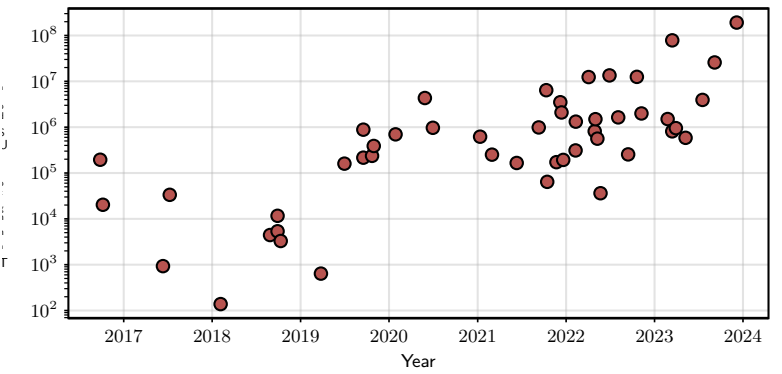


图 F.1.1: *Training cost (in US dollars) of notable AI models released from 2016. Training cost is correlated to the three key factors of scaling laws: size of the datasets, compute power, and size of the models. As performance steadily increases, variations in modeling become asymptotically less significant [HBE+24]. Data reproduced from the Stanford AI Index Report 2024.*[2]

人工智能被R. Sutton称为*bitter lesson*。[1]

如果我们接受尺度定律为既定，我们手中就剩下了一个几乎神奇的工具。简而言之，神经网络被优化来近似从其中抽取数据的一些概率分布。原则上，这种近似可能会失败：例如，现代神经网络如此之大，以至于它们可以轻易地记住它们所展示的所有数据 [ZBH+21] 并将其转化为一个简单的查找表。相反，训练好的模型被证明即使在训练数据中没有明确考虑的任务上也能很好地泛化 [ASA+23]。事实上，随着数据集规模的增加，什么是 *in-distribution* 和什么是 *out-of-distribution* 的概念变得模糊，大规模模型显示出强大的泛化能力的迹象和

[1] http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

[2] https://hai.stanford.edu/research/ai-index-report

迷人的低依赖纯记忆，即过度拟合 [PBE+22]。

极端大型模型的出现，这些模型可以用于各种下游任务（有时称为基础模型），以及充满活力的开源社区，[3] 也改变了我们与这些模型互动的方式。现在许多任务可以通过简单地 *prompting*（，即与文本或视觉指令交互) 在网络上找到的预训练模型 [ASA+23] 来解决，而模型的内部结构仍然是一个完全的黑盒。从高层次的角度来看，这类似于从必须用 C++ 等语言编写库，转向依赖于开源或商业软件，这些软件的源代码不可访问。这个比喻并不像它看起来那么牵强：如今，全球很少有团队拥有设计和发布真正大规模模型（如 Llama LLMs [TLI+23]）所需的计算能力和技术专长，就像很少有公司拥有构建企业 CRM 软件所需的资源。

同样，就像开源软件提供了从零开始定制或设计程序的无尽可能性一样，客户级硬件和一些独创性为你提供了大量选项来实验不同的可微分模型，从为你的任务微调它们 [LTM+22] 到合并模型 [AHS23]，为低功耗硬件量化它们，测试它们的鲁棒性，甚至设计全新的变体和想法。为此，你需要“揭开盖子”并理解这些模型如何内部处理和操作数据，以及所有从经验和调试中产生的技巧和怪癖。这本书是这个世界的入门点：如果，

[3] https://huggingface.co/

像爱丽丝一样，你天生好奇，我希望你会欣赏这段旅程。

关于本书

我们假设读者熟悉机器学习（ML）的基础知识，特别是监督学习（SL）。SL可以通过收集关于所需行为的数据来解决复杂任务，并通过“训练”（优化）系统来近似该行为。这个看似简单的想法非常强大：例如，图像生成可以转化为收集足够大的图像及其标题的集合的问题；模拟英语语言成为收集大量文本并学习预测前一句句子的问题；诊断X光片则等同于拥有包含相关医生决策的大型扫描数据库（图F.1.2）。

一般来说，学习是一个搜索问题。我们首先定义一个包含大量*degree-of-freedoms*（的程序，我们称之为参数），然后我们调整这些参数直到模型性能令人满意。为了使这个想法变得实用，我们需要高效的搜索方法来寻找最优配置，即使在存在数百万（或数十亿，或数万亿）参数的情况下。正如其名所示，可微分模型通过限制模型的选择为可微分组件（即我们可以求导的数学函数）来实现这一点。能够计算高维函数的导数（梯度）意味着我们知道如果我们稍微扰动它们的参数会发生什么，这反过来又导致了对它们的自动程序

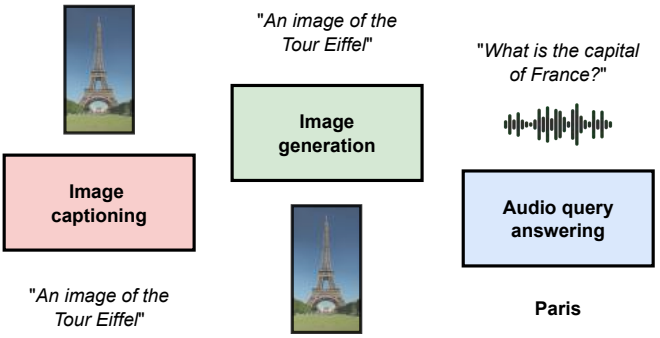


图 F.1.2：*Most tasks can be categorized based on the desired input - output we need: image generation wants an image (an ordered grid of pixels) from a text (a sequence of characters), while the inverse (image captioning) is the problem of generating a caption from an image. As another example, audio query answering requires a text from an audio (another ordered sequence, this time numerical). Fascinatingly, the design of the models follow similar specifications in all cases.*

优化（特别是自动微分和梯度下降）。描述这个设置是本书第一部分（第一部分，指南针和罗盘）的主题，从第二章到第六章。

通过将神经网络视为可微原语的组合，我们可以提出两个基本问题（图F.1.3）：首先，我们可以处理哪些数据类型作为输入或输出？其次，我们可以使用哪些原语？可微性是一个强烈的要求，它不允许我们直接处理许多标准数据类型，例如字符或整数，这些数据类型本质上是不连续的，因此是*discrete*。相比之下，我们将看到可微模型可以轻松地与表示为数字的大数组（我们称之为张量）的更复杂数据一起工作，例如图像，

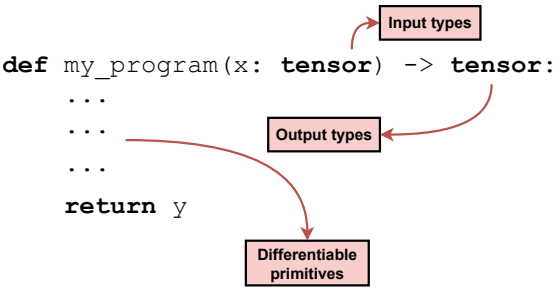


图 F.1.3：*Neural networks are sequences of differentiable **primitives** which operate on structured arrays (**tensors**): each primitive can be categorized based on its input/output signature, which in turn defines the rules for composing them.*

可以通过基本的线性和非线性变换的组合进行代数操作。

在本书的第二部分，我们关注一个可微分的组件原型示例，即卷积算子（第II部分，从第7章到第9章）。当我们的数据可以表示为有序元素序列时，可以应用卷积：这包括音频、图像、文本和视频等。在这个过程中，我们还介绍了一些有用的技术来设计*deep* (，即由许多步骤按顺序组成的模型，以及一些重要思想，如文本分词、序列的自回归生成和因果建模，这些构成了最先进LLMs的基础。

书籍的第三部分（第三部分，兔子洞之旅）继续通过考虑集合（最重要的是第10章和第11章中的注意力层和Transformer模型）的替代设计、图（第12章）以及最后的时间序列的循环层（第13章）来探索可微模型。

这本书辅以一个网站[4]，我将（希望）收集关于不专注于特定类型数据的兴趣主题的额外章节和材料，包括生成建模、条件计算、迁移学习和可解释性。这些章节在性质上更偏向研究，可以按任何顺序阅读。此外，我还提供了一系列以笔记本形式进行的指导性实验课程，涵盖了书中大部分内容以及对比学习和模型合并等高级主题.[5]

在可微分的土地上

神经网络拥有悠久而丰富的历史。这个名字本身是对20世纪早期尝试模拟（生物）神经元的早期尝试的回顾，类似的术语一直广泛存在：为了与现有框架保持一致，在接下来的章节中，我们可能会提到*neurons*、*layers*或例如*activations*。在经历了多波次的兴趣之后，2012年至2017年期间，由于大规模基准和竞赛的推动，网络复杂性出现了前所未有的增长，最值得注意的是我们在第9章中介绍的ImageNet大规模视觉识别挑战（ILSVRC）。2017年，由于引入了转换器（第10章），又出现了一次主要兴趣的浪潮：就像几年前计算机视觉被卷积模型取代一样，自然语言处理在很短的时间内被转换器所取代。这些年来，视频、图（第12章）和音频的进一步改进，最终导致了当前对以下内容的兴奋：

[4] https://sscardapane.it/alice-book

[5] http://tinyurl.com/guided-labs

LLMs、多模态网络和生成模型.[6]

这个时期与术语的快速演变相平行，从80年代的联接主义 [RHM86] 到使用深度学习来指代现代网络，以反对过去较小的 *shallower* 模型 [Ben09, LBH15]。尽管如此，所有这些术语仍然不可避免地模糊不清，因为现代（人工）网络几乎与生物神经网络和神经学 [ZER+23] 没有相似之处。观察现代神经网络，它们的本质特征是由可微块组成：因此，在本书中，当可行时，我更喜欢使用可微模型这个术语。将神经网络视为可微模型直接引申到更广泛的话题——可微编程，这是一门新兴学科，它将计算机科学和优化相结合，更广泛地研究可微计算机程序 [BR24].[7]

随着我们穿越这个可微分模型的世界，我们也在穿越历史：线性模型数值优化的基本概念（在第4章中介绍）至少在19世纪就已为人所知 [Sti81]；我们后来使用的所谓“全连接网络”可以追溯到20世纪80年代 [RHM86]；卷积模型也是已知的

[6] This is not the place for a complete historical overview of modern neural networks; for the interested reader, I refer to [Met22] as a great starting point.

[7] Like many, I was inspired by a ‘manifesto’ published by Y. LeCun on Facebook in 2018: https://www.facebook.com/yann.lecun/posts/10155003011462143. For the connection between neural networks and open-source programming (and development) I am also thankful to a second manifesto, published by C. Raffel in 2021: https://colinraffel.com/blog/a-call-to-build-models-like-we-build-open-source-software.html.


The New York Times

NEW NAVY DEVICE LEARNS BY DOING; Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

July 8, 1958

图 F.1.4：*AI hype - except it is 1958, and the US psychologist Frank Rosenblatt has gathered up significant media attention with his studies on "perceptrons", one of the first working prototypes of neural networks.*

在20世纪90年代末就已经使用 [LBBH98].[8] 然而，需要数十年的时间才能收集到足够的数据和力量，以实现它们在足够数据和足够参数下的表现。

虽然我们没有足够的空间深入探讨所有可能的主题（也由于研究进展迅速），但我希望这本书能提供足够的材料，使读者能够轻松地浏览最新的文献。

符号和记号

处理可微模型时的基本数据类型是张量，[9]，我们将其定义为

[8] For a history of NNs up to this period through interviews to some of the main characters, see [AR00]; for a large opinionated history there is also an *annotated history of neural networks* by J. Schmidhuber: `https://people.idsia.ch/~juergen/deep-learning-history.html`.

[9] In the scientific literature, tensors have a more precise definition as multilinear operators [Lim21], while the objects we use in the book are simpler multidimensional arrays. Although a misnomer, the use of *tensor* is so widespread that we keep this convention here.

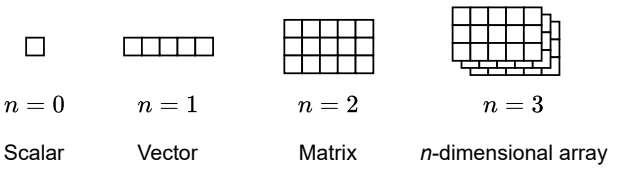


图 F.1.5：*Fundamental data types: scalars, vectors, matrices, and generic n-dimensional arrays. We use the name* ***tensors*** *to refer to them. n is called the* ***rank*** *of the tensor. We show the vector as a row for readability, but in the text we assume all vectors are column vectors.*

n 维度数组对象，通常是实数值。向任何阅读我们的数学家道歉，我们称 n 为张量的秩。书中的符号根据 n 而异：

1. 单项张量（$n = 0$）只是一个单一值（一个标量）。对于标量，我们使用小写字母，例如 x 或 y。[10]

2. 值的列($n = 1$)是向量。对于向量，我们使用小写粗体字，例如x。当需要区分它们时，对应的行向量表示为$\mathrm{x}^\top$。如果从上下文中可以清楚地看出，我们也可以忽略转置以提高可读性。

3. 值的矩形数组 (n =^2) 是矩阵。我们使用大写粗体字，例如 X 或 Y。

4. 对于 $n >$ 没有使用特定的符号 2. 我们避免使用如 $\mathscr{X}$ 这样的手写体符号，这些符号我们保留用于集合或概率分布。

[10] If you are wondering, scalars are named like this because they can be written as scalar multiples of one. Also, I promise to reduce the number of footnotes from now on.

对于处理张量，我们使用各种索引策略，这些策略在2.1节中描述得更好。在大多数情况下，理解一个算法或操作归结为理解每个涉及的张量的形状。为了简洁地表示形状，我们使用以下符号：{v*}

$$X \sim (b, h, w, 3)$$

这是一个形状为 (b、h、w、3) 的4阶张量。某些维度可以预先指定（例如，3），而其他维度可以用变量表示。我们使用相同的符号来表示从概率分布中抽取，例如 $\varepsilon \sim \mathcal{N}(0, 1)$，但我们很少这样做，并且符号的含义应该始终从上下文中清楚。因此，$\mathrm{x} \sim (d)$ 将取代更常见的 $\mathrm{x} \in \mathbb{R}^d$，对于 $\mathrm{X} \sim (n、d)$ 也同样如此，而不是 $\mathrm{X} \in \mathbb{R}^{n \times d}$。最后，我们可能想要约束张量的元素，为此我们使用特殊的符号：

1. $\mathrm{x} \sim$ 二进制(c) 表示一个只有二进制值的张量，即来自集合 $\{0, 1\}$ 的元素。2. $\mathrm{x} \sim \Delta(a)$ 表示属于所谓单纯形的向量，即 $x_i \geq 0$ 和 $\sum_i x_i = 1$。对于更高阶的张量，例如 $\mathrm{X} \sim \Delta(n, c)$，我们假设应用了相对于最后一个维度的归一化（例如，在这种情况下，X_i 的每一行都属于单纯形）。

在必要时，每章都会引入附加符号。我们还有一些符号在旁边：

- 一瓶用于强调一些定义。我们有很多定义，尤其是在早期章节中，我们使用这个符号来直观地区分最重要的定义。

- 一个用于理解本书其余部分至关重要的章节的时钟 - 请不要跳过这些！
- 相反，一个用于更轻松章节的茶杯——这些章节通常较为随意，并且相对于整本书来说大多是可选的。

离别前的最后思考

本书源于我想要将我在罗马萨皮恩扎大学数据科学硕士学位课程中教授的《数据科学应用中的神经网络》课程内容以连贯的形式呈现的愿望。本书的核心章节构成了课程的主要内容，而其余章节则是根据年份的不同，我偶尔涉及的主题。一些部分通过我教授的（或我打算教授的）额外课程得到了补充，包括计算机工程中的神经网络部分、电信工程机器学习简介，以及多年来的一些教程、博士课程和暑期学校。

现代深度神经网络主题已有许多优秀（且近期）的书籍，包括[Pri23、ZLLS23、BB23、Fle23、HR22]。本书在开头涵盖与这些书籍相似的内容，而阐述和一些额外部分（或在高级章节的几个部分）交集较少，它们主要依赖于我的研究兴趣。我希望我能提供额外的（且补充的）观点对现有材料。

如我的名字选择所暗示的，理解可微分的 *programs* 来源于理论和编码：我们在两者之间有一个持续的互动。

设计模型及其实现方式，例如自动微分就是最好的例子。神经网络（大约从2012年开始）的当前复兴在很大程度上可以归因于强大软件库的可用性，从Theano [ARAA+16]到Caffe、Chainer，然后直接到TensorFlow、PyTorch和JAX等现代迭代版本。我尽可能将讨论与现有编程框架中的概念联系起来，重点关注PyTorch和JAX。然而，这本书不是编程手册，我建议查阅每个库的文档以获得完整的介绍。

在继续之前，我想列出这本书的一些额外内容 *is not*。首先，我试图挑选出一些既（a）现在普遍存在，又（b）足够通用，足以在不久的将来使用的概念。然而，我无法预见未来，也不追求完整性，因此这些章节的某些部分可能在您阅读时可能不完整或过时。其次，对于每个概念，我尝试提供一些文献中存在的变体示例（例如，从批量归一化到层归一化）。然而，请记住，还有数百个存在：我邀请您探索 Papers With Code 的许多页面。最后，这是一本关于可微分模型基本组件的书，但要在规模上实现它们（并使它们工作）需要工程复杂性以及（一点）直觉。我在硬件方面涉及很少，而对于后者，没有比经验和个人博客文章更好的了.[11]

[11] See for example this blog post by A. Karpathy: `http://karpathy.github.io/2019/04/25/recipe/`, or his recent **Zero to Hero** video series: `https://karpathy.ai/zero-to-hero.html`.

致谢

方程式的着色归功于ST John的一个美丽的LaTeX包.[12]《爱丽丝梦游仙境》的颜色图像以及页边空白中的黑白符号均从Shutterstock.com获得授权。正文中的《爱丽丝梦游仙境》图像是从原始约翰·坦尼尔插画中复制的，感谢维基媒体。我感谢Roberto Alma对本书早期草稿的反馈，并鼓励我出版本书。我还感谢Corrado Zoccolo、Emanuele Rodolà、Marcin S aby、Konstantin Burlachenko和Diego Sandoval对当前版本提供的广泛修改和建议，以及所有通过电子邮件给我反馈的人。

许可证

[12] https://github.com/st--/annotate-equations/tree/main

[13] https://creativecommons.org/licenses/by-sa/4.0/

第一部分 罗盘和指针

“Would you tell me, please, which way I ought to go from here?”
“That depends a good deal on where you want to get to,” said the Cat.
“I don’t much care where” said Alice.
“Then it doesn’t matter which way you go,” said the Cat.

第6章，猪和胡椒

2 | 数学预备知识

> 关于本章
>
> 我们在此压缩了阅读本书所需的数学概念。我们假设对这些主题有先验知识，更专注于描述特定符号并给出一个连贯的概述。在可能的情况下，我们强调这些材料（例如，张量）与其实际应用之间的关系。

该章节由三个依次相连的部分组成，从线性代数开始，过渡到n-维对象的梯度定义，最后讨论如何利用这种梯度来优化函数。附录A提供了一个关于概率理论的独立概述，重点关注最大似然原理。

本章充满了内容和定义：请耐心等待一下！

2.1 线性代数

我们在此回顾一些线性代数的基本概念，这些概念在以下内容中将会很有用（并且为了达成共识，我们将采用统一的符号）。本书大部分内容围绕着张量的概念展开。

定义 D.2.1（张量）

*A **tensor** X is an n-dimensional array of elements of the same type. In the book we use:*

$$X \sim (s_1, s_2, \ldots, s_n)$$

*to quickly denote the **shape** of the tensor.*

对于 $n = 0$ 我们得到标量（单个值），而对于 $n = 1$ 我们有向量，对于 $n = 2$ 我们有矩阵，其他情况下有更高维的数组。回想一下，我们用小写 x 表示标量，用小写粗体 x 表示向量，用大写粗体 X 表示矩阵。在此所述意义上的张量在深度学习中是基本的，因为它们非常适合大规模并行实现，例如使用 GPU 或更专业的硬件（例如 TPUs、IPUs）。

一个张量由其元素的类型及其 *shape* 描述。我们的大部分讨论将围绕浮点值张量（其具体格式我们将在稍后考虑）展开，但它们也可以定义为整数（例如，在分类中）或字符串（例如，用于文本）。张量可以被索引以获取其值的切片（子集），并且大多数NumPy索引约定 [1] 也可以应用。

[1] If you want a refresher: `https://numpy.org/doc/stable/user/basics.indexing.html`. For readability in the book we index from 1, not from 0. See also the exercises at the end of the chapter.

应用。对于简单方程，我们使用脚标：例如，对于一个三维张量 $X \sim (a、b、c)$，我们可以写 X_i 来表示大小为 $(b、c)$ 或 X_{ijk} 的一个切片。对于更复杂的表达式，我们使用逗号，例如 $X_{i,:,j:k}$ 来表示大小为 $(b、k-j)$ 的一个切片。当需要避免混乱时，我们使用浅灰色符号：

$$[X]_{ijk}$$

为了在视觉上将索引部分与其他部分分开，其中 [的参数
•] 也可以是一个表达式。

2.1.1 常见向量运算

我们主要关注可以写成可微操作组合的模型。事实上，我们的大多数模型将由基本的求和、乘法以及一些额外的非线性组合构成，例如指数 $\exp(x)$、正弦和余弦以及平方根。

向量 $\mathrm{x} \sim (d)$ 是一维张量的例子。线性代数书籍关注区分列向量 x 和行向量 $\mathrm{x}^\top$，我们将尽可能遵守这一惯例。在代码中这更复杂，因为行和列向量对应于形状为 $(1、d)$ 或 $(d、1)$ 的二维张量，这与形状为 (d) 的一维张量不同。这一点很重要，因为大多数框架都实现了受 NumPy 启发的广播规则[2]，从而导致非直观的行为。参见 Box C.2.1 了解在以下情况中出现的非常常见的错误示例：

[2] In a nutshell, broadcasting aligns the tensors' shape from the right, and repeats a tensor whenever possible to match the two shapes: https://numpy.org/doc/stable/user/basics.broadcasting.html.

```
导入 torch
x = torch.randn((4, 1)) # "列" y = torch.randn((4,)) # 1D 张
量 print((x + y).shape) # [输出]: (4,4) (因为广播!)
```

盒子 C.2.1: *An example of (probably incorrect) broadcasting, resulting in a matrix output from an elementwise operation on two vectors due to their shapes. The same result can be obtained in practically any framework (NumPy, TensorFlow, JAX, ...).*

隐式广播张量形状。

向量拥有自己的代数（我们称之为向量空间），在这种意义上，任何两个形状相同的向量x和y都可以进行线性组合 $z = ax + by$，以提供一个第三个向量：

$$z_i = ax_i + by_i$$

如果我们把向量理解为 d-维欧几里得空间中的一个点，那么和被解释为构成一个平行四边形，而向量到原点的距离由欧几里得 (ℓ_2) 范数给出：

$$\|\mathbf{x}\| = \sqrt{\sum_i x_i^2}$$

平方范数 $\|x\|^2$ 特别有趣，因为它对应于元素平方的和。我们感兴趣的基矢量操作是内积（或点积），它通过逐元素相乘两个向量，并将结果值相加得到。

定义 D.2.2（内积）

The inner product between two vectors x, y $\sim (d)$ *is given by the expression:*

$$\langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x}^\top \mathbf{y} = \sum_i x_i y_i \tag{E.2.1}$$

⟨的表示法 $\bullet, \bullet\rangle$ 在物理学中很常见，我们有时为了清晰起见会使用它。重要的是，两个向量的点积是一个标量。例如，如果 $\mathrm{x} = [0.1, 0, -0.3]$ 和 $\mathrm{y} = [-4.0, 0.05, 0.1]$:

$$\langle \mathbf{x}, \mathbf{y} \rangle = -0.4 + 0 - 0.03 = -0.43$$

一个点积的简单几何解释可以通过它与两个向量之间的角度 α 的关系给出:

$$\mathbf{x}^\top \mathbf{y} = \|\mathbf{x}\| \|\mathbf{y}\| \cos(\alpha) \tag{E.2.2}$$

因此，对于两个归一化向量，使得 $\| \bullet \| = 1$，点积等于它们之间角度的余弦值，在这种情况下，我们称点积为余弦相似度。余弦相似度 $\cos(\alpha)$ 在 1（两个向量指向同一方向）和 -1（两个向量指向相反方向）之间振荡，当 $\langle \mathrm{x}, \mathrm{y} \rangle = 0$ 时，产生垂直方向的正交向量。从另一个角度来看，对于两个归一化向量（具有单位范数），一旦我们固定第一个参数 x，我们得到:

$$\mathbf{y}^* = \arg\max \langle \mathbf{x}, \mathbf{y} \rangle = \mathbf{x} \tag{E.2.3}$$

在 arg max 表示寻找与最高可能输出值对应的 y 值的操作。从 (E.2.3) 我们可以看出，为了最大化点积，第二个向量必须等于第一个向量。这很重要，因为在接下来的章节中，x 将代表输入，而 w 将代表（可调整）参数，这样当 x 与 w “共鸣”时（模板匹配），点积就可以最大化。

我们以两个额外的观察结束，这些观察将是有用的。首先，我们可以将一个向量元素的求和写成它与由全为1的向量1的点积，即$\mathbf{1} = [1, 1, \ldots, 1]^\top$：

$$\langle \mathbf{x}, \mathbf{1} \rangle = \sum_{i=1}^{d} x_i$$

其次，两个向量之间的距离也可以用它们的点积来表示：

$$\|\mathbf{x} - \mathbf{y}\|^2 = \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{y}, \mathbf{y} \rangle - 2\langle \mathbf{x}, \mathbf{y} \rangle$$

案例 y = 0 给我们 $\|x\|^2 = \langle x, x \rangle$。这两个方程在编写方程或代码时都可能很有用。

2.1.2 常见矩阵运算

在二维情况下，我们有以下矩阵：

$$\mathbf{X} = \begin{bmatrix} X_{11} & \cdots & X_{1d} \\ \vdots & \ddots & \vdots \\ X_{n1} & \cdots & X_{nd} \end{bmatrix} \sim (n, d)$$

在这种情况下，我们可以谈论一个有 n 行和 d 列的矩阵。对于以下内容，特别重要的是，可以将矩阵理解为 n 个 (x_1，x_2，...，x_n) 向量的堆叠，其中堆叠以行方式组织：

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix}$$

我们说X代表一批数据向量。正如我们将看到的，定义模型（无论是数学上还是代码上）来处理这类批量数据是惯例。矩阵的一个基本操作是乘法：

定义 D.2.3（矩阵乘法）

For any two matrices $X \sim (a,b)$ *and* $Y \sim (b, c)$ *of compatible shape, matrix multiplication* $Z = XY$*, with* $Z \sim (a, c)$ *is defined element-wise as:*

$$Z_{ij} = \langle \mathbf{X}_i, \mathbf{Y}_j^\top \rangle \tag{E.2.4}$$

i.e., the element (i, j) *of the product is the dot product between the* i*-th row of* X *and the* j*-th column of* Y.

作为一个特殊情况，如果第二项是一个向量，我们有一个矩阵-向量乘积：

$$\mathbf{z} = \mathbf{W}\mathbf{x} \tag{E.2.5}$$

如果我们将X解释为一组向量，矩阵乘法$XW^\top$是一种简单的向量化的方式

计算 n 的点积，如 (E.2.5) 所示，对 X 的每一行进行一次，使用单个线性代数运算。作为另一个例子，矩阵与其转置的乘积 $\mathrm{XX}^\top \sim (n, n)$，是同时计算 X 的所有可能行对点积的向量化方法。

我们最后提一下几个对矩阵的重要操作。

> 定义 D.2.4（Hadamard 乘法）
>
> *For two matrices of the same shape, the* ***Hadamard multiplication*** *is defined element-wise as:*
>
> $$[\mathbf{X} \odot \mathbf{Y}]_{ij} = X_{ij} Y_{ij}$$

虽然Hadamard乘法不具有标准矩阵乘法所有的有趣代数性质，但它常用于可微模型中执行*masking*操作（例如，将某些元素设置为0）或缩放操作。乘性交互在某些最近的一些模型家族中也变得流行，正如我们接下来将要看到的。

有时我们写出如 exp(X) 这样的表达式，这些表达式应被解释为对操作 *element-wise* 的应用：

$$[\exp(\mathbf{X})]_{ij} = \exp(X_{ij}) \tag{E.2.6}$$

与比较，“真”矩阵指数定义为平方矩阵为：

```
X = torch.randn((5, 5)) # 元素级指数 X = to
rch.exp(X) # 矩阵指数 X = torch.linalg.matr
ix_exp(X)
```

盒子 C.2.2: *Difference between the element-wise exponential of a matrix and the matrix exponential as defined in linear algebra textbooks. Specialized linear algebra operations are generally encapsulated in their own sub-package.*

$$\text{mat-exp}(\mathbf{X}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \tag{E.2.7}$$

重要地，（E.2.6）可以定义任何形状的张量，而（E.2.7）仅对（平方）矩阵有效。这就是为什么所有框架，如PyTorch，都有专门的模块来收集所有矩阵特定操作，例如逆和行列式。参见盒C.2.2以获取示例。

最后，我们可以对 *reduction* 操作（求和、平均值等）进行跨轴操作，而不指定下标和上标，在这种情况下，我们假设求和沿着整个轴进行：

$$\sum_i \mathbf{X}_i = \sum_{i=1}^{n} \mathbf{X}_i$$

在PyTorch和其他框架中，降维操作对应于具有axis参数的方法：

```
r = X.sum(axis=1)
```

关于矩阵乘法定义

为什么矩阵乘法定义为（E.2.4）而不是Hadamard乘法？考虑一个向量x和定义在其上的某个通用函数f。如果函数满足$f(\alpha \mathrm{x}_1 + \beta \mathrm{x}_2) = \alpha f(\mathrm{x}_1) + \beta f(\mathrm{x}_2)$，则称该函数为线性函数。任何这样的函数都可以表示为一个矩阵A（这可以通过扩展基表示中的两个向量来看到）。然后，矩阵向量乘积Ax对应于函数应用，$f(\mathrm{x}) = \mathrm{Ax}$，矩阵乘积AB对应于函数复合$f \circ g$，其中$(f \circ g)(\mathrm{x}) = f(g(\mathrm{x}))$和$g(\mathrm{x}) = \mathrm{Bx}$。

计算复杂性

我将使用矩阵乘法来介绍操作的*complexity*主题。观察(E.2.4)，我们看到，如果我们直接应用定义（我们称之为*time*复杂度），从输入参数X∼ $(a、b)$和Y∼ $(b、c)$计算矩阵Z∼ $(a、c)$需要ac个维度为b的内积，而顺序实现的内存需求仅仅是输出矩阵的大小（我们称之为*space*复杂度）。

为了从具体的硬件细节中抽象出来，计算机科学关注所谓的大的-$\mathcal{O}$记法，从德语*ordnung*（中，它代表*order*的近似）。如果一个函数$f(x)$被称为$\mathcal{O}(g(x))$，其中我们假设输入和输出都是非负的，如果我们能找到一个常数c和一个值x_0，使得：

$$f(x) \leq c g(x) \quad \text{for any } x \geq x_0 \tag{E.2.8}$$

这意味着当 x 足够大时，我们可以忽略分析中 $g(x)$ 之外的所有因素。这被称为渐近分析。因此，我们可以说矩阵乘法的朴素实现是 $\mathcal{O}(abc)$，与所有三个输入参数线性增长。对于大小为 (n、n) 的两个方阵，我们说矩阵乘法在输入维度上是 *cubic*。

推理渐近复杂度很重要（且优雅），但仅根据大-$\mathcal{O}$复杂度选择算法并不一定能够转化为实际性能的提升，这取决于许多细节，例如使用的硬件、支持的并行性等等.[3]作为一个例子，已知对于大小为 (n，n) 的两个平方矩阵相乘的最佳渐近算法，对于常数 $c < 2.4$ [CW82]，其规模为 $\mathcal{O}(n^c)$，这比原始实现的三次 $\mathcal{O}(n^3)$ 要求要好得多。然而，这些算法在高度并行的硬件（如GPU）上难以高效并行化，因此在实践中很少使用。

注意，从渐近复杂性的角度来看，访问具有 k 处理器的并行环境没有影响，因为它只能提供（最多）一个常数 $\frac{1}{k}$ 加速，超过非并行实现。此外，渐近复杂性不考虑将数据从一个位置移动到另一个位置所需的时间，

[3]When you call a specific primitive in a linear algebra framework, such as matrix multiplication `A @ B` in PyTorch, the specific low-level implementation that is executed (the **kernel**) depends on the run-time hardware, through a process known as **dispatching**. Hence, the same code can run via a GPU kernel, a CPU kernel, a TPU kernel, etc. This is made even more complex by compilers such as XLA (`https://openxla.org/xla`), which can optimize code by fusing and optimizing operations with a specific target hardware in mind.

这可能在许多情况下成为主要瓶颈.[4]在这些情况下，我们说实现是*memory-bound*而不是*compute-bound*。实际上，这只能通过在代码上运行分析器来检查。我们将看到，由于渐近复杂性和观察到的复杂性的相互作用，分析算法的复杂度远非易事。

2.1.3 高阶张量运算

向量与矩阵有趣，因为它们允许我们定义大量在更高维度中未定义或复杂的运算（例如，矩阵指数，矩阵乘法，行列式等）。当我们进入更高维度时，我们感兴趣的多数运算要么是矩阵运算的批处理变体，要么是矩阵运算和归约运算的特定组合。

例如，考虑两个张量 $X \sim (n,\ a,\ b)$ 和 $Y \sim (n,\ b,\ c)$。批矩阵乘法（BMM）定义为:

$$[\mathrm{BMM}(X,Y)]_i = \mathbf{X}_i\mathbf{Y}_i \sim (n,a,c) \tag{E.2.9}$$

操作在大多数框架中对其参数的批处理版本进行透明操作，这些参数在此情况下假定是 *leading dimensions*（第一个维度)。例如，PyTorch 中的批处理矩阵乘法与标准矩阵乘法相同，参见盒 C.2.3。

作为一个减少操作的例子，考虑两个

[4] https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html

> X = torch.randn((4, 5, 2)) Y = torch.randn((4, 2, 3)) (to rch.matmul(X, Y)).shape # 或 X @ Y # [Out]: (4, 5, 3)

盒子 C.2.3: *BMM in PyTorch is equivalent to standard matrix multiplication. Most operations can work on (possibly) batched inputs.*

张量 X，$Y \sim (a,\ b,\ c)$。点积（GDT）的广义形式可以写为：

$$\mathrm{GDT}(X,Y) = \sum_{i,j,k} [X \odot Y]_{ijk} \tag{E.2.10}$$

这是一个简单的点积，其输入经过“展平”。本简要概述涵盖了本书其余部分将使用的大多数张量运算，必要时再介绍额外材料。

2.1.4 爱因斯坦的符号表示

这是一个可选部分，涵盖了einsum，[5]一套约定，允许用户使用基于文本字符串的简单语法指定几乎所有的张量操作（包括降维、求和、乘法）。

为了引入这个符号，让我们再次考虑前面在（E.2.9）和（E.2.10）中给出的两个例子，并明确写出所有轴：

[5] https://numpy.org/doc/stable/reference/generated/numpy.einsum.html

```
# 批量矩阵乘法 M = torch.einsum('ijz,izk->ijk', A, B)
# 广义点积 M = torch.einsum('ijk,ijk->', A, B)
```

盒C.2.4：*Examples of using einsum in PyTorch.*

$$M_{ijk} = \sum_z A_{ijz} B_{izk} \tag{E.2.11}$$

$$M = \sum_i \sum_j \sum_k X_{ijk} Y_{ijk} \tag{E.2.12}$$

与爱因斯坦的符号一致，[6]，我们可以通过移除求和符号来简化这两个方程，按照约定，任何出现在右侧但不在左侧的索引都被求和：

$$M_{ijk} = A_{ijz} B_{izk} \triangleq \sum_z A_{ijz} B_{izk} \tag{E.2.13}$$

$$M = X_{ijk} Y_{ijk} \triangleq \sum_i \sum_j \sum_k X_{ijk} Y_{ijk} \tag{E.2.14}$$

然后，我们可以通过将索引隔离在唯一的字符串中（此时操作数现在在左侧）来压缩这两个定义：

- ‘$ijz,izk \rightarrow ijk$’（批矩阵乘法）；
- ‘$ijk,ijk \rightarrow$’（广义点积）。

[6]The notation we use is a simplified version which ignores the distinction between upper and lower indices: https://en.wikipedia.org/wiki/Einstein_notation.

```
M = jax.numpy.einsum('ijz,izk->ijk', A, B)
```

盒子 C.2.5: *Example of using einsum in JAX - compare with Box C.2.4.*

存在(E.2.13)-(E.2.14)中的定义与其简化的字符串定义之间直接的、一对一的对应关系。这在大多数框架中的einsum操作中实现，参见框C.2.4。

这个记法的优点是，我们不需要记住框架的API来实现给定的操作；并且从一个框架转换到另一个框架是透明的，因为 einsum 语法是等效的。例如，PyTorch 有几种矩阵乘法方法，包括 matmul 和 bmm，具有不同的广播规则和形状约束，而 einsum 为所有这些提供了统一的语法。此外，我们批处理矩阵乘法的 einsum 定义与 JAX 中的定义相同，例如，参见 Box C.2.5。

与转置轴一起工作也很简单。例如，对于 $A \sim (n、a、b)$ 和 $B \sim (n、c、b)$，通过在 einsum 定义中切换相应的轴，可以得到 $[\mathrm{A}]_i$ 乘以 $[\mathrm{B}^\top]_i$ 的批量乘法：

```
M = torch.einsum('ijz,ikz->ijk', A, B)
```

由于这些原因，einsum及其推广（如流行的einsops[7]包）最近获得了广泛的流行。

[7] http://einops.rocks

2.2 梯度和雅可比矩阵

如名称 *differentiable* 所暗示的，梯度在本书中起着关键作用，通过从梯度下降中派生的半自动机制提供了一种优化我们模型的方法。我们在此回顾一些关于多值函数微分的基本定义和概念。我们关注那些将后来变得至关重要的属性，这在一定程度上是以数学精度为代价的。

2.2.1 标量函数的导数

从简单的函数 $y = f(x)$ 开始，该函数具有标量输入和标量输出，其导数定义如下。

> 定义 D.2.5（导数）*The* ***derivative*** *of* $f(x)$ *is defined as:*
>
> $$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{E.2.15}$$

我们使用各种符号来表示导数：∂将泛指任何维度的导数和梯度（向量、矩阵）；∂_x或$\frac{\partial}{\partial x}$用于突出我们对其求导的输入参数（当需要时）；而$f'(x)$专门用于标量函数，有时也称为*Lagrange's notation*。

我们在此不关注函数导数的存在性（即使在连续函数中，导数也并非处处存在），我们将其视为已知。我们将在第6章中讨论非光滑函数的导数时，例如在0处的$f(x) = |x|$，再触及这一点。

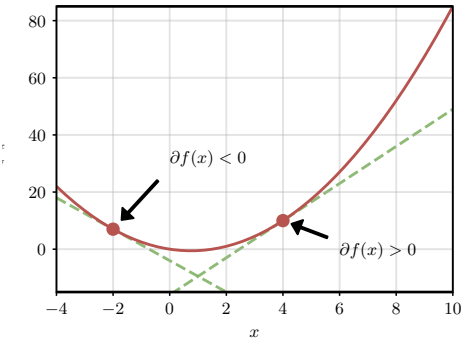


图 F.2.1：*Plot of the function $f(x) = x^2$ along with the derivatives on two separate points.*

导数的简单函数可以通过直接应用定义来获得，例如，多项式、对数或正弦的导数应该是熟悉的：

$$\partial x^p = px^{p-1}$$
$$\partial \log(x) = \frac{1}{x}$$
$$\partial \sin(x) = \cos(x)$$

几何上，导数可以理解为通过一点的切线的斜率，或者等价地，为该点处函数本身的一阶最佳近似，如图 F.2.1 所示。这是一个基本观点，因为直线的斜率告诉我们函数在邻近区域是如何演变的：对于正斜率，函数在右侧增加，在左侧减少（再次强调，对于足够小的区间），而对于负斜率，情况则相反。我们将看到，这种洞察力也适用于向量值函数。

我们回忆一些扩展到多维情况下的导数的重要性质：

- 线性：导数是线性的，因此导数

一个和的导数之和：

$$\partial\Big[f(x)+g(x)\Big]=f'(x)+g'(x).$$

- 乘积法则：

$$\partial\Big[f(x)g(x)\Big]=f'(x)g(x)+f(x)g'(x),$$

- 链式法则：函数复合的导数通过乘以相应的导数给出：

$$\partial\Big[f(g(x))\Big]=f'(g(x))g'(x) \tag{E.2.16}$$

2.2.2 梯度和方向导数

考虑现在一个函数 $y=f(\mathrm{x})$，它以向量 $\mathrm{x}\sim(d)$ 作为输入。在这里谈论无穷小扰动没有意义，除非我们指定这个扰动的方向（而在标量情况下我们只有“左”和“右”，在这种情况下，我们在欧几里得空间中有无限可能的方向）。在最简单的情况下，我们可以考虑沿着 i-轴移动，同时保持所有其他值不变：

$$\partial_{x_i}f(\mathbf{x})=\frac{\partial y}{\partial x_i}=\lim_{h\to 0}\frac{f(\mathbf{x}+h\mathbf{e}_i)-f(\mathbf{x})}{h}, \tag{E.2.17}$$

$\mathrm{e}_i\sim(d)$ 是 i-th 基础向量（单位矩阵的第 i-行）：

$$[\mathbf{e}_i]_j=\begin{cases}1 & \text{if } i=j\\ 0 & \text{otherwise}\end{cases} \tag{E.2.18}$$

(E.2.17) 被称为偏导数。将所有偏导数堆叠在一起，我们得到一个称为 d-维向量的东西

函数的梯度。

定义 D.2.6（梯度）

*The **gradient** of a function* $y = f(\mathrm{x}\{\mathrm{v}^2\}$

$$\nabla f(\mathbf{x}) = \partial f(\mathbf{x}) = \begin{bmatrix} \partial_{x_1} f(\mathbf{x}) \\ \vdots \\ \partial_{x_d} f(\mathbf{x}) \end{bmatrix} \tag{E.2.19}$$

因为梯度是基本的，我们使用特殊符号 $\nabla f(\mathrm{x})$ 来区分它们。那么在一般方向 v 上的位移呢？在这种情况下，我们得到方向导数：

$$\mathrm{D}_{\mathrm{v}} f(\mathbf{x}) = \lim_{h \to 0} \frac{f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x})}{h}, \tag{E.2.20}$$

空间中的运动可以通过考虑每个轴上的单个位移来分解，因此可以很容易地证明方向导数由梯度与位移向量 v 的点积给出：

$$\mathrm{D}_{\mathrm{v}} f(\mathbf{x}) = \langle \nabla f(\mathbf{x}), \mathbf{v} \rangle = \sum_i \partial_{x_i} f(\mathbf{x}) v_i \tag{E.2.21}$$

Displacement on the i-th axis

因此，知道如何计算函数的梯度就足以计算所有可能的方向导数。

2.2.3 雅可比矩阵

让我们现在考虑一个函数 $\mathrm{y} = f(\mathrm{x})$ 的通用情况，其中 x $\sim (d)$ 是一个向量输入，就像之前一样，这次输出为 a *vector*y $\sim (o)$。正如我们将看到的，这是我们需要考虑的最一般情况。因为我们有多个

输出，我们为每个输出计算一个梯度，它们的堆叠提供了一个称为 f 的雅可比矩阵的 $(o、d)$ 矩阵。

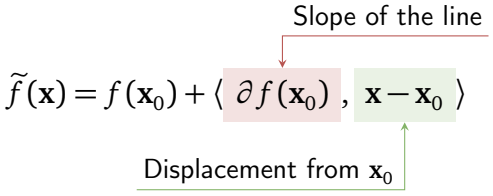
定义 D.2.7（雅可比）*The **Jacobian** matrix of a function* $\mathrm{y} = f(\mathrm{x})$*,* $\mathrm{x} \sim (d)$*,* $\mathrm{y} \sim (o)$ *is given by:*

$$\partial f(\mathbf{x}) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_o}{\partial x_1} & \cdots & \frac{\partial y_o}{\partial x_d} \end{pmatrix} \sim (o, d) \tag{E.2.22}$$

我们恢复 $o = 1$ 的梯度，以及 $d = o = 1$ 的标准导数。雅可比矩阵继承了导数的所有性质：重要的是，函数复合的雅可比矩阵现在是对应单个雅可比矩阵的 *matrix multiplication*：

$$\partial [f(g(\mathbf{x}))] = [\partial f(\bullet)] \partial g(\mathbf{x}) \tag{E.2.23}$$

在 $g(\mathrm{x}) \sim (h)$ 中评估一阶导数。参见 [PP08，第 2] 章节以获取计算出的梯度雅可比的数值示例。与标量情况类似，梯度和雅可比可以理解为与特定点相切的线性函数。特别是，梯度在以下意义上是最佳的“一阶近似”。对于点 x_0，在 $f(\mathrm{x}_0)$ 的无穷小邻域中的最佳线性近似由以下给出：



这是泰勒定理。参见盒C.2.6和图F.2.2以了解标量情况下的可视化$f(x) = x^2 - 1.5x$。

```
# Generic function
f = lambda x: x**2-1.5*x

# Derivative (computed manually for now)
df = lambda x: 2*x-1.5

# Linearization at 0.5
x = 0.5
f_lin = lambda h: f(x) + df(x)*(h-x)

# Numerical check
print(f(x + 0.01))      # -0.5049
print(f_lin(x + 0.01)) # -0.5050
```

盒子 C.2.6: *Example of computing a first-order approximation (scalar case). The result is plotted in Figure F.2.2.*

关于雅可比矩阵的维度

我们以一个关于维度的繁琐注释结束，这将在以下内容中很有用。考虑以下函数：

$$\mathbf{y} = \mathbf{W}\mathbf{x}$$

当将 {v*} 视为 x 的函数时，其导数与之前一样，是一个 $(o,\ d)$ 矩阵，并且可以证明：

$$\partial_{\mathbf{x}}[\mathbf{W}\mathbf{x}] = \mathbf{W}$$

当将输入视为W的函数时，输入本身是一个$(o,\ d)$矩阵，在这种情况下，“雅可比”的形状为$(o,\ o,\ d)$（（见下一页的框图））。然而，我们总是可以想象一个相同的（同构的）函数，它以W的向量化为输入，即$\text{vect}(\mathrm{W}) \sim (od)$，在这种情况下，雅可比将是一个形状为$(o,\ od)$的矩阵。

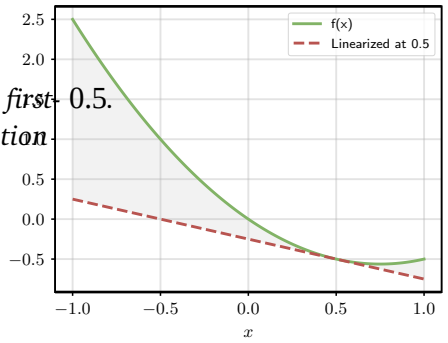


图 F.2.2：*The* 1.5x *and its first-order approximation at* 0.5. *function* $f(x) = x^2 -$ *shown in*

求解雅可比矩阵

为了计算雅可比矩阵 $\partial_{\mathbf{W}}\mathbf{W}\mathbf{x}$，我们可以逐元素重写该表达式：

$$y_i = \sum_j W_{ij} x_j$$

从其中我们立即发现：

$$\frac{\partial y_i}{\partial W_{ij}} = x_j \tag{E.2.24}$$

注意，要显式地实现雅可比矩阵（将其存储在内存中），我们需要很多重复的值。正如我们在第6章中将会看到的，这可以通过以下方式避免，因为在实践中，我们只关心雅可比矩阵对另一个张量的应用。

这个快速示例阐明了我们所说的从符号角度来看，仅使用向量输入和输出就足够了的意思。然而，在第六章中，当我们为了符号的简洁性使用矩阵雅可比矩阵时（特别是为了避免索引的泛滥），需要注意这一点，但这些雅可比矩阵的大小可能“隐藏”在输入的实际形状中。

并且输出，最重要的是批量大小。在第6章中，我们将看到通过考虑所谓的向量-雅可比乘积，实际上可以避免显式计算雅可比矩阵。这也可以通过将雅可比矩阵视为抽象线性映射来形式化——参见[BR24]以了解该主题的正式概述。

2.3 梯度下降

为了理解访问梯度的有用性，考虑最小化一个通用函数 $f(\mathrm{x})$ 的问题，其中 $\mathrm{x} \sim (d)$：

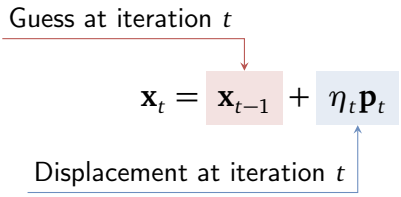
$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}) \tag{E.2.25}$$

类似于 arg max，$\arg\min f(\mathrm{x})$ 表示寻找与 $f(\mathrm{x})$ 可能的最低值相对应的 x 的值。我们假设该函数具有单个输出（单目标优化），并且我们优化 x 的域是不受约束的。

在本书的其余部分，x将编码我们模型的参数，而f将描述模型本身在我们数据上的性能，这是一种称为监督学习的设置，我们将在下一章介绍。在保持一般性的情况下，我们可以考虑最小化而不是最大化，因为最小化$f(\mathrm{x})$与最大化$-f(\mathrm{x})$等价，反之亦然（为了可视化这一点，想象一个一维函数并将其绕x轴旋转，想象其低点会发生什么）。

在极少数情况下，我们可能能够用闭式形式表达解（我们将在4.1.2节中看到最小二乘优化中的一个例子）。一般来说，

然而，我们被迫求助于迭代过程。假设我们从随机猜测 x_0 开始，并且对于每一次迭代，我们迈出一步，将其分解为其大小 η_t （（步长））和方向 p_t：



(E.2.26)

我们称 η_t 为步长（或者，在机器学习术语中，称为学习率，原因将在下一章中变得清晰）。存在一个 η_t 使得 $f(x_t) \leq f(x_{t-1})$ 的方向 p_t 被称为下降方向。如果我们能为每次迭代选择一个下降方向，并且我们在步长选择上小心谨慎，那么 (E.2.26) 中的迭代算法将在短时间内收敛到一个最小值。

对于可微函数，我们可以通过使用(E.2.20)中的方向导数精确量化所有下降方向，因为它们可以被定义为相对于我们之前的猜测x_{t-1}引起负变化的那些方向：

$$\mathbf{p}_t \text{ is a descent direction } \Rightarrow \ D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) \leq 0$$

使用我们在第2.2节中学到的知识和(E.2.2)中关于余弦相似度的点积定义，我们得到：

$$D_{\mathbf{p}_t} f(\mathbf{x}_{t-1}) = \langle \nabla f(\mathbf{x}_{t-1}), \mathbf{p}_t \rangle = \|\nabla f(\mathbf{x}_{t-1})\| \|\mathbf{p}_t\| \cos(\alpha)$$

在 α 是 p_t 和 $\nabla f(x_{t-1})$ 之间的角度。考虑到右侧的表达式，第一项与 p_t 无关。因为我们假设 p_t 只编码运动方向，所以我们也可以安全地限制它

到 $\|\mathrm{p}_t\| = 1$，将第二项视为另一个常数。因此，根据余弦的性质，我们得出结论，任何角度在 $\pi/2$ 和 $3\pi/2$ 之间，且与 $\nabla f(\mathrm{x}_{t-1})$ 相关的 p_t 都是一个下降方向。在这些方向中，角度为 π) 的方向 $\mathrm{p}_t = -\nabla f(\mathrm{x}_{t-1})$ (具有最低可能的方向导数，我们将其称为最速下降方向。

将此见解与（E.2.26）中的迭代过程结合起来，我们得到了一个最小化任何可微函数的算法，我们称之为（最速）梯度下降。

定义 D.2.8（最速下降法）

*Given a differentiable function $f(\mathrm{x})$, a starting point x_0, and a step size sequence η_t, **gradient descent** proceeds as:*

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta_t \nabla f(\mathbf{x}_{t-1}) \tag{E.2.27}$$

我们将不会关注寻找适当步长的问题，我们只需假设“足够小”，以便梯度下降迭代可以减少f。在下一节中，我们将关注从通用初始化运行梯度下降所获得哪些点。请注意，梯度下降与计算梯度的过程一样高效：我们在第6章中引入了一个通用算法来此目的。

2.3.1 梯度下降的收敛

当讨论梯度下降的收敛性时，我们需要明确我们所说的函数“最小值点”是什么意思。如果你不关心收敛性并且信任梯度下降，可以毫不犹豫地进入下一节。

定义 D.2.9（最小）

*A **local minimum** of $f(\mathrm{x})$ is a point x^+ such that the following is true for some $\varepsilon > 0$:*

$$f(\mathbf{x}^+) \leq f(\mathbf{x}) \quad \forall \mathbf{x} : \underbrace{\|\mathbf{x} - \mathbf{x}^+\| < \varepsilon}_{\text{Ball of size } \varepsilon \text{ centered in } \mathbf{x}^+}$$

在文字上，当考虑x^+的一个足够小的邻域时，$f(\mathrm{x}^+)$的值是最小的。直观上，在这个点上，切线的斜率将为0，而在x^+的邻域内其他地方的梯度将向上。我们可以通过驻点概念来形式化第一个想法。

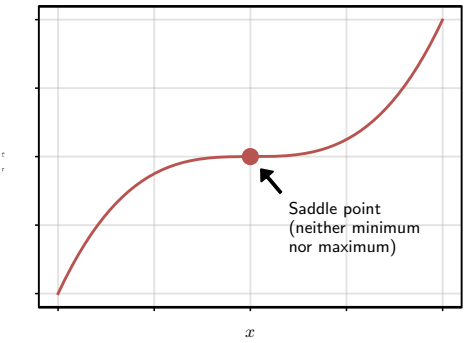
定义 D.2.10（驻定点）

*A point x^+ is called a **stationary point** of $f(\mathrm{x})$ if $\nabla f(\mathrm{x}^+) = 0$.*

静态点不仅限于极小值：它们可以是极大值（$-f(\mathrm{x})$的极小值）或鞍点，这些是函数曲率变化的拐点（见图F.2.3的示例）。一般来说，如果没有对f的任何约束，梯度下降只能被证明收敛到一个依赖于其初始化的通用静态点。

我们能做得更好吗？想象一个抛物线：在这种情况下，函数没有鞍点，它只有一个最小值。这个最小值也很特殊，从意义上讲，该点处的函数在整个定义域内达到其可能的最小值：我们称这为全局最小值。

图 F.2.3:
Simple example of a saddle point (try visualizing the tangent line in that point to see it is indeed stationary).



定义 D.2.11（全局最小值）

*A **global minimum** of $f(\mathrm{x})$ is a point x^* such that $f(\mathrm{x}^*) \leq f(\mathrm{x})$ for any possible input* x.

直观上，如果梯度下降在抛物线（从任何可能的初始化）上运行，它将收敛到这个全局最小值，因为所有梯度都将指向它。我们可以通过函数凸性的概念来推广这个想法。凸性的定义有很多种，我们为了阐述的简便选择以下定义。

定义 D.2.12（凸函数）

A function $f(\mathrm{x})$ is convex if for any two points x_1 and x_2 and $\alpha \in [0, 1]$ we have:

$$f(\underbrace{\alpha \mathbf{x}_1 + (1-\alpha)\mathbf{x}_2}_{\text{Interval from } \mathbf{x}_1 \text{ to } \mathbf{x}_2}) \leq \overbrace{\alpha f(\mathbf{x}_1) + (1-\alpha) f(\mathbf{x}_2)}^{\text{Line segment from } f(\mathbf{x}_1) \text{ to } f(\mathbf{x}_2)} \tag{E.2.28}$$

(E.2.28)的左侧是区间从x_1到x_2内任意点的f的值，而右侧是连接$f(x_1)$和$f(x_2)$的直线上相应的值。如果函数始终位于连接任意两点的直线下方，则它是凸函数（例如，向上开口的抛物线是凸函数）。

凸性在以下意义上保证了函数优化的简单性 [JK+17]:

1. 对于一个通用的 *non-convex* 函数，梯度下降收敛到 *stationary point*。除非我们查看高阶导数（导数的导数），否则无法再说更多。

2. 对于一个 *convex* 函数，梯度下降将收敛到一个 *global minimum*，无论初始化如何。

3. 如果(E.2.28)中的不等式以严格方式满足（严格凸性），则全局最小值也将是 *unique*。

这是一个困难属性：在非凸问题中使用梯度下降法找到全局最小值，唯一的解决方案是从任何可能的初始化点无限次运行优化器，将其转化为一个NP难任务 [JK+17]。

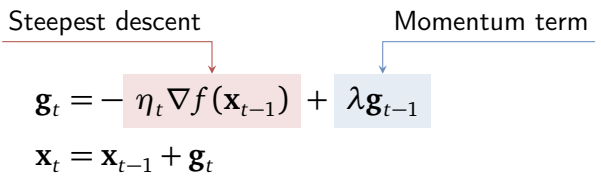
这次讨论具有强烈的历史意义。正如我们在第5章中将会看到的，任何非平凡模型都是非凸的，这意味着其优化问题可能有多个驻点。这与监督学习的替代算法（如支持向量机）形成对比，后者在保持非线性同时允许进行凸优化。有趣的是，即使在面临这种限制的情况下，复杂可微模型似乎也能很好地工作，其优化在以下意义上:

从合理的初始化开始，收敛到具有良好经验性能的点。

2.3.2 加速梯度下降

负梯度描述了最陡下降的方向，但仅在点的无穷小邻域内。正如我们在第5章（其中我们介绍了随机优化）中将会看到的，这些方向可能非常嘈杂，尤其是在处理大型模型时。已经开发出各种技术，通过选择更好的下降方向来加速优化算法的收敛。出于计算原因，我们特别感兴趣的方法是不需要高阶导数（例如，Hessian）或对函数的多次调用。

我们在这里描述了一种这样的技术，即动量，并参考[ZLLS23，第12章]以获得更广泛的介绍.[8]如果您将梯度下降想象成一个球“从山上滚下”，那么运动相对无序，因为每个梯度可以指向一个完全不同的方向（实际上，对于完美的步长选择和凸损失函数，后续迭代中的任何两个梯度都将相互垂直）。我们可以通过引入一个“动量”项来平滑这种行为，该动量项保留了一些来自先前梯度迭代的方向：



[8]See also this 2016 blog post by S. Ruder: https://www.ruder.io/optimizing-gradient-descent/.

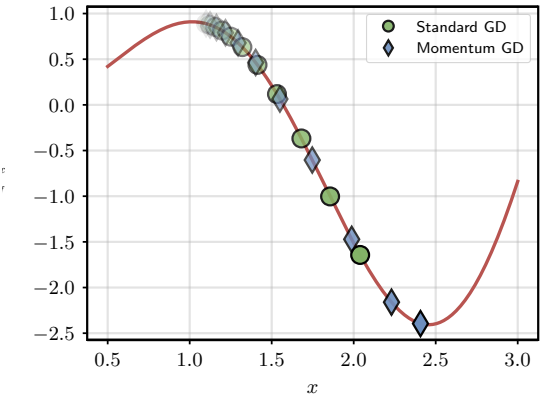


图 F.2.4：*GD and GD with momentum when minimizing the function* $x\sin(\hat{}2x)$ *starting from* $x = 1 + \varepsilon$*, with* $\lambda = 0.3$.

在初始化 $g_0 = 0$。请参阅图 F.2.4 以获取示例。系数 λ 决定了前一项被衰减的程度。实际上，展开两个项：

$$\begin{aligned}\mathbf{g}_t &= -\eta_t \nabla f(\mathbf{x}_{t-1}) + \lambda(-\eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda \mathbf{g}_{t-2}) \\ &= -\eta_t \nabla f(\mathbf{x}_{t-1}) - \lambda \eta_t \nabla f(\mathbf{x}_{t-2}) + \lambda^2 \mathbf{g}_{t-2}\end{aligned}$$

泛化地，时间 $t-n$ 的迭代通过一个因子 λ^{n-1} 被衰减。动量可以通过平滑优化路径 [SMDH13] 来证明可以加速训练。另一种常见的技术是根据梯度的幅度调整每个参数的步长 [ZLLS23]。结合这些想法的常见优化算法是 Adam [KB15]。Adam 的一个优点是它被发现对超参数的选择相对鲁棒，[9] 默认选择

[9] A hyper-parameter is a parameter which is selected by the user, as opposed to being learnt by gradient descent.

在大多数框架中，作为大多数情况下的良好起点。设计能够“取代”Adam（或其变体，如AdamW [LH19]）在深度学习中的默认优化器位置的新型优化器仍然是一个开放的研究问题，例如，参见[BN24]关于从第一原理设计神经网络定制优化器的最新工作。

使用加速优化算法的一个缺点可能是增加存储需求：例如，动量需要我们在内存中存储之前的梯度迭代，这会使优化算法所需的存储空间加倍（尽管在大多数情况下，计算梯度所需的内存是影响内存的最主要因素，正如我们将在第6.3节中看到的）。

从理论到实践

> **关于练习**
>
> 这本书没有经典的章节末尾练习，这些内容在许多现有的教科书中都有涉及。相反，我提出了一条自学路径，帮助你在阅读本书的过程中探索两个框架（JAX和PyTorch）。练习题的答案将在本书的网站上发布.[a] 这些部分充满了链接到在线材料的URL - 在你搜索时，它们可能已经过期或被移动。
>
> [a] https://www.sscardapane.it/alice-book

从基础开始

任何可微分模型设计者的起点是对NumPy的仔细研究。NumPy实现了一套通用的函数来操作多维数组（我们在书中称之为*tensors*），只要函数

对他们的内容进行索引和转换。您可以在库的快速入门部分了解更多信息.[10] 您应该熟悉在NumPy中处理数组，尤其是它们的索引：rougier/numpy-100[11] 存储库提供了一系列很好的、节奏较慢的练习，以测试您的知识。

[10] https://numpy.org/doc/stable/user/quickstart.html

[11] https://github.com/rougier/numpy-100

转向现实框架

尽管有影响，NumPy 在支持并行硬件（如GPU，除非使用额外的库）以及自动微分（在第6章中介绍）方面存在局限性。JAX 在复制NumPy接口的同时增加了扩展硬件支持、自动计算梯度以及向量化的map（jax.vmap）等额外转换。像PyTorch这样的框架也在其核心实现了类似NumPy的接口，但它们在命名和功能上进行了细微调整，并添加了构建可微分模型的高级工具。花时间浏览一下 jax.numpy.array 和 torch.tensor 的文档，了解它们与NumPy有多少相似之处。现在，您可以忽略像 torch.nn 这样的高级模块。在第6章介绍它们的梯度计算机制之后，我们将有更多关于这些框架设计的内容要讲。

实现梯度下降算法

要熟练掌握所有三个框架（NumPy、JAX、PyTorch），我建议将下面的练习重复三次——如果你熟悉语法，每个变体只需几分钟。考虑一个二维函数$f(\mathrm{x})$，$\mathrm{x} \sim (2)$，我们将定义域设为 $[0, 10]$:[12]

$$f(\mathbf{x}) = \sin(x_1)\cos(x_2) + \sin(0.5x_1)\cos(0.5x_2)$$

在继续阅读本书之前，为每个框架重复以下操作：

[12] I asked ChatGPT to generate a nice function with several minima and maxima. Nothing else in the book is LLM-generated, which I feel is becoming an important disclaimer to make.

1. 以向量化方式实现该函数，即给定一个输入矩阵X $\sim (n, 2)$的n，它应该返回一个向量$f(\mathrm{X}) \sim (n)$，其中$[f(\mathrm{X})]_i = f(\mathrm{X}_i)$。

2. 实现另一个计算其梯度的函数（硬编码的 - 我们还没有接触自动微分）。3. 编写一个基本的梯度下降过程，并可视化从多个起始点进行的优化过程路径。4. 尝试添加动量项并可视化梯度的范数，随着算法向平稳点移动，该范数应收敛到零。

如果您使用 JAX 或 PyTorch 解决练习，点（3）是使用 v map 向量化函数进行实验的好地方。

3 | 数据集和损失

> 关于本章
>
> 本章形式化了监督学习场景。我们介绍了数据集、损失、经验风险最小化和监督学习中做出的基本假设。我们最后提供了一个基于最大似然概念的监督学习概率公式。这一简短章节构成了本书其余部分的基础。

3.1 什么是数据集?

我们考虑一种场景，在这种情况下手动编码某个函数是不切实际的（例如，从现实世界图像中识别对象），但收集所需行为的示例却足够容易。这种例子比比皆是，从语音识别到机器人导航。我们用以下定义来形式化这个想法。

定义 D.3.1（数据集）

*A **supervised dataset** $\mathscr{S}_n$ of size n is a set of n pairs $\mathscr{S}_n = \{(x_i, y_i)\}_{i=1}^n$, where each (x_i, y_i) is an example of an input-output relationship we want to model. We further assume that each example is an **identically** and **independently** distributed (i.i.d.) draw from some unknown (and unknowable) probability distribution $p(x, y)$.*

查看附录A，如果您在阅读定义后想复习概率论。最后一个假设看起来很技术，但它是为了确保我们试图建模的关系是有意义的。特别是，样本具有相同的分布意味着我们试图逼近的是在时间上足够稳定且不变的东西。作为一个代表性例子，考虑从照片中识别汽车型号的数据集收集任务。如果我们收集短期内的图像，这个假设将得到满足，但如果从过去几十年收集图像，则无效，因为汽车型号会随时间变化。在后一种情况下，在数据集上训练和部署模型将失败，因为它将无法识别新模型，或者在使用时性能不佳。

类似地，样本独立分布意味着我们的数据集在收集过程中没有偏差，并且它足够代表整个分布。回到之前的例子，收集靠近特斯拉经销商的图像将是无效的，因为我们将会收集过多某种类型的图像，同时失去其他制造商和型号的图像。请注意，这些假设的有效性取决于上下文：在意大利收集的汽车数据集可能当

在罗马或米兰部署我们的模型，而当我们将在东京或台湾部署我们的模型时，它可能无效。应始终仔细检查独立同分布的假设，以确保我们将我们的监督学习工具应用于有效场景。有趣的是，现代大型语言模型是在如此大的数据分布上训练的，以至于甚至理解哪些任务是真正*in-distribution*针对什么*out-of-distribution* (以及模型能够推广多少)变得模糊[YCC+24]。

> 更多关于独立同分布性质的内容
>
> 重要地，确保独立同分布（i.i.d.）属性不是一个一次性过程，必须在模型的整个生命周期中不断进行检查。在汽车分类的情况下，如果不进行检查，随着时间的推移汽车分布的微妙变化将降低机器学习模型的性能，这是领域偏移的一个例子。作为另一个例子，推荐系统将改变用户与某个应用程序的交互方式，因为他们将开始对推荐系统的建议做出反应。这会创建反馈循环 [CMMB22]，需要不断重新评估系统和应用程序的性能。

3.1.1 监督学习的变体

存在许多标准监督学习场景的变体，尽管大多数成功的应用都以某种形式使用监督学习。例如，某些数据集可能没有可用的目标y_i，在这种情况下，我们谈论无监督学习。无监督学习的典型应用是聚类算法，其中我们希望聚合我们的

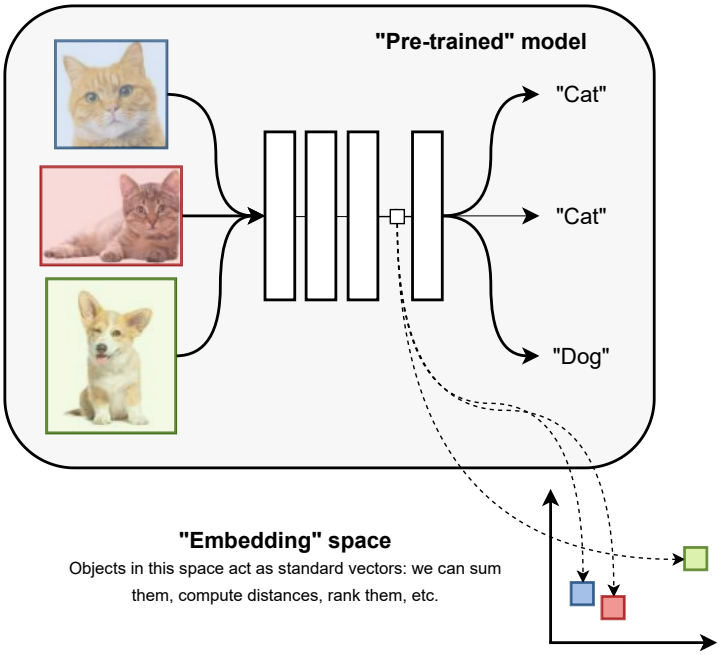


图 F.3.1：*Differentiable models process data by transforming it sequentially via linear algebra operations. In many cases, after we optimize these programs, the internal representations of the input data of the model (what we call a* ***pre-trained*** *model) have geometric properties: for example, semantically similar images are projected to points that are close in this "latent" space. Transforming data from a non-metric space (original input images) to a metric space (bottom right) is called* ***embedding*** *the data.*

输入数据到 *clusters*，使得簇内的点相似，簇间的点不相似 [HTF09]。作为另一个例子，在一个检索系统中，我们可能想要在一个大型数据库中搜索与用户给出的查询最相似的 top-k 个元素。

处理复杂数据，如图像时，这并不简单，因为如果我们对像素进行操作（即，即使是微小的扰动也可能修改数百万个像素），图像上的距离就定义不明确。然而，假设我们有一些可微分的模型，我们已经为其他任务对其进行了优化，我们假设它足够通用，例如图像分类。我们称之为预训练模型。正如我们将看到的，该模型的内部状态可以解释为高维空间中的向量。在许多情况下，这些向量已被证明具有有用的几何属性，即语义上相似的对象被（嵌入）到这些表示中彼此靠近的点。因此，我们可以使用这些潜在表示与标准聚类模型，如高斯混合模型 [HHWW14]。参见图 F.3.1 了解这一想法的高级概述。

如果我们无法访问预训练模型怎么办？无监督学习的一种常见变体被称为自监督学习（SSL，[ZJM+21]）。自监督学习的目标是自动从通用无监督数据集中找到一些监督目标，以优化一个可以在大量下游任务中使用的模型。例如，如果我们能够访问大量的文本语料库，我们总可以优化一个程序来预测一小段文本可能如何继续 [RWC+19]。当神经网络以自监督方式预训练时，它们也能有效地嵌入文本这一认识，产生了

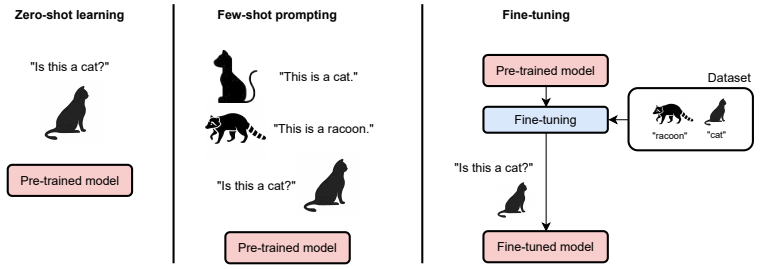


图 F.3.2：*Three ways of using trained models.* ***Zero-shot****: a question is directly given to the model. This can be achieved with generative language models (introduced in Chapter 8).* ***Few-shot prompting*** *is similar, but a few examples are provided as input. Both techniques can be employed only if the underlying model shows a large amount of generalization capabilities.* ***Fine-tuning****: the model is optimized via gradient descent on a small dataset of examples. This proceeds similarly to training the model from scratch.*

对社区产生深远影响 [MSC+13].[1]

我们将在第8章和第10章中看到，LLMs可以被视为这一基本思想的现代迭代，因为优化GPT或Llama [TLI+23]等模型始终从基于下一个标记预测的基本自监督训练开始。这些模型有时被称为基础模型。在最简单的情况下，它们可以直接用于新任务，例如回答查询：在这种情况下，我们说它们以零样本的方式使用。对于LLMs，也可以提供少量新任务的示例作为输入提示，在这种情况下，我们谈论少量提示。在

[1]Large-scale web datasets are also full of biases, profanity, and vulgar content. Recognizing that models trained on this data internalize these biases was another important realization [BCZ+16] and it is one of the major criticisms of closed-source foundation models [BGMMS21].

在大多数通用情况下，我们可以通过在新任务上对预训练的基础模型进行梯度下降来优化其参数：这被称为微调模型。参见图F.3.2以比较三种方法。在这本书中，我们专注于从头开始构建模型，但可以通过类似的方式进行微调。

微调由于在线上存在大量开源仓库而变得特别容易。[2] 微调可以在起始模型的全部参数上进行，或者只考虑更小的子集或少量额外的参数：这被称为参数高效微调（PEFT）[LDR23]。[3] 我们将在下一卷中考虑PEFT技术。

许多其他监督学习的变体是可能的，我们在这里没有空间详细列出，除了提供一些通用提示。如果只有部分数据集被标记，我们有一个半监督场景 [BNS06]。在第12章中，我们将看到一些半监督学习的例子。此外，我们还可以有多个数据集属于“相似”分布的场景，或者在不同时间段上的相同分布，这会产生无数问题，取决于任务或数据的提供顺序，包括领域自适应、元学习 [FAL17]、持续学习 [PKP+19、BBCJ20]、度量学习、反学习等。其中一些将在下一卷中讨论。

[2] https://huggingface.co/models

[3] Few-shot learning can also be done by fine-tuning the model. In cases in which fine-tuning is not needed, we say the model is performing **in-context learning** [ASA+23].

3.2 损失函数

一旦收集到数据，我们需要将我们对“近似”所需行为的想法形式化，我们通过引入损失函数的概念来实现这一点。

定义 D.3.2（损失函数）

Given a desired target y and the predicted value $\hat{y} = f(x)$ from a model f, a ***loss function*** $l(y, \hat{y}) \in \mathbb{R}$ *is a scalar, differentiable function whose value correlates with the performance of the model, i.e., $l(y, \hat{y}_1) < l(y, \hat{y}_2)$ means that the prediction $\hat{y}_1$ is better than the prediction $\hat{y}_2$ when considering the reference value (target) y.*

一个损失函数将我们对任务的了解和我们对解决方案空间的偏好嵌入到可以用于优化算法的实值尺度上。由于可微，它允许我们将我们的学习问题转化为可以通过梯度下降通过最小化数据集上的平均损失来解决的数学优化问题。

为此，给定一个数据集 $\mathcal{S}_n = \{(x_i, y_i)\}$ 和一个损失函数 $l(\cdot, \cdot)$，一个合理的优化任务是解决在数据集上通过任何可能的 *differentiable* 模型 f 实现的最小平均损失：

Average over the dataset

$$f^* = \arg\min_f \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)) \tag{E.3.1}$$

Prediction on the i-th sample

由于历史原因，（E.3.1）被称为经验风险最小化（ERM），其中 *risk* 被用作 *loss* 的通用同义词。有关该术语的起源，请参阅下一页的框。

在（E.3.1）中，我们隐含地假设我们在所有可能函数定义的空间上最小化我们的输入x。我们很快就会看到，我们的模型总可以通过一组张量w（（称为模型的参数））来参数化，最小化是通过搜索这些参数的最优值来实现的，我们用$f(x, w)$表示这种数值优化。因此，给定一个数据集$\mathcal{S}_n$，一个损失函数l和一个模型空间f，我们可以通过梯度下降（E.2.27）来优化经验风险（E.3.1）来训练我们的模型：

$$w^* = \arg\min_w \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i, w)) \tag{E.3.2}$$

在最小化时现在是对参数的张量 w 进行。

关于损失的可微性

在继续之前，我们对 ERM 框架提出两个观察。首先，注意 l 的可微性要求是基本的。考虑一个简单的二分类任务（我们将在下一章中适当介绍），其中 $y \in \{-1、+1\}$ 只能取两个值，-1 或 1。给定一个实值模型 $f(x) \in \mathbb{R}$，我们可以将这两个决策与 f 的符号相对应——我们将其表示为 sign$(f(x))$——并定义一个 0/1 损失如下：

$$l(y, \hat{y}) = \begin{cases} 0 & \text{if } \operatorname{sign}(\hat{y}) = y \\ 1 & \text{otherwise} \end{cases} \tag{E.3.3}$$

虽然这与我们直观的“正确”概念相符，但由于其梯度几乎总是为零（除非f的符号发生改变），因此作为损失函数是无用的，任何梯度下降算法都会在初始化时陷入停滞。在这种情况下，一个不那么直观的量是间隔$y\hat{y}$，它根据模型的符号是否与期望的符号一致而正负 [负]，但它与 (E.3.3) 中的 0/1 损失连续变化不同。在这种情况下，一个可能的损失函数是铰链损失$l(y, \hat{y}) = \max(0, 1 - y\hat{y})$，它用于训练支持向量模型。抛开细节不谈，这表明了设计损失函数以编码我们的性能概念与同时使其对数值优化有用的固有张力。

风险和损失

经验风险和期望风险以这种方式进行最小化通常与俄罗斯计算机科学家V. Vapnik的工作相关联 [Vap13]，这导致了*statistical learning theory* (SLT)领域的发展。SLT特别关注将(E.3.1)视为在某个限制函数类f和底层复杂度度量[PS$^+$03，SSBD14，MRT18]下的(E.3.5)的有限样本近似时的行为。现代神经网络（如强泛化，在过拟合本应被预期之后）的逆直觉特性在SLT中开辟了许多新的研究方向 [PBL20]。另见第9章的引言。

3.2.1 预期风险和过拟合

作为第二个观察，请注意，通过定义：{v*}

$$f(x) = \begin{cases} y & \text{if } (x,y) \in \mathcal{S}_n \\ \bar{y} & \text{otherwise} \end{cases}. \tag{E.3.4}$$

x is in the training set

Default value, e.g., 0

这是一个查找表，如果数据集中包含对 (x, y)，则返回预测 y，否则默认为某个常量预测 $\bar{y}$ (，例如，0)。假设当 $y = \hat{y}$ 时损失有下限，此模型将始终实现经验风险的最低可能值，同时不提供任何实际实用价值。

这显示了记忆和学习（优化）之间的差异。虽然我们通过优化训练数据上的某些平均损失量来寻找模型，如（E.3.1）中所述，但我们的真正目标是最小化这个量在某些未知、尚未看到的未来输入上。我们的训练集元素只是这个目标的代理。我们可以通过定义期望风险最小化问题来形式化这个想法。

> 定义 D.3.3（预期风险）
>
> *Given a probability distribution* $p(x, y)$ *and a loss function* l*, the* ***expected risk*** *(ER) is defined as:*
>
> $$\mathrm{ER}[f] = \mathbb{E}_{p(x,y)}[l(y, f(x))] \tag{E.3.5}$$

最小化（E.3.5）可以解释为最小化

平均（预期）损失跨 *all possible input-output pairs* (，例如，所有可能看到的电子邮件)。显然，具有低预期风险的模型将保证正确工作。然而，(E.3.5)中的数量在实际中无法计算，因为枚举和标记所有数据点是不可能的。经验风险提供了在给定数据集选择下的预期风险的估计，可以看作是ER项的蒙特卡洛近似。

损失函数之间的差异称为泛化差距：像（E.3.4）这样的纯记忆算法将具有较差的泛化能力，或者换句话说，它将过度拟合我们提供的特定训练数据。泛化能力可以通过保留一个单独的测试数据集 $\mathscr{T}_m$ 来在实践中进行测试，其中包含 m 个在训练期间从未使用过的数据点 $\mathscr{S}_n \cap \mathscr{T}_m = \emptyset$。然后，$\mathscr{S}_n$ 和 $\mathscr{T}_m$ 之间的经验损失差异可以用作过度拟合的近似度量。

3.2.2 如何选择一个有效的损失函数?

> 如果您尚未这样做，现在是学习（或快速浏览）附录A中材料的好时机，特别是概率分布、充分统计量和最大似然估计。

我们将在下一章中看到，损失编码了我们关于要解决的问题的先验知识，并且它对性能有重大影响。在某些情况下，对问题的简单考虑就足以设计有效的损失（例如，在第3.2节中对铰链损失所做的那样）。

然而，可以更原则性地工作

时尚通过将整个训练过程完全以概率术语重新表述，正如我们现在所展示的。这种表述为学习提供了一个不同的视角，这在某些情况下可能更直观或更有用。它也是许多书籍 [BB23] 的首选视角。我们在本节中提供了基本思想，并在本书后面的部分考虑具体应用。

关键观察如下。在第3.1节中，我们首先假设我们的示例来自分布 $p(x,\ y)$。根据概率乘法规则，我们可以将 $p(x,\ y)$ 分解为 $p(x,\ y) = p(x)p(y \mid x)$，使得 $p(x)$ 依赖于观察每个输入 x 的概率，条件项 $p(y \mid x)$ 描述了在给定输入 x 的条件下观察某个输出 y 的概率。如果我们假设概率质量主要集中在单个点 y 附近，即 $p(y \mid x)$ 接近所谓的狄拉克δ函数，那么用函数 $f(x)$ 近似 $p(y \mid x)$ 是有意义的，这极大地简化了整体问题表述。

然而，我们可以通过假设我们的模型 $f(x)$ 并不直接提供预测，而是用来参数化一个条件概率分布 $p(y \mid f(x))$ 的充分统计量，来放松这一点。例如，考虑一个分类问题，其中 $y \in \{1, 2, 3\}$ 可以取三个可能值。我们可以假设我们的模型有三个输出，这些输出参数化了一个在这些类别上的分类分布，

[4]We can also decompose it as $p(x, y) = p(x \mid y)p(y)$. Methods that require to estimate $p(x)$ or $p(x \mid y)$ are called **generative**, while methods that estimate $p(y \mid x)$ are called **discriminative**. Apart from language modeling, in this book we focus on the latter case. We consider generative modeling more broadly in the next volume.

使得：

$$p(\mathbf{y} \mid f(x)) = \prod_{i=1}^{3} f_i(x)^{y_i}$$

在 {v*} 中，y ∼ Binary(3) 是类别 y[5] 和 $f(x) \sim \Delta(3)$ 的独热编码，$f(x) \sim \Delta(3)$ 是每个类别的预测概率。作为另一个例子，假设我们想要预测一个单个标量值 $y \in \mathbb{R}$ (回归)。我们可以用双值函数 $f(x) \sim (2)$ 来建模，使得预测是一个具有适当均值和方差的高斯分布：

$$p(y \mid f(x)) = \mathcal{N}(y \mid f_1(x), \ f_2^2(x)\ ) \tag{E.3.6}$$

Squared to ensure positivity

在 $f(x)$ 的第二个输出被平方以确保预测方差保持正数的情况下。如所见，这是一个非常通用的设置，它包含了我们之前的讨论，并为设计者提供了更多的灵活性，因为选择 $p(y \mid x)$ 的特定参数化可能比选择特定的损失函数 $l(y$、$\hat{y})$ 更容易。此外，这个框架提供了一个更直接的方式来建模不确定性，例如 (E.3.6) 中的方差。

3.2.3 最大似然

如何训练一个概率模型？请记住，我们假设数据集中的样本 $\mathcal{S}_n$ 是来自概率分布 $p(x,\ y)$ 的独立同分布样本。因此，给定一个模型 $f(x)$，分配给数据集本身的概率

[5] Given an integer i, its one-hot representation is a vector of all zeros except the i-th element, which is 1. This is introduced formally in Section 4.2.

通过特定函数f的选择，数据集中每个样本的乘积给出：

$$p(\mathcal{S}_n \mid f) = \prod_{i=1}^{n} p(y_i \mid f(x_i))$$

数据集的量$p(\mathcal{S}_n \mid f)$被称为似然。对于$f(x)$的随机选择，模型将在所有可能的输入和输出上随机分配概率，我们特定数据集的似然将很小。因此，一个合理的策略是选择模型，使得数据集的似然最大化。这是最大似然方法的直接应用（参见附录A中的A.6节）。

定义 D.3.4（最大似然）

Given a dataset $\mathcal{S}_n = \{(x_i, y_i)\}$ *and a family of probability distributions* $p(y \mid f(x))$ *parameterized by* $f(x)$*, the* ***maximum likelihood*** *solution is given by:*

$$f^* = \arg\max_f \prod_{i=1}^{n} p(y_i \mid f(x_i)).$$

当所有概率分布都选定后，我们现在可以直接从概率定律中得到优化问题，这与之前不同，之前特定的损失是设计空间的一部分。然而，这两个观点是紧密相连的。在对数空间中工作并转换为最小化问题，我们得到：

$$\arg\max_f \left\{ \log \prod_{i=1}^{n} p(y_i \mid f(x_i)) \right\} = \arg\min_f \left\{ \sum_{i=1}^{n} -\log(p(y_i \mid f(x_i)) \right\} \tag{E.3.7}$$

因此，如果我们把 $-\log(p(y \mid f(x))$ 识别为要优化的“伪损失”，那么这两个公式是相同的。正如我们将看到的，实践中使用的所有损失函数都可以通过在特定选择此术语的情况下应用机器学习原理获得。这两个观点都很有趣，我们敦促读者在阅读本书的过程中牢记它们。

3.3 贝叶斯学习

我们在这里讨论了概率公式的进一步推广，称为贝叶斯神经网络（BNNs），这在文献中引起了兴趣。我们只提供一般想法，并建议读者参考众多深入教程之一，例如[JLB+22]，以获取更多详细信息。

通过设计一个概率函数 $p(y \mid f(x))$ 而不是直接设计 $f(x)$，我们可以处理对多个预测感兴趣的情况（即，概率函数具有多个峰值）。然而，我们的方法仍然从所有可能函数的空间中返回一个 *single function* $f(x)$，而可能在整个模型空间中存在多个有效的参数化。在这种情况下，能够访问所有这些参数化对于更准确的预测可能是有用的。

再次，我们可以通过设计来实现这一目标

另一个概率分布，然后让概率规则引导我们。由于我们现在计划获得所有可能函数的分布，我们首先定义一个先验概率分布 $p(f)$ 在所有可能的函数上（回想一下，在实践中 f 由一组有限的参数描述，在这种情况下，先验 $p(f)$ 变成对这些权重的先验）。例如，我们将看到在许多情况下，具有较小范数的函数更受青睐（因为它们更稳定），在这种情况下，我们可以为 f 的某个范数 $\|f\|$ 定义一个先验 $p(f) \propto \frac{1}{\|f\|}$。

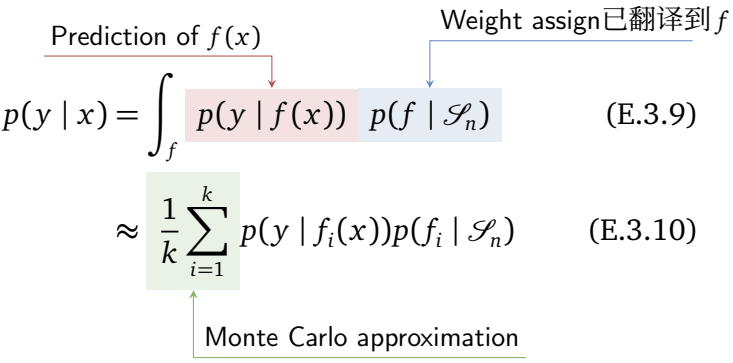
一旦观察到数据集，f上的概率会根据先验和似然而改变，更新由贝叶斯定理给出：

Prior (*before* observing the dataset)

$$p(f \mid \mathcal{S}_n) = \frac{p(\mathcal{S}_n \mid f)\, p(f)}{p(\mathcal{S}_n)} \tag{E.3.8}$$

Posterior (*after* observing the dataset)

术语 $p(f \mid \mathcal{S}_n)$ 被称为后验分布函数，而分母中的术语 $p(\mathcal{S}_n)$ 被称为证据，它需要确保后验得到适当的归一化。现在假设我们能够访问后验。与之前不同，分布可以编码对多个函数 f 的偏好，这可能提供更好的预测能力。给定一个输入 x，我们可以通过平均所有基于其后验权重的可能模型来进行预测：



在（E.3.10）中，我们通过从后验分布 $f_k \sim p(f \mid \mathscr{S}_n)$ 中抽取 k 个随机样本的蒙特卡洛平均来近似积分。这种设置的总体美感被这样一个事实所破坏，即后验通常无法以闭式形式计算，除非对先验和似然 [Bis06] 进行非常特定的选择。如果没有这一点，人们被迫使用近似解，无论是通过马尔可夫链蒙特卡洛还是通过变分推理 [JLB+22]。我们将在9.3.1节中看到一个对模型参数进行贝叶斯处理的例子，称为蒙特卡洛dropout。

我们在本节结束前对后验概率的两个有趣事实进行评论。首先，假设我们只对具有最高后验密度的函数感兴趣。在这种情况下，证据项可以忽略，解可以分解为两个独立的项：

$$f^* = \arg\max_f p(\mathscr{S}_n \mid f)p(f) = \tag{E.3.11}$$

$$\arg\max_f \left\{ \log p(\mathscr{S}_n \mid f) + \log p(f) \right\} \tag{E.3.12}$$

这被称为最大后验概率（MAP）解。如果所有函数在先验上具有相同的权重（即，$p(f)$ 在函数空间上是均匀的），那么第二项是一个常数，问题简化为最大似然解。然而，在一般情况下，MAP解将对偏离我们先验分布太多的函数施加惩罚。我们将看到，这是一个有用的想法来对抗过拟合并对函数 f 施加特定约束。术语 $\log p(f)$ 通常被称为函数空间的正则化器，因为它将解推向由先验分布定义的吸引域.[6]

其次，完整的贝叶斯处理提供了一种简单的方法来整合新数据，例如来自同一分布的新数据集 $\mathscr{S}_n'$。为此，我们将 (E.3.8) 中的先验函数替换为我们对数据集第一部分计算的后验分布，这现在代表了对 f 可能值的起始假设，该假设通过查看新数据得到更新.[7] 这可以缓解在线训练模型时的问题，最值得注意的是所谓的 *catastrophic forgetting* 信息 [KPR+17] 问题。

[6]The difference between maximum likelihood and maximum a posteriori solutions is loosely connected to the difference between the **frequentist** and **Bayesian** interpretation of probability [Bis06], i.e., probabilities as frequency of events or probabilities as a measure of uncertainty. From a very high-level point of view, ML sees the parameters as an unknown fixed term and the data as a random sample, while a Bayesian treatment sees the data as fixed and the parameters as random variables.

[7]Think of the original prior function as the distribution on f after having observed an initial *empty set* of values.

4 | 线性模型

关于本章

编程是通过选择适当的原始操作序列来解决任务。通过类比，构建模型是通过选择正确的 *differentiable* 块序列。在本章中，我们介绍了最简单的块，即线性模型，它假设输入通过加权平均对输出进行加性作用。从某种意义上说，所有可微模型都是线性块的智能变体和组合。

4.1 最小二乘回归

总结前一章，可以通过选择输入类型 x、输出类型 y、模型 f 和损失函数 l 来定义一个监督学习问题。在本章中，我们考虑所有这些可能的最简单选择，即：

- 输入是一个向量 x {v*}，对应一组特征（例如，{v*} 银行客户的个人特征）。我们使用标量 {v*} 作为 {v*} 的简称来

表 T.4.1: *Basic shapes to remember for this chapter. For uniformity, we will use the same letters as much as possible throughout the book.*

n	**size of the dataset**
c	**features**
m	**classes**

表示与以下章节一致的特性数量。

- 输出是一个单个实数值 $y \in \mathbb{R}$。在无约束的情况下，我们称这为回归任务。如果 y 只能取 m 种可能值之一，即 $y \in \{1, \ldots, m\}$，我们称这为分类任务。在 $m = 2$ 的特殊情况下，我们称这为二元分类任务。

- 我们取 f 为线性模型，在某些情况下为我们提供简单的闭式解，最显著的是最小二乘回归（第4.1.2节）。

基本形状总结在表T.4.1中。我们首先讨论回归情况下的损失选择。我们从回归情况开始，因为，正如我们稍后所示，通过少量修改回归情况，就可以解决分类问题。

4.1.1 平方损失及其变体

寻找回归损失相对简单，因为预测输出 $e = (\hat{y} - y)$ 与真实期望输出 y 之间的预测误差是定义良好的目标，它是模型输出的连续函数，单调递减。由于在

一般我们不在乎预测误差的符号，一个常见的选择是平方损失：

$$l(\hat{y}, y) = (\hat{y} - y)^2 \tag{E.4.1}$$

在此及以下内容中，我们使用符号 $\hat{y}$ 表示模型的预测。正如我们将看到的，使用（E.4.1）为我们提供了解决方案的多项好处。其中之一是，平方损失的梯度是模型输出的线性函数，这使得我们可以以闭式形式求解最优解。

回忆最大似然原理（第3.2.2节），通过假设模型的输出遵循以$f(\mathbf{x})$为中心、具有常数方差σ^2的高斯分布，可以得到平方损失：

$$p(y \mid f(\mathbf{x})) = \mathcal{N}(y \mid f(\mathbf{x}), \sigma^2)$$

在这种情况下，单个点的对数似然可以写成：[1]

$$\log(p(y \mid f(\mathbf{x}), \sigma^2)) = -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{1}{2\sigma^2}(y - f(\mathbf{x}))^2 \tag{E.4.2}$$

最小化（E.4.2）对于f，我们看到右侧的前两个项是常数，第三个项恢复为平方损失。对于 σ^2 的最小化可以独立于f的优化进行，有一个简单的闭式解（见下文，方程（E.4.9））。

[1] Recalling that $\log(ab) = \log(a) + \log(b)$ and $\log(a^b) = b\log(a)$.

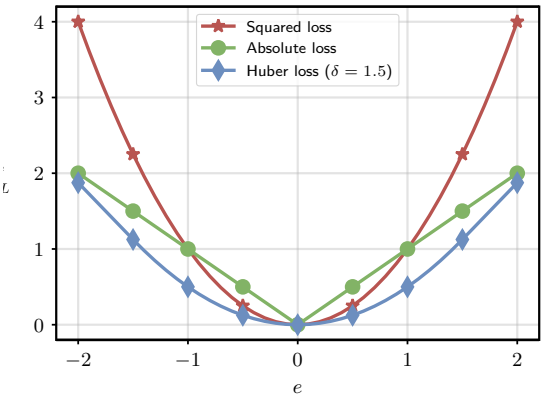


图 F.4.1：*Visualization of the squared loss, the absolute loss, and the Huber loss with respect to the prediction error* $e = (\hat{y} - y)$.

平方损失的变体也很容易想到。例如，平方损失的一个缺点是，更高的误差将以与误差成二次增长的力量进行惩罚，这可能会对异常值产生不适当的影响，即错误标记严重的点。可以减少异常值影响的其它选择包括绝对值损失 $l(\hat{y}、y) = |\hat{y} - y|$ 或 Huber 损失（平方损失和绝对损失的组合）：

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq 1 \\ \left(|y - \hat{y}| - \frac{1}{2}\right) & \text{otherwise} \end{cases} \tag{E.4.3}$$

这是关于0误差邻近度的二次函数，否则是线性的（添加了$-\frac{1}{2}$项以确保连续性）。参见图F.4.1以可视化这些损失与预测误差的关系。

绝对损失在我们的上下文中似乎不是一个合适的选择，因为它在0处由于绝对值的存在有一个不可微的点。我们稍后将会看到，具有 {v*} 的函数

一个（或少数几个）这种形式的点实际上并不真正成问题。从数学上讲，它们可以通过子梯度（导数的一个轻微推广）的概念来处理。实际上，你可以想象，如果我们从一个随机的初始化开始，梯度下降永远不会以完美的精度达到这些点，对于任何 $\varepsilon > 0$，$|\varepsilon|$ 的导数总是定义的。

4.1.2 最小二乘模型

有了损失函数在手，我们考虑以下模型（一个线性模型）来完成我们第一个监督学习问题的规范。

> 定义 D.4.1（线性模型）
>
> *A **linear model** on an input* x *is defined as:*
>
> $$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$$
>
> *where* $\mathrm{w} \sim (c)$ *and* $b \in \mathbb{R}$ *(the bias) are trainable parameters.*

直觉是模型为每个输入特征x_i分配一个固定的权重w_i，并通过线性求和给定输入x的所有效应来提供预测，当$\mathrm{x} = 0$时，恢复到等于b的默认预测。从几何上看，该模型为$d = 1$定义了一条线，为$d = 2$定义了一个平面，为$d > 1$定义了一个通用超平面。从符号的角度来看，我们有时可以通过假设1作为x的最后一个特征来避免写出偏差项：

$$f\left(\begin{bmatrix}\mathbf{x}\\1\end{bmatrix}\right) = \mathbf{w}^\top \begin{bmatrix}\mathbf{x}\\1\end{bmatrix} = \mathbf{w}_{1:c}^\top \mathbf{x} + w_{c+1}$$

将线性模型、平方损失和经验风险最小化问题相结合，我们得到最小二乘优化问题。

定义 D.4.2（最小二乘法）

*The **least-squares** optimization problem is given by:*

$$\mathbf{w}^*, b^* = \arg\min_{\mathbf{w},b} \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \mathbf{x}_i - b\right)^2 \qquad \text{(E.4.4)}$$

在继续分析这个问题之前，我们将最小二乘法重写为仅涉及矩阵运算（矩阵乘法和范数）的向量形式。这很有用，因为，如前所述，现代用于训练可微分模型的代码是围绕 n-维数组构建的，并具有优化硬件来执行这些矩阵运算。为此，我们首先将训练集的所有输入和输出堆叠成一个输入矩阵：

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{bmatrix} \sim (n, c)$$

并且一个类似的输出向量 $y = [y_1, \ldots, y_n]^\top$。我们可以将批处理模型输出（值为迷你批次的模型输出）写成：

$$f(\mathbf{X}) = \mathbf{X}\mathbf{w} + \mathbf{1}b \qquad \text{(E.4.5)}$$

相同的偏差 b 对所有 n 预测

```
def linear_model(w: Float[Tensor, "c"],
                 b: Float,
                 X: Float[Tensor, "n c"])
                 -> Float[Tensor, "n"]:
  return X @ w + b
```

盒子 C.4.1: *Computing a batched linear model as in* (E.4.5). *For clarity, we are showing the array dimensions as type hints using* jaxtyping (*https://docs.kidger.site/jaxtyping/*).

等式（E.4.5）几乎可以逐行在代码中复制 - 请参阅盒C.4.1中的PyTorch示例。

目前只有边缘兴趣，但后来会更有重要性，我们注意到输入矩阵和输出向量的行排序在根本上是任意的，即在某种意义上，交换它们的行只会导致$f(\mathrm{X})$的行发生相应的排列。这是称为排列等变性的现象的一个简单例子，它将在以后发挥更加重要的作用。

最小二乘优化问题以向量形式写出变为：

$$\mathrm{LS}(\mathbf{w}, b) = \frac{1}{n} \|\mathbf{y} - \mathbf{X}\mathbf{w} - \mathbf{1}b\|^2 \tag{E.4.6}$$

我们在哪里回忆到向量的范数定义为 $\|\mathrm{e}\|^2 = \sum_i e_i^2$ 。

4.1.3 求解最小二乘问题

为了通过梯度下降法解决最小二乘问题，我们需要其梯度的方程。尽管

```
from torch import linalgdef ls_solve(X: Float[Tensor, "n c"], y: Float[Tensor, "n"], numerically_stable = True) \ -> Float[Tensor, "c"]: # 显式解法    if not numerically_stable:        return linalg.inv(X.T @ X) @ X.T @ y    else:        return linalg.solve(X.T @ X, X.T @ y)
```

盒子 C.4.2: *Solving the least-squares problem with the closed form solution. The numerically stable variant calls a solver specialized for systems of linear equations.*

我们将很快开发一个通用的算法框架来自动计算这些梯度（第6章），在这个简单场景中查看梯度本身是有教育意义的。忽略偏差（如上所述的原因，我们可以将其纳入权重向量），以及其他常数项，我们有：

$$\nabla \mathrm{LS}(\mathbf{w}) = \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$$

LS问题在模型权重上是凸的，可以通过注意到方程描述了权重空间中的抛物面（一个二次函数）来非正式地理解。全局最小值由以下方程描述：

$$\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0 \;\; \Rightarrow \;\; \mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y}$$

这些被称为正则方程。重要的是，正则方程描述了一个线性方程组在

w,[2] 表示在适当的条件下（对应于矩阵 $\mathrm{X}^\top\mathrm{X}$ 的可逆性），我们可以求解最优解如下：

$$\mathbf{w}_* = \left(\mathbf{X}^\top\mathbf{X}\right)^{-1}\mathbf{X}^\top\mathbf{y} \tag{E.4.7}$$

信息碎片

矩阵$\mathrm{X}^\dagger = \left(\mathrm{X}^\top\mathrm{X}\right)^{-1}\mathrm{X}^\top$被称为非方阵X的伪逆（或摩尔-彭罗斯逆），因为$\mathrm{X}^\dagger\mathrm{X} = \mathrm{I}$。在(E.4.7)中进行求逆并不总是可能的：例如，如果一个特征是另一个特征的标量倍数，矩阵X没有满秩（这被称为共线性）。最后，请注意，最小二乘模型的预测可以写成$\hat{\mathrm{y}} = \mathrm{My}$，其中$\mathrm{M} = \mathrm{X}\mathrm{X}^\dagger$。因此，最小二乘也可以解释为执行训练标签的加权平均，其中权重由X诱导的列空间的投影给出。这被称为最小二乘的对偶形式。对偶形式提供了模型内在的调试级别，因为它们允许通过检查相应的对偶权重[ICS22]来检查哪些输入对预测最为相关。

这是唯一一种我们可以用封闭形式表达最优解的情况，将这个解与梯度下降解进行比较是有教育意义的。为此，我们在Box C.4.2中展示了使用(E.4.7)以封闭形式求解最小二乘法的一个例子，在Box C.4.3中展示了相应的梯度下降公式。在损失函数的典型演化中

[2] That is, we can write them as $\mathbf{A}\mathbf{w} = \mathbf{b}$, with $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ and $\mathbf{b} = \mathbf{X}^\top\mathbf{y}$.

```
def ls_gd(X: Float[Tensor, "n c"],
          y: Float[Tensor, "n 1"],
          lr=1e-3) \
          -> Float[Tensor, "c"]:
    # Initializing the parameters
    w = torch.randn((X.shape[1], 1))

    # Fixed number of iterations
    for i in range(15000):
      # Note the sign (why?)
      w = w + lr * X.T @ (y - X @ w)

    return w
```

盒子C.4.3: *Same task as Box C.4.2, solved with a naive implementation of gradient descent with a fixed learning rate that defaults to* $\eta = 0.001$.

后者在图F.4.2中绘制。由于我们选择了一个非常小的学习率，梯度下降过程中的每一步都提供了稳定的损失下降，直到收敛。实际上，可以通过数值方法检查收敛，例如，通过评估两次迭代之间范数的差异对于某个数值阈值$\varepsilon > 0$:

$$\|\mathbf{w}_{t+1} - \mathbf{w}_t\|^2 < \varepsilon \tag{E.4.8}$$

我们将会看到，理解更复杂的模型何时收敛将是一个更微妙的工作。

考虑到(E.4.2)中的高斯对数似然，一旦训练了权重，我们还可以对σ^2进行优化，得到:

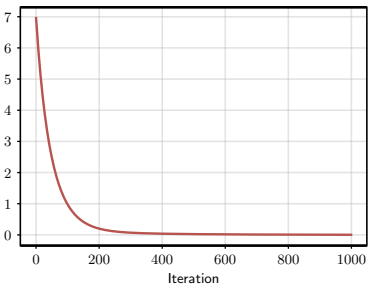


图 F.4.2：*An example of running code from Box 1 6.4.2, where the data is composed of n = 60 points drawn from a linear model* $w^\top x + \varepsilon$, *with* $w_i \sim \mathcal{N}(0, 1)$ *and* $\varepsilon \sim \mathcal{N}(0, 0.01)$. *ils apart, note the smooth descent: each step provides a decrease in loss.*

$$\sigma_*^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{w}_*^\top \mathbf{x}_i)^2 \,. \tag{E.4.9}$$

该模型方差恒定（根据定义）且由训练数据上的平均预测误差平方给出，具有直观意义。通过假设方差本身由模型预测（异方差模型），可以获得更复杂的概率模型，参见[Bis06]。

4.1.4 一些计算考虑事项

即使可以计算逆矩阵，解的质量将取决于$X^\top X$的条件数，对于条件较差的矩阵，可能会出现大的数值误差.[3] 此外，求解(E.4.7)的计算成本可能过高。矩阵求逆的规模大致为$\mathcal{O}(c^3)$。至于矩阵乘法，

[3]The condition number of a matrix $\mathbf{A}$ is defined as $\kappa(\mathbf{A}) = \|\mathbf{A}\|\|\mathbf{A}^{-1}\|$ for some choice of matrix norm $\|\bullet\|$. Large conditions number can make the inversion difficult, especially if the floating-point precision is not high.

算法需要将一个 $c \times n$ 矩阵与另一个 $n \times c$ 矩阵相乘，以及一个 $c \times c$ 矩阵与一个 $c \times n$ 矩阵之间的乘法。这两个操作都将按 $\mathcal{O}(c^2 n)$ 规模扩展。

通常，我们总是更喜欢在特征维度 c 和批量大小 n 上都呈线性缩放的算法，因为超线性算法会很快变得不切实际（例如，一个大小为 1024 × 1024 的 32 个 RGB 图像有 $c \approx 1e^7$）。通过正确计算乘法顺序，我们可以避免梯度方程中的二次复杂度，即首先计算矩阵-向量积 Xw。因此，纯梯度下降在 c 和 n 上都是线性的，但只有当实现时采取适当的注意：推广这个想法是反向模式自动微分（也称为反向传播）发展的基本洞察（第 6.3 节）。

4.1.5 正则化最小二乘解

再次审视逆运算的潜在不稳定性，假设我们有一个矩阵几乎奇异的数据库，但我们仍然希望进行闭式解。在这种情况下，可以稍微修改问题，以实现一个“尽可能接近”原始解的解，同时又是可计算的。例如，一个已知的技巧是将一个小的倍数，$\lambda > 0$，的恒等矩阵加到要逆的矩阵上：

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

这使矩阵变得更加“对角”，并提高了其条件数。回溯到原始问题，我们注意到这是闭式解

关于一个修改后的优化问题：

$$\text{LS-Reg}(\mathbf{w}) = \frac{2}{n}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

这个问题被称为正则化最小二乘法（或岭回归），损失中的红色部分是ℓ_2-正则化（或更一般地，正则化）的一个实例。请注意，正则化不依赖于数据集，因为它仅仅编码了对某种类型解的偏好（在这种情况下，低范数权重），而偏好的强度本身由超参数λ定义。从贝叶斯视角（第3.3节）来看，正则化最小二乘法对应于在权重上定义一个以零为中心且方差恒定的高斯先验时的MAP解。

4.2 线性分类模型

我们现在转向分类，其中包含$y_i \in \{1, \ldots, m\}$，其中m定义了类别的数量。正如我们稍后将会看到的，这是一个具有广泛影响力的难题，涵盖了计算机视觉（例如，图像分类）和自然语言处理（例如，下一个标记预测）中的各种任务。我们可以通过相对于回归情况进行轻微的变体来解决这个问题。

虽然我们可以通过直接对整数值y_i进行回归来解决这个任务，但考虑为什么这可能不是一个好主意是有教育意义的。首先，模型直接预测一个整数值是困难的，因为这需要一些阈值处理，这几乎会使其梯度在所有地方都变为零。相反，我们可以回归一个实值$\tilde{y}_i \in [1、m]$，在从 1 到 m (的区间内，正如我们将要展示的，将模型的输出限制在区间内可以

完成得容易）。在推理过程中，给定输出 $\hat{y}_i = f(\mathrm{x}_i)$，我们通过四舍五入将其映射回原始域：

$$\text{Predicted class} = \text{round}(\hat{y}_i)$$

例如，$\hat{y}_i = 1.3$ 将被映射到类别 1，而 $\hat{y}_i = 3.7$ 将被映射到类别 4。请注意，这仅是在推理时才能进行的值的后处理。这不是一个好的建模选择的原因是，我们引入了类别的虚假排序，这可能会被模型本身利用，其中类别 2 比 4 更接近类别 3。我们可以通过将 y 转换为经典的独热编码版本来避免这种情况，我们将其表示为 $\mathrm{y}^{\text{oh}} \sim \text{Binary}(m)$：

$$[\mathbf{y}^{\text{oh}}]_j = \begin{cases} 1 & \text{if } y = j \\ 0 & \text{otherwise} \end{cases}$$

例如，在三个类别的案例中，我们会有 $\mathrm{y}^{\text{oh}} = [1\,0\,0]^\top$ 用于类别 1，$\mathrm{y}^{\text{oh}} = [0\,1\,0]^\top$ 用于类别 2，以及 $\mathrm{y}^{\text{oh}} = [0\,0\,1]^\top$ 用于类别 3（这种表示方式应该对有一定机器学习背景的读者来说很熟悉，因为它是对分类变量的标准表示）。

单热向量是无序的，在给定的两个通用输出y_1^{oh}和y_2^{oh}的情况下，它们的欧几里得距离要么是0（同一类别）要么是$\sqrt{2}$（不同类别）。虽然我们可以在单热编码的输出上直接执行多值回归，在这种情况下，均方误差被称为Brier分数，但我们下面将展示一个更好且更优雅的解决方案，即逻辑回归。

4.2.1 软化函数

我们不能训练一个模型直接预测一个独热编码向量（出于上述相同的原因），但我们可以通过轻微的放松来实现类似的效果。为此，我们重新引入概率单纯形。

> 定义 D.4.3（概率单纯形）
>
> *The* ***probability simplex*** Δ_n *is the set of vectors* x {v*} {v*}
>
> $$x_i \geq 0, \sum_i x_i = 1$$

几何上，你可以将单热向量的集合想象为 n-维多边形的顶点，而单纯形则是其凸包：单纯形内部，如 [0.2、0.05、0.75] 的值并不精确对应于一个顶点，但它们允许进行梯度下降，因为我们可以在多边形内部平滑移动。给定一个值 $x \in \Delta_n$，我们可以将其投影到最近的顶点（预测类别）上，如下：

$$\arg\max_i \{\mathbf{x}_i\}$$

根据名称的含义，我们可以将单纯形内的值解释为概率分布，而投影到最近的顶点则是找到分布中的众数（最可能的类别）。在这种解释中，一个独热编码向量是一个“特殊情况”，其中所有概率质量都集中在单个类别上（我们知道这是正确的类别）。

为了预测这个单纯形中的值，我们需要两个

对线性模型（E.4.4）的修改：首先，我们需要同时预测一个整个向量；其次，我们需要约束输出位于单纯形中。作为第一步，我们将线性模型修改为预测一个 m- 维向量：

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \tag{E.4.10}$$

在 W $\sim$ (m, c) 可以解释为并行运行的 m 线性回归模型，并且 b $\sim (m)$。此输出不受约束，且不保证在单纯形内。逻辑回归的核心思想是将 (E.4.10) 中的线性模型与一个简单、参数无关的变换相结合，该变换将投影到单纯形内，称为 softmax 函数。

定义 D.4.4（Softmax 函数）

*The **softmax** function is defined for a generic vector* x $\sim (m)$ *as:*

$$[\text{softmax}(\mathbf{x})]_i = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{E.4.11}$$

为了理解正在发生的事情，我们通过引入两个中间项来分解(E.4.11)中的项。首先，softmax的分子通过指数运算将每个数字转换为正值 h_i：

$$h_i = \exp(x_i) \tag{E.4.12}$$

其次，我们计算一个归一化因子 Z，作为这些新（非负）值的总和：

$$Z = \sum_j h_j \tag{E.4.13}$$

softmax的输出由h_i除以Z得到，从而确保新的值之和为1：

$$y_i = \frac{h_i}{Z} \tag{E.4.14}$$

另一个视角来自考虑softmax的一个更通用版本，其中我们添加了一个额外的超参数$\tau > 0$，称为温度：

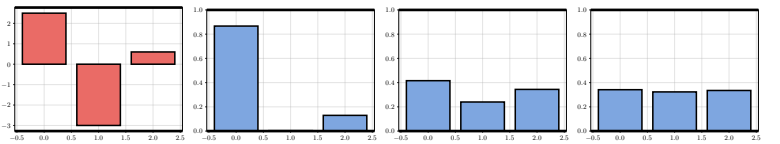
$$\text{softmax}(\mathbf{x}; \tau) = \text{softmax}(\mathbf{x}/\tau)$$

softmax在所有τ的值中保持x_i的值的相对顺序，但它们的绝对距离根据温度增加或减少。特别是，我们有以下两种极限情况：

$$\lim_{\tau \to \infty} \text{softmax}(\mathbf{x}; \tau) = 1/c \tag{E.4.15}$$

$$\lim_{\tau \to 0} \text{softmax}(\mathbf{x}; \tau) = \arg\max_i \mathbf{x} \tag{E.4.16}$$

对于无限温度，相对距离将消失，输出回归到均匀分布。相反，在0温度下，softmax回归到（难以微分的）argmax操作。因此，softmax可以看作是argmax的一个简单可微近似，更好的名字应该是softargmax。然而，我们在这里将保留最标准的名字。参见



(a) *Inputs* **(b)** $\tau = 1$ **(c)** $\tau = 10$ **(d)** $\tau = 100$

图 F.4.3：*Example of softmax applied to a three-dimensional vector (a), with temperature set to 1 (b), 10 (c), and 100 (d). As the temperature increases, the output converges to a uniform distribution. Note that inputs can be both positive or negative, but the outputs of the softmax are always constrained in* $[0, 1]$.

图 F.4.3 用于展示在不同温度值下对通用三维向量应用softmax的可视化。

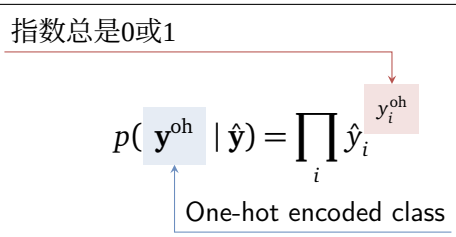
4.2.2 逻辑回归模型

我们可以通过将(E.4.11)中的softmax与(E.4.10)中的线性模型相结合，来总结我们之前的讨论，从而获得一个用于分类的线性模型：

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

预归一化输出 h = Wx + b 被称为模型的 logits，这个名字将在下一节中更详细地讨论。

我们现在需要一个损失函数。从第3.2.2节中的概率观点出发，因为我们的输出被限制在概率单纯形中，所以我们将其用作分类分布的参数：



计算此情况下的最大似然解（试试看）得到交叉熵损失。

定义 D.4.5（交叉熵损失）

*The **cross-entropy** loss function between* y^{oh} *and* $\hat{\mathrm{y}}$ *is given by:*

$$\text{CE}(\mathbf{y}^{\text{oh}}, \hat{\mathbf{y}}) = -\sum_i y_i^{\text{oh}} \log(\hat{y}_i) \tag{E.4.17}$$

损失也可以表示为两个概率分布之间的KL散度。虽然一开始可能不太直观，但通过注意到只有y^{oh}的一个值将不为零，对应于真实的类别$y = \arg\max_i \{y_i^{\text{oh}}\}$，我们可以将损失简化为：

$$\text{CE}(y, \hat{\mathbf{y}}) = -\log(\hat{y}_y) \tag{E.4.18}$$

概率分配给真实类别

从（E.4.18）中，我们看到最小化CE损失的效果是最大化对应于真实类别的输出概率。这是因为，由于softmax中的分母，任何输出项的增加都会自动导致其他项的减少。将所有这些放在一起，我们得到逻辑回归优化问题：

$$\text{LR}(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^{n} \text{CE}\left(\mathbf{y}_i^{\text{oh}}, \text{softmax}(\mathbf{W}\mathbf{x}_i + \mathbf{b})\right).$$

与最小二乘法不同，我们不能再计算闭式解了，但我们仍然可以使用梯度下降法进行计算。我们将在下一节中展示这种情况下的梯度示例，并在第6.3节中介绍一种通用的计算梯度技术。

4.3 更多关于分类

4.3.1 二元分类

考虑现在 $m = 2$ 的具体情况。在这种情况下，我们有 $y \in \{0, 1\}$，问题简化为二元分类，有时也称为概念学习（因为我们需要学习某个二元“概念”是否存在于输入中）。使用标准逻辑回归，这可以通过具有两个输出的函数来建模。然而，由于softmax分母，逻辑回归的最后输出总是冗余的，因为已知输出必须求和为1：

$$f_m(\mathbf{x}) = \sum_{i=1}^{m-1} f_i(\mathbf{x})$$

基于此，我们可以通过考虑一个具有单个输出 $f(\mathrm{x}) \in [0, 1]$ 的标量模型来略微简化公式。

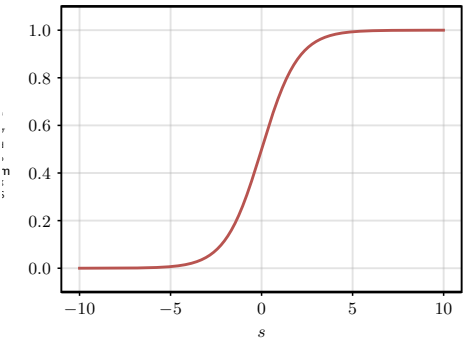


图 F.4.4:
Plot of the sigmoid function. Note that $\sigma(0) = 0.5$.

$$\text{Predicted class} = \text{round}(f(\mathbf{x})) = \begin{cases} 0 & \text{if } f(\mathbf{x}) \leq 0.5 \\ 1 & \text{otherwise} \end{cases}$$

为了在 [0, 1] 中实现所需的归一化，两个值softmax的第一个输出可以重写为$\frac{\exp(x_1)}{1+\exp(x_1)}$，我们可以通过将两边除以 exp (x_1) 进一步简化它。结果是sigmoid函数。

定义 D.4.6（Sigmoid 函数）

The ***sigmoid*** *function* $\sigma(s)\ \mathbb{R} \to [0, 1]$ *is given by:*

$$\sigma(s) = \frac{1}{1 + \exp(-s)}$$

Sigmoid提供了一种通用变换，将任何实数值投影到[0, 1]区间（两个极端值仅以渐近方式达到）。其图形如图F.4.4所示。

二进制逻辑回归模型是通过将一维线性模型与Sigmoid函数相结合得到的

输出重缩放：

$$f(\mathbf{x}) = \sigma\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

交叉熵同样简化为：

$$\text{CE}(\hat{y}, y) = \underbrace{-y \log(\hat{y})}_{\text{Loss for class 1}} \underbrace{-(1-y)\log(1-\hat{y})}_{\text{Loss for class 2}} \tag{E.4.19}$$

因此，在二分类情况下，我们可以用两种等效的方法解决这个问题：(a) 使用标准softmax的二元模型，或(b) 使用sigmoid输出变换的简化单值输出。

作为一个有趣的旁注，考虑二元逻辑回归模型相对于w的梯度（对于标准多类情况也可以写出类似的梯度）：

$$\nabla \text{CE}(f(\mathbf{x}), y) = (f(\mathbf{x}) - y)\mathbf{x}$$

注意与标准线性模型回归梯度的相似性。这种相似性可以通过将我们的模型重新写为以下形式来进一步理解：

$$\underbrace{\mathbf{w}^\top \mathbf{x} + b}_{\text{Logits}} = \underbrace{\log\left(\frac{y}{1-y}\right)}_{\text{Sigmoid inverse: } \sigma^{-1}(y)} \tag{E.4.20}$$

这解释了为什么我们称这个模型为

“线性模型”用于分类：我们可以将其重写为关于输出非线性变换的纯线性模型（在这种情况下，sigmoid的逆，也称为对数几率）。事实上，逻辑回归模型是更广泛模型家族的一部分，该家族扩展了这一想法，称为广义线性模型。对于好奇的读者，名称*logits*可以在这个上下文中理解为与概率单位函数相关.[4]

4.3.2 对数和指数技巧

这是一个更技术性的小节，它阐明了我们迄今为止所描述的实现方面的内容。观察像TensorFlow或PyTorch这样的框架，我们可以找到多个现有的交叉熵损失的实现，这取决于输出是描述为整数还是作为one-hot编码向量。这很容易理解，因为我们已经看到我们可以在这两种情况下对交叉熵损失进行公式化。然而，我们还可以找到接受logits而不是softmax归一化输出的变体，如Box C.4.4所示。

为了理解为什么我们需要这个，考虑交叉熵的 i-th 项，从对数几率 p 的角度来考虑：

$$-\log\left(\frac{\exp p_i}{\sum_j \exp p_j}\right).$$

此术语可能导致几个数值问题，尤其是由于（可能无界的）logits与指数运算之间的相互作用。为了解决这个问题，我们首先重写

[4] https://en.wikipedia.org/wiki/Probit

从torch.nn导入functional作为F
二元交叉熵 F.binary_cross_entropy # 接受logits的二进制交叉熵 F.binary_cross_entropy_with_logits # 从logits开始的交叉熵 F.cross_entropy # 以log f(x)为输入的交叉熵 F.nll_loss

盒子C.4.4: *Cross entropy losses in PyTorch. Some losses are only defined starting from the logits of the model, instead of the post-softmax output. These are the functional variants of the losses - equivalent object-oriented variants are also present in most frameworks.*

它作为:

$$-\log\left(\frac{\exp p_i}{\sum_j \exp p_j}\right) = -p_i + \underbrace{\log\left(\sum_j \exp p_j\right)}_{\triangleq \text{logsumexp}(\mathbf{p})}$$

第一个项不受不稳定性影响，而第二个项（logits的对数和）是整个logits向量的函数，并且可以证明在以下意义上对给定的标量 $c \geq 0$ 保持不变：[5]

$$\text{logsumexp}(\mathbf{p}) = \text{logsumexp}(\mathbf{p} - c) + c$$

注意 ∇softmax(•) = 对数和指数之间通过取 c = max(p)，我们可以通过将最大logit值限制在0来防止数值问题。然而，这只有在我们可以访问原始logit的情况下才可能，这就是为什么数值稳定的交叉熵变体需要将它们作为输入的原因。这会产生少量的

[5] https://gregorygundersen.com/blog/2020/02/09/log-sum-exp/

歧义，因为在模型或损失函数中现在都可以包含softmax。

4.3.3 校准和分类

我们通过简要讨论分类器的校准这一重要主题来结束本章。为了理解它，考虑以下事实：尽管我们的模型提供了可能类别的整个分布，但我们的训练标准仅针对真实类别的最大化。因此，以下句子是合理的：

$$\text{预测的 } f(\mathrm{x}) \text{ 类别为 } \arg\max_i [f(\mathrm{x})]_i$$

相反，这个更一般的句子可能不正确：

$$\mathrm{x} \text{属于类别} i \text{的概率是} [f(\mathrm{x})]_i。$$

当网络的置信度得分与给定预测正确的概率相匹配时，我们说网络的输出是校准的。

> 定义 D.4.7（校准）
>
> *A classification model $f(\mathrm{x})$ giving in output the class probabilities is said to be calibrated if the following holds for any possible prediction:*
>
> $$[f(\mathbf{x})]_i = p(y = i \mid \mathbf{x})$$

尽管交叉熵应该在无限数据 [HTF09] 的极限下恢复对无限制模型类别的条件概率分布，但在实践中，两者之间的不匹配可能很高 [BGHN24]，尤其是对于我们将要介绍的后面的更复杂模型。

要理解准确性和校准之间的区别，考虑以下两种情况。首先，考虑一个具有完美准确率的二元分类模型，但总是以0.8的置信度预测真实类别。在这种情况下，模型在预测上显然是*underconfident*，因为通过查看置信度，我们可能假设其中20%是错误的。其次，考虑一个具有完美平衡类别的4类问题，模型总是预测[0.25，025，0.25，0.25]。在这种情况下，模型完全校准，但从准确性的角度来看是无用的。

访问校准模型在预测可能具有不同成本的情况下非常重要。这可以通过定义一个所谓的 *cost matrix* 来形式化，为任何被预测为类别 i 的输入分配成本 C_{ij}。一个标准示例是具有表 T.4.2 中所示成本矩阵的二分类问题。

表 T.4.2：*Example of cost matrix for a classification problem having asymmetric costs of misclassification.*

	True class 0	True class 1
Predicted class 0	0	10
Predicted class 1	1	0

我们可以将表T.4.2解释如下：做出正确的预测不会产生成本，而做出错误的阴性错误（0而不是1）比做出错误的阳性错误成本高10倍。例如，在医疗诊断中，错误的阴性错误比错误的阳性错误要严重得多，进一步的测试可能会纠正错误。校准模型可以帮助我们更好地估计其部署的平均风险，并微调我们对于错误阳性错误和错误阴性错误的平衡。

为了看到这一点，用C ∼ (m， m) 表示多类问题的通用成本矩阵（如表T.4.2中的2 × 2矩阵）。合理的选择是选择一个类别，该类别基于我们模型分配的分数最小化预期成本：

$$\arg\min_i \sum_{j=1} C_{ij}[f(\mathrm{x})]_j$$

如果 $C_{ij} =$ 为 1 当且仅当 $i \neq j$ 为 1，否则为 0，这简化为选择 f 的 argmax，但对于一般成本矩阵，预测类别的选择将受到具体错误相对成本的影响。这是决策理论 [Bis06] 的一个简单例子。

4.3.4 估计校准误差

为了估计模型是否校准，我们可以将其预测进行分组，并比较其在每个分组中的校准与准确性。为此，假设我们将区间 $[0, 1]$ 分成 b 个等距的分组，每个分组的大小为 $1/b$ 。取一个大小为 n 的验证集，并用 $\mathscr{B}_i$ 表示置信度落在分组 i 中的元素。对于每个分组，我们可以进一步计算模型的平均置信度 p_i（这将在分组的中间，大约），以及平均准确性 a_i。在直方图上绘制对 (a_i, p_i) 的集合称为可靠性图，如图 F.4.5 所示。为了有一个单一的标量校准指标，我们可以使用，例如，期望校准误差（ECE）：

$$\mathrm{ECE} = \sum_i \frac{|\mathscr{B}_i|}{n} |a_i - p_i| \tag{E.4.21}$$

Calibration for bin i

Fraction falling into bin i

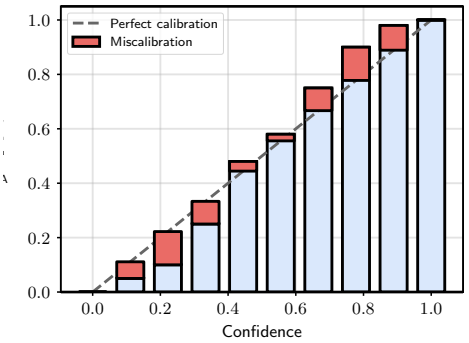


图 F.4.5：*An example of reliability plot with* $b = 10$ *bins. The blue bars show the average accuracy of the model on that bin, while the red bars show the miscalibration for the bin, which can be either under-confident (below the diagonal) or over-confident (above the diagonal). The weighted sum of the red blocks is the ECE in* (E.4.21).

其他指标，如对分箱的最大值，也是可能的。如果发现模型未校准，则需要做出修改。例如，通过温度缩放 [GPSW17] 对预测进行重新缩放或使用不同的损失函数，如焦点损失 [MKS+20] 进行优化。

我们最后提到模型直接校准的替代方案，称为一致性预测，最近变得很受欢迎 [AB21]。假设我们固定一个阈值 γ，我们取模型预测的类别集合，其对应的概率高于 γ：

$$\mathscr{C}(\mathbf{x}) = \{i \mid [f(\mathbf{x})]_i > \gamma\} \tag{E.4.22}$$

即，模型的答案现在是一个 *set* $\mathscr{C}(\mathrm{x})$ 的潜在类别。一个例子如图 F.4.6 所示。符合预测的想法是选择最小的 γ 使得

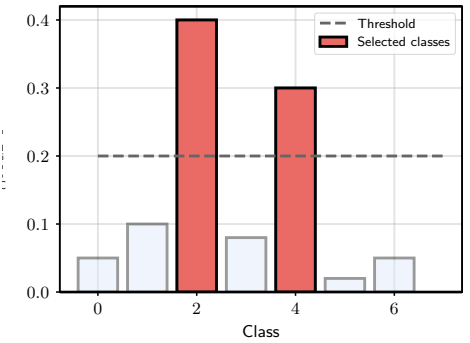


图 F.4.6：*Calibration by turning the model's output into a set: we return all classes whose predicted probability exceeds a given threshold. By properly selecting the threshold we can bound the probability of the true class being found in the output set.*

找到集合中正确类别 y 的概率高于用户定义的错误 α:[6]

$$p(y \in \mathscr{C}(\mathbf{x})) \geq 1 - \alpha \tag{E.4.23}$$

直观上，γ 和 α 之间存在反比关系。共形预测提供自动算法以保证（E.4.23），但代价是输出不再有单一类别。

[6]Note that it is always possible to satisfy this property by selecting $\gamma = 0$, i.e., including all classes in the output set.

从理论到实践

从第2章起，你应该对NumPy、JAX和PyTorch的torch.tensor有很好的掌握。这章所需的就是这些，不需要其他任何内容。从下一章开始，我们将进入它们的高级API。

我建议一个简短的练习，让您从头开始训练您的第一个可微模型：

1. 加载一个玩具数据集：例如，scikit-learn数据集模块中包含的其中一个.[7]

2. 建立一个线性模型（根据数据集是回归还是分类）。考虑如何使代码尽可能模块化：正如我们将看到的，您至少需要两个函数，一个用于初始化模型的参数，另一个用于计算模型的预测。

3. 通过梯度下降训练模型。目前你可以手动计算梯度：尝试想象如何使这部分也模块化，即，如果你想要动态地添加或删除模型中的偏差，你将如何改变梯度的计算？

4. 在一个独立的测试集上绘制损失函数和准确率。如果你了解一些标准机器学习，你可以将结果与其他监督学习模型进行比较，例如决策树或*k*-NN，始终使用scikit-learn。

[7] https://scikit-learn.org/stable/datasets/toy_dataset.html

5 | 全连接模型

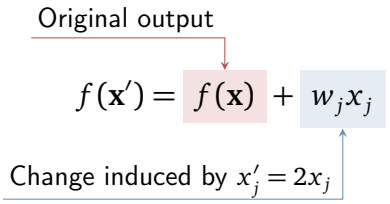
> 关于本章
>
> 在这一章中，我们展示了如何通过组合一系列所谓的 *fully- connected layers* 来构建可微模型。由于历史原因，这些模型也被称为多层感知器（MLPs）。MLPs将线性块（类似于第4章）与非线性函数交织在一起，有时称为 *activation functions*。

5.1 线性模型的局限性

线性模型在本质上有限制，因为根据定义，它们无法对特征之间的非线性关系进行建模。例如，考虑两个输入向量 x 和 x′，除了由 j 索引的单个特征外，它们是相同的：

$$x_i' = \begin{cases} x_i & \text{if } i \neq j \\ 2x_i & \text{otherwise} \end{cases}$$

例如，这可以表示两家银行的客户，在所有方面都相同，除了他们的收入，其中x′的收入是x的两倍。如果*f*是一个线性模型（没有偏差），则有：



因此，输入变化唯一的后果是输出的小线性变化，由 w_j 决定。假设我们在对用户进行评分，我们可能希望模拟诸如“*an income of 1500 is low, except if the age* < 30”.[1] 显然，由于上述分析，这不能通过线性模型来完成。

这是这种类型的典型示例是XOR数据集，一个二元数据集，其中每个特征只能取{0，1}中的值。因此，整个数据集只有4种可能性：

$$f([0,0]) = 0\,,\ f([0,1]) = 1\,,\ f([1,0]) = 1\,,\ f([1,1]) = 0$$

在两个输入中的 *only one* 为正时，输出始终为正。尽管其简单，这也不可线性分离，无法通过线性模型以100%的准确率解决 - 请参阅图F.5.1以进行可视化。

[1] You probably shouldn't do credit scoring with machine learning anyways.

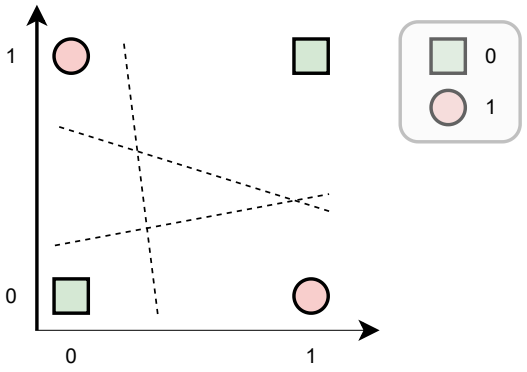


图 F.5.1：*Illustration of the XOR dataset: green squares are values of one class, red circles are values of another class. No linear model can separate them perfectly (putting all squares on one side and all circles on the other side of the decision boundary). We say that the dataset is not* ***linearly separable****.*

5.2 组成和隐藏层

编程中的一个强大思想是分解，即递归地将问题分解为其组成部分，直到每个部分都可以用简单、可管理的操作来表示。在我们的情况下，可以通过想象我们的模型 f 实际上是两个可训练操作的组合来实现类似的效果：

$$f(\mathbf{x}) = (f_2 \circ f_1)(\mathbf{x})$$

在 $f_2 \circ f_1$ 是两个函数的组成：$(f_2 \circ f_1)(\mathrm{x}) = f_2(f_1(\mathrm{x}))$，并且我们假设每个函数实例化其自己的可训练参数集。我们可以继续细分计算：

$$f(\mathbf{x}) = (f_l \circ f_{l-1} \circ \cdots \circ f_2 \circ f_1)(\mathbf{x})$$

我们现在总共有 l 个函数正在被组合。注意，只要每个 f_i 不改变其输入数据的“类型”，我们就可以将这些变换链式连接起来，我们想要的任意多个，每个都会添加它自己的可训练参数集。

例如，在我们的情况下，输入 x 是一个向量，因此任何向量到向量的运算（例如，矩阵乘法 $f_i(\mathrm{x}) = \mathrm{W}\mathrm{x}$）可以无限次地组合在一起。然而，必须注意一些事项。假设我们将两个不同的线性投影链在一起：

$$\mathbf{h} = f_1(\mathbf{x}) = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1 \tag{E.5.1}$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.2}$$

它很容易证明这两个投影“合并”成一个：

$$y = \underbrace{(\mathbf{w}_2^\top \mathbf{W}_1)}_{\triangleq \mathbf{A}}\mathbf{x} + \underbrace{(\mathbf{w}_2^\top \mathbf{b}_1 + b_2)}_{\triangleq \mathbf{c}} = \mathbf{A}\mathbf{x} + \mathbf{c}$$

全连接（FC）模型的概念，也称为多层感知器（MLPs），出于历史原因，是在投影之间插入一个简单的逐元素非线性 $\phi : \mathbb{R} \to \mathbb{R}$ 以避免崩溃：

元素级非线性

$$\mathbf{h} = f_1(\mathbf{x}) = \phi\,(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{E.5.3}$$

$$y = f_2(\mathbf{h}) = \mathbf{w}_2^\top \mathbf{h} + b_2 \tag{E.5.4}$$

第二个块可以是线性的，如（E.5.4）中所示，或者根据任务将其包装到另一个非线性函数中（例如，用于分类的softmax函数）。函数ϕ可以是任何非线性函数，例如多项式、平方根或sigmoid函数σ。正如我们将在下一章中看到的，选择它对模型的梯度有强烈的影响，从而对优化也有影响，挑战在于选择一个ϕ，它“非线性足够”以防止崩溃，同时在其导数中尽可能接近恒等函数。一个好的默认选择是所谓的ReLU（修正线性单元）。

定义 D.5.1（修正线性单元）

*The **rectified linear unit** (ReLU) is defined elementwise as:*

$$\mathrm{ReLU}(s) = \max(0, s) \tag{E.5.5}$$

我们将在下一章中有很多关于ReLU要说的。随着ϕ的添加，我们现在可以链式连接任意多的变换：

$$y = \mathbf{w}_l^\top \phi\left(\mathbf{W}_{l-1}\left(\phi\left(\mathbf{W}_{l-2}\phi\left(\cdots\right) + \mathbf{b}_{l-2}\right)\right) + \mathbf{b}_{l-1}\right) \tag{E.5.6}$$

在本章的其余部分，我们专注于分析这类模型的训练和逼近特性。然而，首先简要讨论一下命名约定。

关于神经网络术语

我们已经提到，神经网络有着悠久的历史和大量的术语，我们在此简要总结。每个f_i被称为模型的层，其中f_l是输出层，f_i、$i = 1$、...、$l - 1$是隐藏层，并且通过一点符号超载，x是

输入层。使用这个术语，我们可以以下批处理形式重新表述全连接层的定义。

定义 D.5.2（全连接层）

For a batch of n vectors, each of size c, represented as a matrix $\mathrm{X} \sim (n, c)$*, a* ***fully-connected*** *(FC) layer is defined as:*

$$\mathrm{FC}(\mathbf{X}) = \phi\left(\mathbf{X}\mathbf{W} + \mathbf{b}\right) \tag{E.5.7}$$

The parameters of the layer are the matrix $\mathrm{W} \sim (c, c')$ *and the bias vector* $\mathrm{b} \sim (c')$*, for a total of* $(c + 1)c'$ *parameters (assuming* ϕ *does not have parameters). Its hyper-parameters are the width* c' *and the non-linearity* ϕ*.*

输出 $f_i(\mathrm{x})$ 被称为层的激活，其中我们有时可以区分预激活和后激活（非线性之前和之后）。非线性 ϕ 本身可以称为激活函数。f_i 的每个输出被称为一个神经元。尽管大部分这个术语已经过时，但它仍然很普遍，当需要时我们会使用它。

每一层的尺寸（输出形状）是一个用户可以选择的超参数，因为它只影响下一层的输入形状，这被称为层的宽度。对于大量层，超参数的数量线性增长，其选择变成一个组合任务。我们将在第9章中回到这一点，当时我们将讨论具有数十（或数百）层的模型设计。

层概念也广泛应用于通用框架中。例如（E.5.7）中的层可以定义为

```
class FullyConnectedLayer(nn.Module):
  def __init__(self, c: int, cprime: int):
    super().__init__()
    # Initialize the parameters
    self.W = nn.Parameter(
                torch.randn(c, cprime))
    self.b = nn.Parameter(
                torch.randn(1, cprime))

  def forward(self, x):
    return relu(x @ self.W + self.b)
```

盒子 C.5.1: *The FC layer in* (E.5.7) *implemented as an object in PyTorch. We require a special syntax to differentiate trainable parameters, such as* W*, from other non-trainable tensors: in PyTorch, this is obtained by wrapping the tensors in a* 参数 *object. PyTorch also has its collection of layers in* torch.nn*, including the FC layer (implemented as* torch.nn.Linear*).*

对象具有两个功能：一个初始化函数，该函数根据所选的超参数随机初始化模型的全部参数，以及一个调用函数，该函数提供层的输出。参见盒C.5.1中的示例。然后，可以通过链接此类层的实例来定义一个模型。例如，在PyTorch中，这可以通过Sequential对象实现：

```
模型 = nn.Sequential( 全连接层(3, 5), 全连
接层(5, 4) )
```

请注意，从它们的输入输出签名来看，Box C.5.1中定义的层与上面定义的模型之间没有很大差异，我们可以等效地将模型用作更大模型中的一层。这种组合性是该特性的定义特征。

可微模型。

5.2.1 MLPs的近似性质

训练MLPs的过程与我们所讨论的线性模型类似。例如，对于回归任务，我们可以最小化均方误差：

$$\min_{\{\mathbf{W}_k,\mathbf{b}_k\}_{k=1}^{l}} \frac{1}{n}\sum_i (y_i - f(\mathbf{x}_i))^2$$

在模型的所有参数上同时进行最小化。我们将在下一章中看到一个计算这种情况下的梯度的通用过程。

目前，我们注意到与具有线性模型相比的主要区别在于添加一个隐藏层使得整体优化问题非凸，具有多个局部最优解，这取决于模型的初始化。这在历史上是一个重要方面，因为监督学习的替代方法（例如，支持向量机 [HSS08]）提供了非线性模型同时保持凸性。然而，过去十年的结果表明，高度非凸模型可以在许多任务中实现显著的良好性能。[2]

从理论角度来看，我们可以问增加隐藏层有什么意义，即如果线性模型只能解决线性可分的问题，那么通过增加隐藏层，可以近似哪些函数类别？结果证明，

[2]The reason differentiable models generalize so well is an interesting, open research question, to which we return in Chapter 9. Existing explanations range from an implicit bias of (stochastic) gradient descent [PPVF21] to intrinsic properties of the architectures themselves [AJB+17, TNHA24].

单层隐藏层就足以具有通用逼近能力。在这个意义上，G. Cybenko于1989年证明了这一开创性结果 {v*}。

> 定理5.1（MLP的通用逼近）
> *Given a continuous function* $g\ \mathbb{R}^d \rightarrow \mathbb{R}$*, we can always find a model* $f(\mathrm{x})$ *of the form* (E.5.3)-(E.5.4) *(an MLP with a single hidden layer) and sigmoid activation functions, such that for any* $\varepsilon > 0$*:*
>
> $$|f(\mathbf{x}) - g(\mathbf{x})| \leq \varepsilon\ ,\ \forall \mathbf{x}$$
>
> *where the result holds over a compact domain. Stated differently, one-hidden-layer MLPs are "dense" in the space of continuous functions.*

这个定理的美丽不应分散人们对这样一个纯粹理论结构的关注，即它利用了模型隐藏层宽度可以无界增长的事实。因此，对于任何前一个不等式不成立的x，我们都可以添加一个新单元来减少逼近误差（参见附录B）。事实上，可以设计出函数类，其中所需的隐藏神经元数量随着输入特征数量[Ben09].[3]呈指数增长。

许多其他作者，如 [Hor91]，逐渐完善了这一结果，使其包括具有任何可能激活函数的模型，包括 ReLUs。此外，对于具有有限 *width* 但可能无限 *depth* 的模型，也可以证明其具有通用逼近能力。

[3]One of these problems, the *parity* problem, is closely connected to the XOR task: https://blog.wtf.sg/posts/2023-02-03-the-new-xor-problem/.

[LPW{v*}17{v*}. 一条独立的研究线探讨了{v*}模型的近似能力，其中参数数量超过训练数据。在这种情况下，在许多有趣的场景中可以证明训练到全局最优{v*}DZPS19，AZLL19{v*}非正式地，对于足够多的参数，模型可以达到每个训练样本的损失最小值，从而实现优化问题的全局最小值{v*}。参见附录B，以一维可视化Cybenko定理。

近似和学习能力是可微模型的巨大研究领域，有无数书籍致力于此，我们在这里只提到了一些重要结果。在本书的其余部分，我们将主要关注模型本身的有效设计，其行为可能比这些定理暗示的更复杂且难以控制（和设计）。

5.3 随机优化

为了优化模型，我们可以在相应的经验风险最小化问题上执行梯度下降。然而，当数据集的大小非常大时，这可能很难实现。我们将在下一章中看到，计算损失梯度需要与示例数量成线性关系的时间，当数量达到10或更多时，这变得不可行或很慢，特别是对于大型模型（不考虑内存问题）。

幸运的是，问题的形式适合于一个很好的近似，其中我们使用数据子集来计算下降方向。为此，假设对于

迭代 {v*} 的梯度下降中，我们从数据集 r 中采样一个子集 $\mathcal{B}_t \subset \mathcal{S}_n$（带有 $r \ll n$）点，我们称之为小批量。我们可以通过只考虑小批量来计算一个近似损失：

$$\widetilde{L}_t = \underbrace{\frac{1}{r}\sum_{(x_i,y_i)\in\mathcal{B}_t} l(y_i, f(x_i))}_{\text{Mini-batch}} \approx \underbrace{\frac{1}{n}\sum_{(x_i,y_i)\in\mathcal{S}_n} l(y_i, f(x_i))}_{\text{Full dataset}} \tag{E.5.8}$$

如果我们假设迷你批次的元素是从数据集中独立同分布采样的，$\widetilde{L}_t$ 是全损失的蒙特卡洛近似，其梯度也是如此。然而，其计算复杂度仅随 r 增长，这可以通过用户控制。粗略地说，迷你批次较低维度的 r 导致迭代更快，梯度方差更高，而较高的 r 则导致迭代更慢、更精确。对于大型模型，内存通常是最大的瓶颈，而迷你批次大小 r 可以选择以填充每次迭代的可用硬件。

梯度下降应用于数据的小批量是随机梯度下降（SGD）的一个例子。由于上述讨论的性质，可以证明SGD在期望上收敛到最小值，并且在训练可微模型时是首选的优化策略。

最后剩下的问题是如何选择小批量。对于大型数据集，随机采样元素可能很昂贵，尤其是如果我们需要将它们在GPU内存中来回移动。一个便于优化中间解决方案如下：

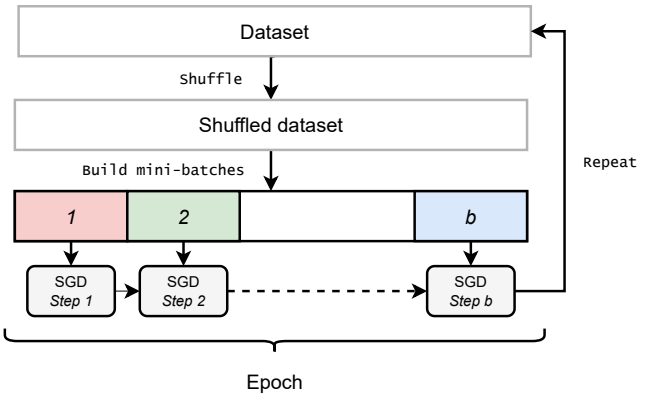


图 F.5.2：*Building the mini-batch sequence: after shuffling, stochastic optimization starts at mini-batch* 1*, which is composed of the first* r *elements of the dataset. It proceeds in this way to mini-batch* b *(where* $b = \frac{n}{r}$*, assuming the dataset size is perfectly divisible by* r*). After one such epoch, training proceed with mini-batch* $b+1$*, which is composed of the first* r *elements of the shuffled dataset. The second epoch ends at mini-batch* $2b$*, and so on.*

1. 首先对数据集进行打乱。2. 然后，将原始数据集划分为包含 r *consecutive* 个元素的微型批次，并依次处理每个批次。假设数据集大小为 $n = r\,b$，这将产生 b 个微型批次，从而有 b 步的随机梯度下降（SGD）。如果我们正在GPU上执行代码，这一步包括将微型批次发送到GPU内存。

3. 在完成以这种方式构建的所有小批量后，返回到1点并迭代。

一个完整的这个过程称为训练周期，它是一个非常常见的超参数来指定（例如，对于包含1000个元素的数据库和20个元素的迷你批次，“*training for 5 epochs*”表示训练250次）。昂贵的洗牌

```
# A dataset composed by two tensors
dataset = torch.utils.data.TensorDataset(
    torch.randn(1000, 3),
    torch.randn(1000, 1))

# The data loader provides
# shuffling and mini-batching
from torch.utils.data import DataLoader
dataloader = DataLoader(dataset,
                shuffle=True,
                batch_size=32)

for xb, yb in dataloader:
  # Iterating over mini-batches (one epoch)
  # xb has shape (32, 3)
  # yb has shape (32, 1)
```

盒子 C.5.2: *Building the mini-batch sequence with PyTorch's data loader: all frameworks provide similar tools.*

操作在每个epoch中只执行一次，而在两个epoch之间，可以通过框架快速预取和优化mini-batches。这如图F.5.2所示。大多数框架提供了一种将数据集组织成可以单独索引的元素的方法，以及一个单独的接口来构建mini-batch序列。例如，在PyTorch中，这是通过Dataset和DataLoader接口分别完成的 - 请参阅框C.5.2。

这种设置也自然地导致跨GPU或跨机器的简单并行形式。如果我们假设每台机器足够大，可以存储整个模型的参数副本，我们可以在机器上并行处理不同的迷你批次，然后对它们的局部贡献进行求和，以进行最终更新，然后将更新广播回每台机器。这被称为数据

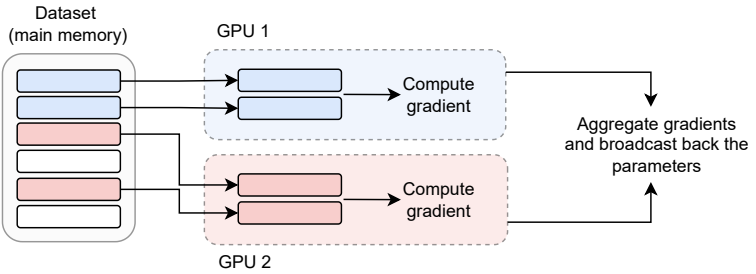


图 F.5.3：*A simple form of distributed stochastic optimization: we process one mini-batch per available machine or GPU (by replicating the weights on each of them) and sum or average the corresponding gradients before broadcasting back the result (which is valid due to the linearity of the gradient operation). This requires a synchronization mechanism across the machines or the GPUs.*

并行设置在PyTorch中，[4]，并在图F.5.3中进行了视觉展示。更复杂的并行形式，例如张量并行，也是可能的，但本书中不涉及它们。

5.4 激活函数

我们通过提供一个关于激活函数选择的简要概述来结束这一章。正如我们在上一节所述，几乎任何逐元素的非线性都是理论上有效的。然而，并非所有选择都有良好的性能。例如，考虑一个简单的多项式函数，对于某些用户定义的正整数 p：

$$\phi(s) = s^p$$

[4] https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

对于大的 p，这将在两侧迅速增长，跨层复合，导致难以训练且数值不稳定的模型。

历史上，神经网络被引入作为生物神经元的近似模型（因此得名*artificial NNs*）。在这个意义上，点积$\mathrm{w}^\top$中的权重是突触的简单模型，偏置b是阈值，当输入的累积和超过阈值时，神经元就会被“激活”：

$$s = \mathbf{w}^\top \mathbf{x} - b \ , \ \phi(s) = \mathbb{I}_{s \geq 0}$$

在 $\mathbb{I}_b$ 是一个指示函数，当 b 为真时取值为 1，否则为 0。由于此激活函数不可微，可以使用 sigmoid $\sigma(s)$ 作为软近似。实际上，我们可以定义一个具有可调斜率 a 的广义 sigmoid 函数 $\sigma_a(s) = \sigma(as)$，我们有：

$$\lim_{a \to \infty} \sigma_a(s) = \mathbb{I}_{s \geq 0}$$

另一个常见变体是双曲正切，它是 $[-1、+1]$ 中sigmoid的缩放版本：

$$\tanh(s) = 2\sigma(s) - 1$$

现代神经网络，自2012年AlexNet普及以来 [KSH12]，反而使用了(E.5.5)中的ReLU函数。ReLU相对于sigmoid函数的相对优势将在下一章讨论。我们在此指出，ReLU有几个反直觉的性质。

例如，它们在0处有一个不可微的点，并且由于所有负输入都被设置为0，因此它们具有很大的输出稀疏性。这个第二个属性可能导致所谓的“死神经元”，其中某些单元对所有输入都有恒定的0输出。这可以通过ReLU的一个简单变体来解决，称为Leaky ReLU：

$$\text{LeakyReLU}(s) = \begin{cases} s & \text{if } s \geq 0 \\ \alpha s & \text{otherwise} \end{cases} \tag{E.5.9}$$

对于非常小的 α，例如 $\alpha = 0.01$。我们还可以为每个单元训练不同的 α（因为函数相对于 α 是可微的）。在这种情况下，我们称 AF 为参数化 ReLU（PReLU）[HZRS15]。可训练的激活函数通常是一种简单的方法，通过少量参数添加少量灵活性——在 PReLU 的情况下，每个神经元一个。

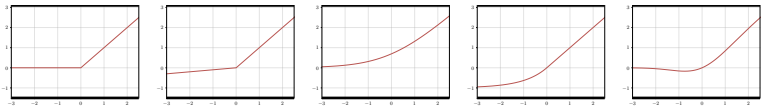
完全可微的ReLU变体也可用，例如softplus：

$$\text{softplus}(s) = \log(1 + \exp(s)) \tag{E.5.10}$$

软加函数不通过原点，并且总是大于0。另一种变体，指数线性单元（ELU），在原点处保持通过，同时将下界切换为−1：

$$\text{ELU}(s) = \begin{cases} s & \text{if } s \geq 0 \\ \exp(s) - 1 & \text{otherwise} \end{cases} \tag{E.5.11}$$

然而，可以通过注意以下方式定义另一类变体



ReLU *LeakyReLU* *Softplus* *ELU* *GELU*

图 F.5.4：*Visual comparison of ReLU and four variants: LeakyReLU* (E.5.9)*, Softplus* (E.5.10)*, ELU* (E.5.11)*, and GELU. LeakyReLU is shown with* $\alpha = 0.1$ *for better visualization, but in practice* α *can be closer to* 0 *(e.g.,* 0.01*)*..

相似度 ReLU 与指示函数。我们可以将 ReLU 重写为：

$$\text{ReLU}(s) = s \cdot \mathbb{I}_{s \geq 0}$$

因此，ReLU与负象限上的指示函数相同，而在正象限上用s替换1。我们可以通过用加权因子$\beta(s)$替换指示函数来推广这一点：

$$\text{GeneralizedReLU}(s) = s \cdot \beta(s)$$

选择 $\beta(s)$ 作为累积高斯分布函数，我们得到高斯 ELU（GELU）[HG16]，而对于 $\beta(s) = \sigma(s)$，我们得到 Sigmoid 线性单元（SiLU）[HG16]，也称为 Swish [RZL17]。我们在图 F.5.4 中绘制了一些这些 AF。除了某些细节（例如负象限中的单调性）外，它们都相对相似，通常很难通过简单地替换激活函数来显著提高性能。

每个函数可以有多个可训练的变体

通过向函数添加可训练参数获得。例如，Swish的常见可训练变体具有四个参数 $\{a、b、c、d\}$，其获得方式如下：

$$\text{Trainable-Swish}(s) = \sigma(as + b)(cs + d) \tag{E.5.12}$$

我们也可以设计非参数激活函数，即不具有固定数量可训练参数的激活函数。例如，考虑一个通用的（不可训练）标量函数集 ϕ_i，由整数 i 索引。我们可以将这些基 n 的线性组合构建为一个完全灵活的激活函数：

$$\phi(s) = \sum_{i=1}^{n} \alpha_i \phi_i(s) \tag{E.5.13}$$

在 n 是一个超参数的情况下，而系数 α_i 通过梯度下降进行训练。它们可以对于所有函数相同，或者对于每一层和/或神经元不同。根据 ϕ_i 的选择，我们获得不同类别的函数：如果每个 ϕ_i 是一个 ReLU，我们获得自适应分段线性（APL）函数 [AHSB14]，而对于更通用的核，我们获得核激活函数（KAF）[MZBG18, SVVTU19]。通过考虑具有多个输入和多个输出的函数，可以获得更通用的模型 [LCX+23]。参见 [ADIP21] 以获取概述。

一般来说，没有“最佳AF”的答案，因为这取决于任务、数据集和架构。ReLU是一个常见的选择，因为它表现良好，代码高度优化，并且有轻微的成本开销。重要的是要考虑基本的计算权衡，即在给定的预算下，更复杂的AF可能会导致宽度或深度更小，这可能会阻碍整个性能。

架构。因此，具有许多可训练参数的AFs较少见。

设计变体

并非每一层都适合 *linear projections* 和 *element-wise* 非线性的框架，我们在此描述三种常见变体。首先，门控线性单元（GLU）[DFAG17] 结合了 (E.5.12) 的结构与乘法（Hadamard）交互：

$$f(\mathbf{x}) = \sigma(\mathbf{W}_1\mathbf{x}) \odot (\mathbf{W}_2\mathbf{x}) \tag{E.5.14}$$

在 W_1 和 W_2 上进行了训练。另一种可能性，SwiGLU，将 (E.5.14) 中的 sigmoid 替换为 Swish 函数 [Sha20]。门控 MLP 实际上是现代 LLM 中的一种流行选择，参见第 11 章。

其次，在maxout网络中 [GWFM$^+$13] 每个单元产生 k (超参数)不同投影的最大值。最后，用形式为（E.5.13）的可训练非线性矩阵 $W_{ij} \to \phi_{ij}(x_j)$ 替换线性投影 W，最近也被提出，称为Kolmogorov-Arnold网络（KANs，[LWV$^+$24]）：

$$h_i = \sum_j \phi_{ij}(x_j)$$

从理论到实践

本章介绍了任何用于训练可微分模型的通用框架的两个关键要求：

1. 处理需要洗牌的大数据集的方法，

   分割成小批量，并在GPU之间来回移动。在PyTorch中，这大部分是通过Dataset和DataLoader接口实现的，如Box C.5.2中所示。

2. 一种从基本块组合构建模型的方法，称为 *layers*。在 PyTorch 中，层通过 torch.nn 模块实现，可以通过 Sequential 接口或通过继承 Module 类来组合，如 Box C.5.1 中所示。

我建议你现在尝试复现PyTorch文档中可用的许多快速指南之一.[5]除了下一章中介绍的梯度计算机制外，其他内容应该都相当清晰。这也是研究Hugging Face Datasets的好时机，它结合了一个庞大的数据集存储库，并使用Apache Arrow提供了一种框架无关的接口来处理和缓存这些数据集.[6]

JAX不提供高级工具。对于数据加载，您可以使用任何现有工具，包括PyTorch的数据加载器和Hugging Face Datasets。对于构建模型，

[5] https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

[6] https://huggingface.co/docs/datasets/en/quickstart

最简单的方法是依赖外部库。因为JAX功能齐全，所以不可能实现像Box C.5.1这样的面向对象抽象。我个人的建议是Equinox [KG21]，它通过结合JAX的基本数据结构（pytree）和可调用节点，提供类似类的体验。

6 | 自动微分

关于本章

上一章强调了计算任何可能操作序列的梯度需要一个高效、自动的程序。在本章中，我们描述了这样一个方法，在神经网络文献中称为反向传播，在计算机科学中称为逆模式自动微分。其分析有多个见解，从模型的选择到优化它的内存需求。

6.1 问题设置

我们考虑高效计算通用 *computational graphs* 的梯度问题，例如通过优化全连接模型上的标量损失函数所诱导的梯度，这是一个称为自动微分（AD）[BPRS18] 的任务。你可以将计算图视为通过运行以下操作获得的原子操作集合（我们称之为原语）：

程序本身。我们将为了简洁考虑顺序图，但一切都可以轻松扩展到无环图甚至更通用的计算图 {v*}GW08，BR24{v*}。

问题可能看似微不足道，因为雅可比链式法则（第2.2节，（E.2.23））告诉我们，函数复合的梯度仅仅是相应雅可比矩阵的矩阵乘积。然而，有效地实现这一过程是本章的关键挑战，而由此产生的算法（反向模式自动微分或反向传播）是神经网络和可微分编程的基石 [GW08, BR24]。理解它也是理解大多数实现和训练此类程序（如TensorFlow或PyTorch或JAX）框架的设计（以及差异）的关键。该算法的简要历史可以在 [Gri12] 中找到。

为了设置问题，我们假设我们有一组原语可供使用：

$$\mathbf{y} = f_i(\mathbf{x}, \mathbf{w}_i)$$

每个原语代表对输入向量 x $\sim (c_i)$ 的操作，由向量 w $_i \sim (p_i)$ (参数化，例如线性投影的权重)，并输出另一个向量 y $\sim (c_i')$。

在我们的原语定义中有很多灵活性，它可以表示基本的线性代数运算（例如，矩阵乘法），第5章中的层（例如，具有激活函数的全连接层），甚至更大的块或模型。这种递归可组合性是编程的关键属性，并扩展到我们的情况。

我们仅假设对于每个原函数，我们知道如何计算相对于两个输入参数的偏导数，我们称这些偏导数为操作的输入雅可比和权重雅可比：

$$\begin{aligned} \textbf{Input Jacobian:} \quad & \partial_{\mathbf{x}}[f(\mathbf{x},\mathbf{w})] \sim (c', c) \\ \textbf{Weight Jacobian:} \quad & \partial_{\mathbf{w}}[f(\mathbf{x},\mathbf{w})] \sim (c', p) \end{aligned}$$

这些是合理的假设，因为我们把我们的分析限制在可微模型上。具有一个或多个非可微点的连续原语，如ReLU，可以通过使用子梯度（第6.4.4节）纳入这个框架。通过找到它们梯度的松弛或等效估计器[NCN+23]，也可以包括非可微操作，如采样或阈值化。我们将在下一卷中涵盖后者情况。

关于我们的符号和一阶雅可比矩阵

我们仅考虑向量值量以增强可读性，因为所有结果梯度都是矩阵。在实践中，现有的原语可能有更高阶的输入、权重或输出。例如，考虑一个基本的在批处理输入上的全连接层：

$$f(\mathbf{X},\mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

在这种情况下，输入X的形状为$(n,\ c)$，权重具有形状$(c,\ c')$和(c') (，其中c'是一个超参数)，输出具有形状$(n,\ c')$。因此，输入雅可比矩阵具有形状$(n,\ c',\ n,\ c)$，权重雅可比矩阵具有形状$(n,\ c',\ c,\ c')$，两者都具有4阶。

在我们的记号中，我们可以考虑等效的展平

向量 x = vect(X) 和 w = [vect(W)；b]，以及我们得到的“展平”后的雅可比矩阵的形状分别为 (nc', nc) 和 (nc', cc')。这在以下内容中至关重要，因为每次我们提到“*the input size c*”时，我们都是在指“*the product of all input shapes*”，包括可能的批处理维度。这也表明，虽然我们可能知道如何计算雅可比矩阵，但由于它们的维度很大，我们可能不希望将它们完全*materialize* 存储在内存中。

作为最后的说明，我们的符号与这些原语在功能库（如JAX）中的实现方式一致。在面向对象的框架（例如TensorFlow、PyTorch）中，我们发现层被实现为对象（参见前一章的Box C.5.1），参数是对象的属性，函数调用被对象的方法所取代。这种风格简化了某些实践，例如所有参数的延迟初始化直到输入形状已知（懒加载），但它增加了一层需要考虑的抽象，以将我们的符号转换为可工作的代码。正如我们将看到的，这些差异在两个框架中实现AD的方式中得到了反映。

6.1.1 问题陈述

有了所有这些细节，我们准备陈述AD任务。考虑一个由l个原始调用组成的序列，随后是一个最终求和：

$$\begin{aligned}\mathbf{h}_1 &= f_1(\mathbf{x}, \mathbf{w}_1)\\ \mathbf{h}_2 &= f_2(\mathbf{h}_1, \mathbf{w}_2)\\ &\vdots\\ \mathbf{h}_l &= f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)\\ y &= \sum \mathbf{h}_l\end{aligned}$$

这被称为程序的评估轨迹。大致上，前 $l-1$ 次操作可以表示可微模型的几层，操作 l 可以是每个输入的损失（例如，交叉熵），最后一步操作汇总了小批次的损失。因此，我们程序的输出始终是一个标量，因为我们需要它进行数值优化。我们将之前的程序简写为 $F(\mathrm{x})$。

定义 D.6.1（自动微分）

Given a program $F(\mathrm{x})$ composed of a sequence of differentiable primitives, ***automatic differentiation*** *(AD) refers to the task of simultaneously and efficiently computing all weight Jacobians of the program given knowledge of the computational graph and all individuals input and weight Jacobians:*

$$AD(F(\mathbf{x})) = \left\{\partial_{\mathbf{w}_i} y\right\}_{i=1}^{l}$$

我们将会看到，存在两种主要的AD算法，称为前向模式和后向模式，对应于个体操作组合中的不同顺序。我们还将看到，后向模式（在神经网络中称为反向传播）

网络文献）在我们的环境中效率显著更高。虽然我们关注一个简化的场景，但将我们的推导扩展到无环图原语（如已提及）以及参数跨层共享的情况相对容易（权重共享）。我们将在第13章中看到一个权重共享的例子。

6.1.2 数值和符号微分

在继续介绍前向模式AD之前，我们讨论AD与其他函数微分算法类别的区别。首先，我们可以直接应用梯度的定义（第2.2节）来获得梯度的合适数值近似。这个过程被称为数值微分。然而，在原始实现中，每个要微分的标量值都需要2次函数调用，这使得这种方法除了在实现上的数值检查之外不可行。

其次，考虑这个简单的函数:

$$f(x) = a\sin(x) + bx\sin(x)$$

我们可以要求一个符号引擎预先计算导数的完整符号方程。这被称为符号微分，在Python中显示在框C.6.1中。

在一个现实实现中，中间值 $h = \sin(x)$ 只会被计算一次并存储在一个中间变量中，这个变量也可以用于梯度追踪中的对应计算（对于 $\cos(x)$ 项也有类似的推理）。

```
导入 sympy 作为 sp x, a, b = sp.symbols('x a b')
y = a*sp.sin(x) + b*x*sp.sin(x) sp.diff(y, x) # [O
ut]: acos(x)+bxcos(x)+bsin(x)
```

盒子 C.6.1：*Symbolic differentiation in Python using SymPy.*

导数)。这比看起来要复杂：找到一个避免任何不必要的计算的雅可比最优实现是一个NP完全问题（最优雅可比累积）。然而，我们将看到我们可以利用我们程序的结构来设计一个足够高效的AD实现，这比上述符号方法要好得多（实际上，它等同于允许存在子序列[Lau19]的符号方法）。

6.2 前向模式微分

我们首先回顾雅可比矩阵的链式法则。考虑两个基本函数的组合：

$$\mathbf{h} = f_1(\mathbf{x}) \, , \, \mathbf{y} = f_2(\mathbf{h})$$

关于它们的梯度，我们有：

$$\partial_{\mathbf{x}} \mathbf{y} = \partial_{\mathbf{h}} \mathbf{y} \cdot \partial_{\mathbf{x}} \mathbf{h}$$

如果 x、h 和 y 分别具有维度 a、b 和 c，则前面的雅可比矩阵需要将一个 $c \times b$ 矩阵（绿色）与一个 $b \times a$ 矩阵（红色）相乘。我们可以将规则解释如下：如果我们已经计算了 f_1 及其

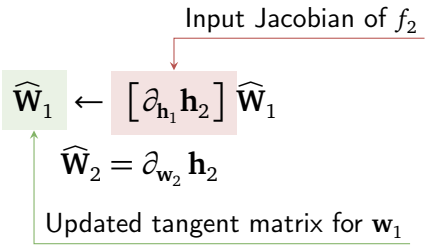
雅可比矩阵（红色项），一旦应用 f_2，我们可以通过乘以相应的雅可比矩阵（绿色项）来“更新”梯度。

我们可以立即应用这一洞察力，得到一个称为前向模式自动微分（F-AD）的工作算法。其思想是，每次我们应用一个原始函数时，我们初始化其对应的权重雅可比矩阵（在此语境中称为切线），同时更新所有之前的切线矩阵。让我们通过一个简单的示例来说明主要算法。

考虑程序中的第一条指令，$\mathrm{h}_1 = f_1(\mathrm{x}, \mathrm{w}_1)$。因为到目前为止还没有存储任何内容，我们将$\mathrm{w}_1$的正切矩阵初始化为其权重雅可比：

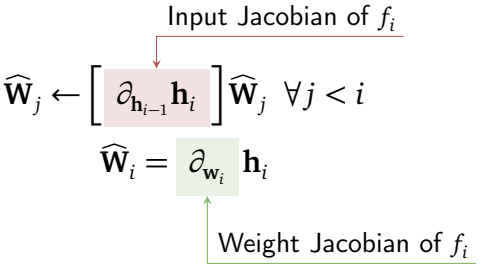
$$\widehat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$$

我们现在进行第二个指令，$\mathrm{h}_2 = f_2(\mathrm{h}_1, \mathrm{w}_2)$。我们在更新前一个切线矩阵的同时初始化第二个矩阵：



更新需要原始输入雅可比矩阵，而第二项需要原始权重雅可比矩阵。抽象化，考虑由 $\mathrm{h}_i = f_i(\mathrm{h}_{i-1}, \mathrm{w}_i)$ 给出的通用 i - 次原始。我们初始化 w_i 的切线矩阵，同时更新 *all*

之前的矩阵：



第一行有 $i-1$ 个更新（每个更新对应我们已存储在内存中的每个切线矩阵），红色项——第 i 次操作的输入雅可比——被所有先前切线共享。程序中的最后一个操作是求和，相应的梯度给出了算法的输出：[1]

$$\nabla_{\mathbf{w}_i} y = \mathbf{1}^\top \widehat{\mathbf{W}}_i \;\; \forall i \tag{E.6.1}$$

完成！让我们更详细地分析这个算法。首先，我们列出的所有操作都可以很容易地与原始程序*interleaved*，这意味着空间复杂度将与我们要微分程序的空间复杂度大致成比例。

在负面上，算法的核心操作（$\widehat{\mathrm{W}}_i$的更新）需要两个矩阵的乘法，其通用形状为(c_i', c_i)和(c_i, p_j)，其中c_i，c_i'是输入/输出形状，p_j是w_j的形状。这是一个极其昂贵的操作：例如，假设输入和输出形状都是(n, d)，

[1] To be fully consistent with notation, the output of (E.6.1) is a row vector, while we defined the gradient as a column vector. We will ignore this subtle point for simplicity until it is time to define vector-Jacobian products later on the in the chapter.

n是迷你批维度，d表示输入/输出特征。然后，矩阵乘法的复杂度为$\mathcal{O}(n^2d^2p_j)$，它在迷你批大小和特征维度上都是二次的。这很容易变得不可行，尤其是在高维输入，如图像的情况下。

我们可以通过注意到算法的最后一个操作是一个更简单的矩阵-向量乘积，这是由于有标量输出所致，从而获得更好的权衡。这一点在下一节中进行了更详细的探讨。

6.3 反向模式求导

为了进行，我们展开计算与第 i 个权重矩阵对应的单个梯度项：

$$\nabla_{\mathbf{w}_i} y = \mathbf{1}^\top \big[\partial_{\mathbf{h}_{l-1}} \mathbf{h}_l\big] \cdots \big[\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}\big] \big[\partial_{\mathbf{w}_i} \mathbf{h}_i\big] \tag{E.6.2}$$

记住，除了符号之外，（E.6.2）只是一个可能很长的矩阵乘法序列，涉及一个常数项（一个全为1的向量），一系列输入雅可比矩阵（红色项）以及相应权重矩阵的权重雅可比矩阵（绿色项）。让我们为红色项定义一个简称：

$$\widetilde{\mathbf{h}}_i = \mathbf{1}^\top \prod_{j=i+1}^{l} \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j \tag{E.6.3}$$

因为矩阵乘法是结合律的，所以我们可以以任何顺序执行(E.6.2)中的计算。在F-AD中，我们从右到左进行，因为这对应于原始函数执行的顺序。然而，通过注意到两个有趣的方面，我们可以做得更好：

1. (E.6.2) 中最左边的项是向量和矩阵的乘积（这是由于输出中存在标量项的结果），其计算性能优于两个矩阵的乘积。其输出也是一个向量。2. (E.6.3) 中的项（从层 i 到层 l 所有输入雅可比的乘积）可以从 *last* 项开始递归计算，并依次乘以输入雅可比，反向迭代。

我们可以将这些观察结果结合起来，开发一种称为反向模式自动微分（R-AD）的第二种自动微分方法，其概述如下。

1. 与F-AD不同，我们首先执行要微分的*entire*程序，存储所有中间输出。

2. 我们初始化一个向量 $\tilde{\mathrm{h}} = 1^{\top}$，它对应于 (E.6.2) 中的最左端项。

3. 以反向顺序移动，即对于索引 i 在 l 范围内，$l-1$，$l-2$，...，1，我们首先计算相对于第 i 个权重矩阵的梯度，如下：

$$\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} \left[\partial_{\mathbf{w}_i} \mathbf{h}_i \right]$$

这是所需的 i-阶梯度。接下来，我们更新我们的“反向传播”输入雅可比矩阵，如下：

$$\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} \left[\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i \right]$$

步骤（1）-（3）描述了一个大致的程序

与原始程序对称，我们称之为对偶程序或逆程序。术语$\tilde{\text{h}}$称为伴随项，它们存储（按顺序）我们程序中关于变量h_1，h_2，...，h_l的所有输出梯度.[2]

在神经网络术语中，我们有时会说原始（原初）程序是前向传递（不要与前向模式混淆），而反向程序是后向传递。与F-AD不同，在R-AD中，必须先执行完整的原初程序，然后才能运行反向程序，我们需要专门的机制来存储所有中间输出以“展开”计算图。不同的框架以不同的方式实现这一点，如下所述。

计算上，R-AD 比 F-AD 效率显著更高。特别是，R-AD 步骤（3）中的两个操作都是仅在线性形状量上缩放的向量矩阵乘法。权衡是执行 R-AD 需要大量的内存，因为原程序的所有中间值都必须使用合适的策略存储在磁盘上。可以使用特定的技术，如梯度检查点，通过增加计算量并部分减少内存需求来改善这种权衡。这是通过仅存储少量中间输出（称为检查点）并在反向传播期间重新计算剩余值来实现的。参见图 F.6.1 以获得可视化。

[2]Compare this with F-AD, where the tangents represented instead the gradients of the $\mathbf{h}_i$ variables with respect to the weights.

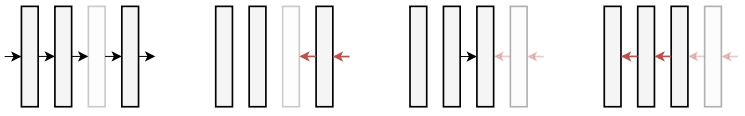


图 F.6.1：*An example of **gradient checkpointing**. (a) We execute a forward pass, but we only store the outputs of the first, second, and fourth blocks (**checkpoints**). (b) The backward pass (red arrows) stops at the third block, whose activations are not available. (c) We run a second forward pass starting from the closest checkpoint to materialize again the activations. (d) We complete the forward pass. Compared to a standard backward pass, this requires 1.25x more computations. In general, the less checkpoints are stored, the higher the computational cost of the backward pass.*

6.4 实际考虑因素

6.4.1 向量-雅可比积

在R-AD算法的第（3）步中观察，我们可以得出一个有趣的结论：我们需要的唯一操作是行向量v与f (雅可比矩阵)之间的乘积，无论是输入还是权重雅可比矩阵)。我们称这两个操作为f向量-雅可比乘积（VJPs）[3]。在下一个定义中，我们通过添加转置来恢复维度一致性。

[3]By contrast, F-AD can be formulated entirely in terms of the transpose of the VJP, called a **Jacobian-vector product** (JVP). For a one-dimensional output, the JVP is the directional derivative (E.2.20) from Section 2.2. Always by analogy, the VJP represents the application of a linear map connected to infinitesimal variations of the *output* of the function, see [BR24].

定义 D.6.2（向量-雅可比乘积（VJP））

Given a function $\mathrm{y} = f(\mathrm{x})$*, with* $\mathrm{x} \sim (c)$ *and* $\mathrm{y} \sim (c')$*, its VJP is another function defined as:*

$$\mathrm{vjp}_f(\mathbf{v}) = \mathbf{v}^\top \partial f(\mathbf{x}) \tag{E.6.4}$$

where $\mathrm{v} \sim (c')$*. If* f *has multiple parameters* $f(\mathrm{x}_1, \ldots, \mathrm{x}_n)$*, we can define n individual VJPs denoted as* $\mathrm{vjp}_{f,\mathbf{x}_1}(\mathrm{v}), \ldots, \mathrm{vjp}_{f,\mathbf{x}_n}(\mathrm{v})$.

特别地，在我们的情况下，我们可以定义两种类型的VJP，对应于原函数对输入和权重参数的偏导数：

$$\mathrm{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) \tag{E.6.5}$$

$$\mathrm{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w}) \tag{E.6.6}$$

我们现在可以将R-AD算法步骤（3）中的两个操作重写为对原始函数的两次VJP调用，带有伴随值（为了可读性，忽略*i*索引），对应于伴随乘以权重VJP，以及伴随乘以输入VJP：

$$\partial_{\mathbf{w}} y = \mathrm{vjp}_{f,\mathbf{w}}\left(\widetilde{\mathbf{h}}\right) \tag{E.6.7}$$

$$\widetilde{\mathbf{h}} \leftarrow \mathrm{vjp}_{f,\mathbf{h}}\left(\widetilde{\mathbf{h}}\right) \tag{E.6.8}$$

因此，我们可以通过首先选择一组原语操作，然后通过相应的VJP对其进行增强，来实现整个自动微分系统，而无需在任何时候将雅可比矩阵在内存中具体化。这被展示为

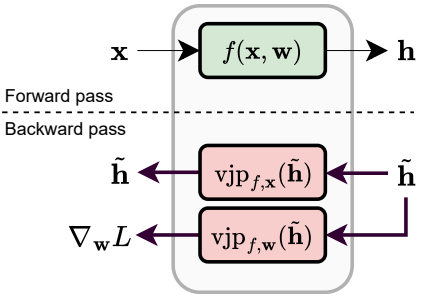


图 F.6.2：*For performing R-AD, primitives must be augmented with two VJP operations to be able to perform a backward pass, corresponding to the input VJP* (E.6.5) *and the weight VJP* (E.6.6). *One call for each is sufficient to perform the backward pass through the primitive, corresponding to* (E.6.7)-(E.6.8).

图F.6.2中示意。

实际上，我们可以通过反复调用VJPs和基向量e_1，...，e_n来恢复雅可比矩阵的计算，一次生成一行，例如，对于输入的雅可比矩阵，我们有：

$$\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w}) = \begin{bmatrix} \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_1) \\ \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_2) \\ \vdots \\ \text{vjp}_{f,\mathbf{x}}(\mathbf{e}_n) \end{bmatrix}$$

为了理解这种重新表述为何方便，让我们看看全连接层的VJP，它由线性投影和（逐元素）非线性组成。首先，考虑一个没有偏置的简单线性投影：

$$f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$$

输入雅可比矩阵在这里是简单的W，但权重雅可比是一个秩为3的张量（第2.2节）。相比之下，

输入VJP没有特殊结构：

$$\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^\top \mathbf{W}^\top = [\mathbf{W}\mathbf{v}]^\top \tag{E.6.9}$$

权重VJP，反而是一个简单的外积，完全避免了三阶张量：

$$\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}\mathbf{x}^\top \tag{E.6.10}$$

计算 VJP

为了计算（E.6.10），我们可以写出$y = \mathrm{v}^\top \mathrm{W}\mathrm{x} = \sum_i \sum_j W_{ij} v_i x_j$，从中我们立即得到$\frac{\partial y}{\partial W_{ij}} = v_i x_j$，这是外积的逐元素定义。

因此，每次我们在正向传播中应用线性投影，我们通过其权重的转置修改反向传播的梯度，并执行外积来计算W的梯度。

考虑现在一个没有可训练参数的逐元素激活函数，例如ReLU：

$$f(\mathbf{x}, \{\}) = \phi(\mathbf{x})$$

因为我们没有可训练的参数，我们只需要考虑输入VJP。梯度是一个对角矩阵，其元素是ϕ的导数：

$$[\partial_{\mathbf{x}} \phi(\mathbf{x})]_{ii} = \phi'(x_i)$$

输入VJP是对角矩阵与向量的乘积，相当于一个Hadamard积（即，一个

```
# 原始函数（平方和）def f(x: Float[Array, "c"]): re
turn (x**2).sum() grad_f = func.grad(f) print(grad_f(t
orch.randn(10)).shape) # [Out]: torch.Size([10])
```

盒子 C.6.2: *Gradient computation as a higher-order function. The* torch.func *interface replicates the JAX API. In practice, the function can be traced (e.g., with* torch.compile*) to generate an optimized computational graph.*

缩放操作):

$$\text{vjp}_{\mathbf{x}}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x}) \tag{E.6.11}$$

有趣的是，在这种情况下，我们也可以在不实际化整个对角矩阵的情况下计算VJP。

6.4.2 实现R-AD系统

存在许多实现R-AD系统的方法，从Wengert列表（如TensorFlow中所示）到源到源代码转换 [GW08]。在此，我们简要讨论一些现有框架中的常见实现。

首先，将原语描述为具有两个参数 $f(\text{x}, \text{w})$ 的函数，与 JAX 等函数式框架相一致，其中一切都是函数。考虑一个具有 $f(\text{x})$ 的函数，输入维度为 c，输出维度为 c'。从这个角度来看，VJP 可以实现为一个具有以下签名的更高阶函数：

$$(\mathbb{R}^c \to \mathbb{R}^{c'}) \to \mathbb{R}^c \to (\mathbb{R}^{c'} \to \mathbb{R}^c) \tag{E.6.12}$$

即，给定一个函数f和一个输入 x′，VJP 返回另一个可以应用于c'-维向量 v 并返回 $\mathrm{v}^\top \partial f(\mathrm{x}')$ 的函数。同样，一维函数的梯度可以实施为另一个具有签名的更高阶函数：

$$(\mathbb{R}^c \to \mathbb{R}) \to (\mathbb{R}^c \to \mathbb{R}^c) \tag{E.6.13}$$

以函数$f(\mathrm{x})$作为输入并返回另一个计算 $\nabla f(\mathrm{x})$ 的函数。在 JAX 中，这些想法分别通过函数 jax.grad 和 jax.jvp 实现，在 PyTorch 的 torch.func 模块中也有相应的实现 - 请参阅 Box C.6.2 以获取示例.[4]

如我们所述，在实践中，我们的模型被实现为对象的组合，其参数作为属性封装（Box C.5.1）。一种可能性是将对象“净化”，使其成为一个纯函数，例如：[5]

```
# Extract the parameters
params = dict(model.named_parameters())
# Functional call over the
# model's forward function
y = torch.func.functional_call(
                model, params, x
                )
```

更一般地，像PyTorch这样的框架通过技术直接处理这种场景，而不引入中间操作。例如，在PyTorch中，张量对象被添加了有关生成它们的操作的信息（图F.6.3,

[4]Many operations, such as computing an Hessian, can be achieved by smartly composing JVPs and VJPs based on their signatures: https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html.

[5]https://sjmielke.com/jax-purify.htm

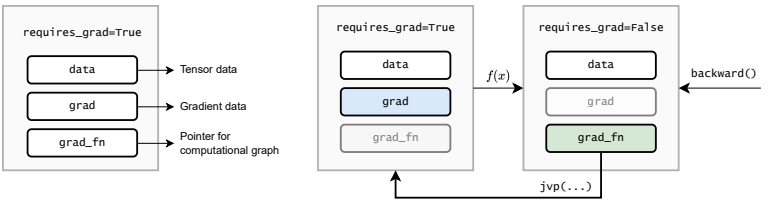


图 F.6.3: *Left: in PyTorch, a tensor is augmented with information about its gradient (empty at initialization), and about the operation that created it. Right: during a backward pass, the* grad_fn *property is used to traverse the computational graph in reverse, and gradients are stored inside the tensor's* grad *property whenever* requires_grad *is explicitly set to* True *(to avoid consumming unnecessary memory).*

左）。每当请求对标量值调用 backward() 时，这些属性被用来反向遍历计算图，存储相应的梯度 *inside* 以及需要它们的张量（图 F.6.3，右）。

这只是对这些系统在实际中如何实现的概述，我们省略了许多细节，具体内容请参考官方文档.[6]

6.4.3 选择激活函数

偶然间，我们现在可以解释为什么ReLU作为激活函数是一个好的选择。仔细观察（E.6.11）告诉我们，每次我们在模型中添加一个激活函数，反向传播中的伴随变量都会按因子ϕ'(x)缩放。对于具有许多层的模型，这可能导致两种病态行为：

[6] I definitely suggest trying to implement an R-AD system from scratch: many didactical implementations can be found online, such as `https://github.com/karpathy/micrograd`.

1. 如果 $\phi'(\cdot) <$ 在任何地方都为 1，那么梯度在层数增加时可能会以指数速度迅速减小到 0。这被称为梯度消失问题。

2. 相反，如果 $\phi'(\cdot) >$ 在任何地方都为1，则会出现相反的问题，梯度在层数上指数级地趋向于无穷大。这被称为梯度爆炸问题。

这些在实践中是严重问题，因为库使用有限精度（通常是32位或更低）来表示浮点数，这意味着在增加层数时，下溢或上溢会迅速显现。

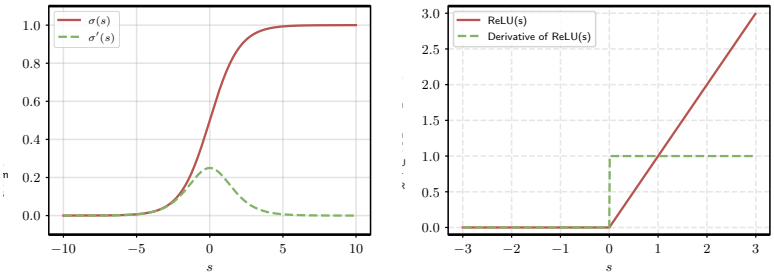
线性非线性模型

令人惊讶的是，在浮点精度下实现的线性层堆栈并非完全线性，因为机器精度存在小的间断！这通常不是问题，但可以利用这一点来训练完全线性的深度神经网络。[a]

[a] https://openai.com/research/深度线性网络中的非线性计算

作为一个消失梯度出现的例子，考虑sigmoid函数 $\sigma(s)$。我们之前已经提到，这在过去是一个常见的AF，因为它是对阶跃函数的软近似。我们还知道 $\sigma'(s) = \sigma(s)(1-\sigma(s))$。结合 $\sigma(s) \in [0,1]$ 的实际情况，我们得到：

$$\sigma'(s) \in [0, 0.25]$$



(a) *Sigmoid* **(b)** *ReLU*

图 F.6.4: *(a) Plot of the sigmoid function (red) and its derivative (green). (b) Plot of ReLU (red) and its derivative (green).*

因此，Sigmoid是梯度消失问题的首选候选者：参见图F.6.4a。

设计一个永远不会出现消失或爆炸梯度的AF是非平凡的，因为唯一在每个地方都有$\phi'(s) = 1$的函数是恒等函数。然后我们需要一个“足够线性”的函数来避免梯度问题，但“足够非线性”以分离线性层。ReLU最终成为了一个好的候选者，因为：

$$\partial_s \text{ReLU}(s) = \begin{cases} 0 & s < 0 \\ 1 & s > 0 \end{cases}$$

梯度要么被置零，导致计算稀疏，要么乘以1，避免缩放问题——这如图F.6.4b所示。

作为旁注，ReLU的梯度在无论我们用ReLU层的输出替换其输入时都相同（因为我们只屏蔽了负值，而保留了正值不变）。因此，使用ReLU作为激活函数的另一个好处是，在执行R-AD时，我们可以通过在正向传播中覆盖层的输入来节省一点内存。

影响AD过程的正确性：例如，在PyTorch中通过设置in_place参数来实现此操作.[7]

6.4.4 子可微性和AD

存在一个我们至今未讨论的小细节：ReLU 在 0 处不可导，这使得整个网络不光滑。在这种情况下会发生什么？“实用”的答案是，通过从随机（非零）初始化进行随机梯度下降最小化，最终恰好落在 $s = 0$ 的概率实际上为零，而梯度在 $\text{ReLU}(\varepsilon)$ 中对任何 $|\varepsilon| > 0$ 都是定义的。

对于更技术性的回答，我们可以引入函数子梯度的概念。

> 定义 D.6.3（子梯度）
>
> *Given a convex function $f(x)$, a subgradient in x is a point z such that, for all y:*
>
> $$f(y) \geq f(x) + z(y - x)$$

注意与凸性定义的相似性：子梯度是“切线”于 $f(x)$ 的直线的斜率，使得整个 f 都被它下界限制。如果 f 在 x 上可微，那么只有一条这样的直线存在，它是 f 在 x 上的导数。在非光滑点，存在多个子梯度，它们形成一个称为的集合

[7] https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

f 在 x 中的下导数：

$$\partial_x f(x) = \{z \mid z \text{ is a subgradient of } f(x)\}$$

使用这个定义，我们可以通过在0处用其次微分替换梯度来完成ReLU梯度的分析：

$$\partial_s \text{ReLU}(s) = \begin{cases} \{0\} & s < 0 \\ \{1\} & s > 0 \\ [0,1] & s = 0 \end{cases}$$

因此，[0, 1]中的任何值都是0的有效子梯度，实践中大多数实现都倾向于$\text{ReLU}'(0) = 0$。在迭代下降过程的每一步选择子梯度被称为子梯度下降。

实际上，情况甚至更加复杂，因为对于非凸函数，子梯度不一定有定义。在这种情况下，可以求助于推广，将之前的定义放宽到 x 的局部邻域，例如Clarke子微分.[8] 子可微性也可能在AD中引起问题，其中相同函数的不同实现可以提供不同的（可能是无效的）子梯度，并且必须考虑更精细的链规则概念以进行正式证明 [KL18，BP20]。[9]

[8] https://en.wikipedia.org/wiki/Clarke_generalized_derivative

[9] Consider this example reproduced from [BP20]: define two functions, $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ and $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$. They are both equivalent to ReLU, but in PyTorch a backward pass in 0 returns 0.0 for ReLU, 1.0 for ReLU_2, and 0.5 for ReLU_3.

从理论到实践

如果您已跟随第5章中的练习，您已经看到了R-AD在PyTorch和JAX中的应用，本章（特别是6.4.2节）应该已经阐明了它们的实现。

一个好的想法是尝试重新实现一个简单的R-AD系统，类似于PyTorch中的系统。例如，专注于标量值量，micrograd存储库[10]是一个非常好的教学实现。我们唯一没有涉及到的细节是，一旦你转向通用无环图，在反向传播之前对计算图中变量的排序是至关重要的，以避免创建错误的反向传播路径。在micrograd中，这是通过变量的非昂贵拓扑排序来实现的。

它也很有趣尝试在PyTorch中实现一个新的原语（在本章使用的意义上），这需要指定其正向传播以及其VJPs。[11]一个例子可以是第5.4节中的一个可训练激活函数。这是一个教学练习，从意义上说，这可以通过子类化nn.Module并让PyTorch的AD引擎计算出反向传播来实现。

所有这些步骤也可以在JAX中复制：

- 实现一个具有教学版本的 JAX，使用{v*}

[10] https://github.com/karpathy/micrograd

[11] https://pytorch.org/docs/master/notes/extending.html

autodidax.[12]

- 编写一个新原语，通过实现相应的VJP来实现.[13]

- 阅读 JAX 自动微分食谱[14]以发现自动微分引擎的高级用例，例如高阶导数、Hessian 矩阵等。

[12] https://jax.readthedocs.io/en/latest/autodidax.html

[13] https://jax.readthedocs.io/en/latest/notebooks/Custom_derivative_rules_for_Python_code.html

[14] https://jax.readthedocs.io/en/latest/notebooks/autodiff_cookbook.html

第二部分
一片陌生的土地

“Curiouser and curiouser!” cried Alice
(she was so much surprised, that for
the moment she quite forgot
how to speak good English).

— 第二章，泪池

7 | 卷积层

> 关于本章
>
> 在这一章中，我们介绍了我们的第二个核心层，即卷积层，该层旨在通过利用我们称为 *locality* 和 *parameter sharing* 的两个关键思想来与图像（或更一般地说，任何类型的顺序数据）一起工作。

全连接层在历史上很重要，但从实际角度来看则不然：在非结构化数据（我们通常称之为表格数据，因为它可以很容易地表示为表格）中，MLP通常不如其他替代方案，如随机森林或调优良好的支持向量机 [GOV22]。然而，一旦我们考虑其他类型的数据，这些数据具有一些可以在层的设计和模型设计中加以利用的结构，情况就不再是如此了。

在这一章中，我们考虑图像域，而在下一章中，我们还将考虑将其应用于时间序列、音频、图和视频。在这些所有情况下，输入都具有序列结构（无论是时间、空间还是其他类型），可以利用这种结构来设计既高效、易于组合又高度Translated Text: 在本章中，我们考虑图像域，而在下一章中，我们还将考虑将其应用于时间序列、音频、图和视频。在这些所有情况下，输入都具有序列结构（无论是时间、空间还是其他类型），可以利用这种结构来设计既高效、易于组合又高度

在参数方面效率高。有趣的是，我们将看到可以通过以全连接层为起点来设计可能的解决方案，然后根据输入的性质适当地限制或泛化它。

7.1 向卷积层迈进

7.1.1 完全连接层不足以

一张图像可以用张量 $X \sim (h, w, c)$ 来描述，其中 h 是图像的高度，w 是图像的宽度，c 是通道数（对于黑白图像可以是 1，对于彩色图像是 3，或者对于例如超光谱图像可以是更高的数值）。因此，图像的小批量通常具有 4 个维度，并且还有一个额外的批处理维度 (b, h, w, c)。这三个维度并不相同，因为 h 和 w 代表 *pixels* 的空间排列，而通道 c 没有特定的顺序，从某种意义上说，将图像以 RGB 或 GBR 格式存储只是惯例问题。

> 关于符号、信道和特征
>
> 我们使用与表格情况中特征相同的符号（c），因为在模型设计中它将扮演类似的角色，即我们可以将每个像素视为由一组通用的c *features*描述，这些c *features*由模型的层并行更新。因此，卷积层将返回一个通用的张量 $(h、w、c')$，每个hw个像素都有一个大小为c'的嵌入。

为了使用全连接层，我们需要

将图像“展平”（向量化）：{v*}

$$\mathbf{h} = \phi(\mathbf{W} \cdot \underbrace{\text{vect}(X)}_{\text{Flattened image}}) \tag{E.7.1}$$

在PyTorch中，矢量(x)等价于x.reshape(-1)，它为通用的秩-n张量$x \sim (i_1, i_2, ..., i_n)$返回一个等价张量x $\sim \left(\prod_{j=1}^{n} i_j\right)$。

尽管这应该很清楚这是一个不优雅的方法，但强调其一些缺点是值得的。首先，我们失去了前一部分的一个非常重要的属性，即可组合性：我们的输入是一个图像，而我们的输出是一个向量，这意味着我们无法连接两个这样的层。我们可以通过将输出向量重塑为图像来恢复这一点：

$$H = \text{unvect}(\phi(\mathbf{W} \cdot \text{vect}(X))) \tag{E.7.2}$$

在假设层不修改像素数量的情况下，我们将输出重塑为 $(h$、w、$c')$ 张量，其中 c' 是一个超参数。

这直接导致第二个问题，即该层有 *huge* 个参数。例如，考虑一个 (1024, 1024) 的 RGB 图像，保持输出维度不变，结果得到 $(1024 * 1024 * 3)^2$ 个参数（或 $(hw)^2cc'$ 一般而言），这达到了 10^{13} 的数量级！我们可以这样解释前一层：对于每个像素，输出中的每个通道都是输入图像中 *all* 个 *all* 像素的加权组合。正如我们将看到的，我们可以通过限制这种计算来获得更有效的解决方案。

更多关于重塑

为了展平（或更一般地，重塑）一个张量，我们需要决定一个处理值的顺序。在实践中，这取决于张量在内存中的存储方式：在大多数框架中，张量的数据以连续的内存块顺序存储，这被称为步幅布局。考虑以下示例：

```
torch.randn(32, 32, 3).stride() # (96, 3, 1)
```

步长是在内存中移动一个位置所需的步数，即张量的最后一个维度是连续的，而要在第一个维度中移动一个位置，则需要96（$32 * 3$）步。这被称为行主序排列，或者在图像分析中，称为光栅顺序.[a]每次重塑操作都是通过沿着这个步长表示进行移动。

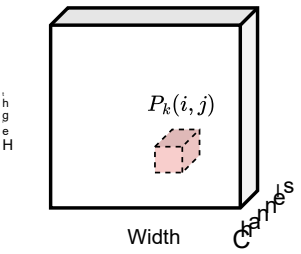
[a] https://zh.wikipedia.org/wiki/光栅扫描

作为一个可视化的运行示例，考虑一个1D序列（我们将在稍后更深入地考虑1D序列；现在，你可以将其视为“*4 pixels with a single channel*”）：

$$\mathbf{x} = \left[x_1, x_2, x_3, x_4\right]$$

在这种情况下，我们不需要任何重塑操作，并且前一层（带有 $c' = 1$）可以写成：

图 F.7.1： *Given*， w，
a tensor (h *e) and a* ， j) *(shown in*2
maximum distance) is the $(k+1, 2k+1, c)$
patch $P_k(i$ *tensor collecting all pixels at distance at most k from the pixel in position* (i, j).



$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} \\ W_{21} & W_{22} & W_{23} & W_{24} \\ W_{31} & W_{32} & W_{33} & W_{34} \\ W_{41} & W_{42} & W_{43} & W_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

7.1.2 本地层

像素的空间排列引入了像素之间的度量（距离）。虽然有许多有效的“距离”概念，但我们将发现使用以下定义很方便，该定义将像素 $(i、j)$ 和 $(i'、j')$ 之间的距离定义为两个轴之间的最大距离：

$$d((i,j),(i',j')) = \max(|i-i'|, |j-j'|) \tag{E.7.3}$$

如何在我们对层的定义中利用这个想法？理想情况下，我们可以想象一个像素对另一个像素的影响随着它们距离的倒数而减小。将这个想法推向极致，我们可以假设当距离大于某个阈值时，影响实际上为零。为了使这一洞察正式化，我们引入了补丁的概念。

定义 D.7.1（图像块）

Given an image X, we define the ***patch*** $P_k(i, j)$ *as the sub-image centered at* (i, j) *and containing all pixels at distance equal or lower than* k*:*

$$P_k(i, j) = [X]_{i-k:i+k, j-k:j+k, :}$$

where distance is defined as in (E.7.3)*. This is shown visually in Figure F.7.1.*

仅对至少距离图像边缘 k 个步骤的像素定义有效：我们现在忽略这个点，稍后再讨论。每个补丁的形状为 (s, s, c)，其中 $s = 2k + 1$，因为我们考虑每个方向上的 k 个像素以及中心像素。由于稍后将会解释的原因，我们称 s 为滤波器大小或核大小。

考虑一个通用的层 $H = f(X)$，它以形状为 $(h、w、c)$ 的张量作为输入，并返回形状为 $(h、w、c')$ 的张量。如果给定像素的输出只依赖于一个预定大小的块，我们称该层为局部层。

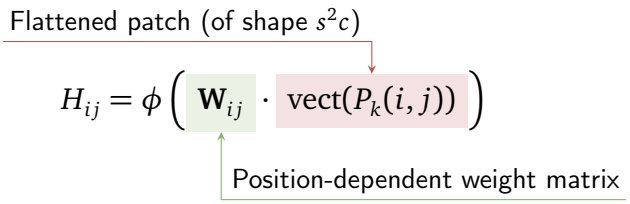
定义 D.7.2（局部层）

Given an input image $X \sim (h, w, c)$, a layer $f(X) \sim (h, w, c')$ is ***local*** *if there exists a k such that:*

$$[f(X)]_{ij} = f(P_k(i, j))$$

This has to hold for all pixels of the image.

我们可以通过将属于每个像素影响区域（感受野）外像素的所有权重设置为0，将层（E.7.1）转换为局部层：



我们称这类层为局部连接层。请注意，对于每个输出像素，我们有一个不同的权重矩阵 $\mathrm{W}_{ij} \sim (c', ssc)$，从而得到 $hw(s^2cc')$ 个参数。相比之下，在初始层中我们有 $(hw)^2cc'$ 个参数，参数数量减少了 $\frac{s^2}{hw}$ 倍。

考虑到我们的玩具示例，假设例如 $k = 1$（因此 $s = 3$），我们可以将结果操作写成：

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{12} & W_{13} & 0 & 0 \\ W_{21} & W_{22} & W_{23} & 0 \\ 0 & W_{31} & W_{32} & W_{33} \\ 0 & 0 & W_{41} & W_{42} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

我们的操作在 x_1 和 x_4 上未定义，在这种情况下，我们通过删除对应于未定义操作的权重来考虑一个“缩短”的滤波器。等价地，您可以在必要时在边界处添加 0：

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_{11} & W_{12} & W_{13} & 0 & 0 & 0 \\ 0 & W_{21} & W_{22} & W_{23} & 0 & 0 \\ 0 & 0 & W_{31} & W_{32} & W_{33} & 0 \\ 0 & 0 & 0 & W_{41} & W_{42} & W_{43} \end{bmatrix} \begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}$$

这种技术称为零填充。在图像中，对于一个大小为$2k+1$的核，我们需要在每个边恰好有k行和列的0，以确保对每个像素的操作有效。否则，输出无法在边界附近计算，输出张量的形状将为$(h-2k,\ w-2k,\ c')$。这两种选项在大多数框架中都是有效的。

> 关于我们对补丁的定义
>
> 卷积的定义采用补丁的概念略显不寻常，但我发现这极大地简化了符号。我稍后会提供一个更传统的、面向信号处理的定义。这两个定义是等价的，可以互换使用。以补丁为导向的定义需要奇数核大小，不允许使用偶数核大小，但在实践中这些情况很少见。

7.1.3 翻译等变性及卷积层

在一个局部连接层中，两个相同的补丁根据其位置可能会产生不同的输出：例如，像素 (5, 2) 上的某些内容将不同于像素 (39, 81) 上的相同内容，因为

两个矩阵 $W_{5,2}$ 和 $W_{39,81}$ 是不同的。然而，大部分情况下，我们可以假设这个信息是不相关的：非正式地说，“一匹马就是一匹马”，无论它在输入图像中的位置如何。我们可以用称为平移等变性的属性来形式化这一点。

定义 D.7.3（翻译等变性）{v*}

We say that a layer $H = f(X)$ *is* ***translation equivariant*** *if translations of the inputs imply an equivalent translation of the output:*

Identical patches

$$P_k(i,j) = P_k(i',j') \text{ implies } f(P_k(i,j)) = f(P_k(i',j'))$$

Identical outputs

要理解命名法，请注意，我们可以将前面的定义解释如下：每当一个对象在图像上从位置 $(i、j)$ 移动到位置 $(i'、j')$ 时，我们在 $(i、j)$ 中拥有的输出 $f(P_k(i、j))$ 现在将在 $f(P_k(i'、j'))$ 中找到。因此，层的激活与输入具有相同的 (*èqui* 在拉丁语中) 平移运动。我们将在稍后更正式地定义等变性和不变性。

一种实现平移等变性简单的方法是通过权重共享，即让每个位置共享同一组权重：

$$H_{ij} = \phi(\mathbf{W} \cdot \text{vect}(P_k(i,j)))$$

Weight matrix independent of (i,j)

这是一个卷积层，它极其

在参数方面高效：我们只有一个形状为 $(c'、ssc)$ 的权重矩阵 W，它与原始图像的分辨率无关（再次，这与仅具有 $hw(s^2c'c)$ 参数的局部连接层形成对比：我们通过另一个因子$\frac{1}{hw}$ 减少了它们）。我们可以通过添加一个形式为偏置向量 $\mathrm{b} \sim (c')$ 的 c' 个额外参数来写出带有偏差的变体。由于其重要性，我们下面重新陈述该层的完整定义。

定义 D.7.4（卷积层）

Given an image $X \sim (h,\ w,\ c)$ *and a kernel size* $s = 2k + 1$*, a* ***convolutional layer*** $H = \mathrm{Conv2D}(X)$ *is defined element-wise by:*

$$H_{ij} = \mathbf{W} \cdot \mathrm{vect}(P_k(i,j)) + \mathbf{b} \tag{E.7.4}$$

The trainable parameters are $\mathrm{W} \sim (c',\ ssc)$ *and* $\mathrm{b} \sim (c')$.
The hyper-parameters are k, c′, and (eventually) whether to apply zero-padding or not. In the former case the output has shape $(h,\ w,\ c')$*, in the latter case it has shape* $(h - 2k,\ w - 2k,\ c')$.

查看框C.7.1以获取代码示例。等效的面向对象实现可以在torch.nn.Conv2D中找到。相比之下，我们的玩具示例可以如下改进：

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \end{bmatrix} = \begin{bmatrix} W_2 & W_3 & 0 & 0 \\ W_1 & W_2 & W_3 & 0 \\ 0 & W_1 & W_2 & W_3 \\ 0 & 0 & W_1 & W_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{E.7.5}$$

```
from torch.nn import functional as F x = torch.randn(1
6, 3, 32, 32) w = torch.randn(64, 3, 5, 5) F.conv2d(x, w
, padding='same').shape # [Out]: torch.Size([16, 64, 32,
32])
```

盒子C.7.1: *Convolution in PyTorch. Note that the channel dimension is – by default – the first one after the batch dimension. The kernel matrix is organized as a* (c', c, k, k) *tensor. Padding can be specified as an integer or a string ('same' meaning that the output must have the same shape as the input, 'valid' meaning no padding).*

我们现在只有三个权重 $\mathrm{W} = [W_1、W_2、W_3]^\top$ (，零填充版本与之前相同，为了简洁我们省略它)。这个权重矩阵具有特殊结构，其中任何对角线上的元素都是常数（例如，在主对角线上我们只找到 W_2）。我们称这些矩阵为托普利茨矩阵[1]，它们对于在现代硬件上正确实现卷积层至关重要。托普利茨矩阵是结构化稠密矩阵 [QPF+24]的例子。方程（E.7.5）还应阐明，卷积仍然是一个 *linear* 操作，尽管与全连接矩阵相比，权重矩阵受到高度限制。

卷积和术语

我们的术语主要来自信号处理。我们可以通过将卷积层的输出重写为更标准的形式来理解这一点。为此，我们首先将权重矩阵重新排列成一个形状为 $(s、s、c、c')$ 的等效权重张量 W，类似于 Box C.7.1 中 PyTorch 的实现。为了方便，我们还

[1] https://en.wikipedia.org/wiki/Toeplitz_matrix

定义一个将区间 $[1, \ldots, 2k+1]$ 中的整数 i' 转换为区间 $[i-k, \ldots, i+k]$ 的函数：

$$t(i) = i - k - 1 \tag{E.7.6}$$

在 $t($ 的参数中隐含 k •). 现在我们用显式的轴上求和重写层的输出：

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \tag{E.7.7}$$

仔细检查索引：对于给定的像素 (i、j) 和输出通道 z (，一个从 1 到 c') 的自由索引，在空间维度 W 上必须按 1、2、...、$2k$+1 索引，而 X 必须按 $i-k$、$i-k+1$、...、$i+k-1$、$i+k$ 索引。索引 d 代替在输入通道上运行。

从信号处理的角度来看，方程（E.7.7）对应于通过一组有限脉冲响应（FIR）滤波器 [Unc15] 对输入信号 X 进行滤波操作（除符号变化外）。这里的每个滤波器对应于权重矩阵的一个切片 $W_{:,:,:,i}$。在标准信号处理中，这些滤波器可以手动设计以在图像上执行特定操作。例如，一个 3 × 3 检测脊的滤波器可以写成：[2]

$$W = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

在卷积层中，相反，这些滤波器以随机方式初始化并通过梯度下降进行训练。我们考虑

[2] https://en.wikipedia.org/wiki/Kernel_(image_processing)

下一节将介绍基于卷积层的卷积模型的设计。卷积层的一个有趣方面是，其输出保持一种“空间一致性”，并且可以绘制出来：我们将输出中的一个切片 $H_{:,:,i}$ 称为层的激活图，表示特定滤波器在每个输入区域“激活”的程度。我们将在下一卷中更详细地探讨这些图的探索。

7.2 卷积模型

7.2.1 设计卷积“块”

有了卷积层的定义，我们现在转向构建卷积模型的任务，也称为卷积神经网络（CNNs）。我们考虑图像分类问题，尽管我们所说的很多内容可以扩展到其他情况。首先，我们正式化感受野的概念。

> 定义 D.7.5（感受野）
>
> *Denote by X an image, and by $H = g(X)$ a generic intermediate output of a convolutional model, e.g., the result of applying 1 or more convolutional layers. The* ***receptive field*** *$R(i,\ j)$ of pixel $(i,\ j)$ is the subset of X which contributed to its computation:*
>
> $$[g(X)]_{ij} = g(R(i,j)),\quad R(i,j) \subseteq X$$

对于单个卷积层，像素的感受野等于一个块：$R(i,\ j) = P_k(i,\ j)$。然而，很容易证明对于两个连续的卷积层

具有相同的内核大小，结果感受野为 $R(i,\ j) = P_{2k}(i,\ j)$，然后是三层中的 $P_{3k}(i,\ j)$，以此类推。因此，感受野随着卷积层数量的增加而增加 *linearly*。这促使我们提出局部性的概念：即使单个层在感受野上受到内核大小的限制，但足够大的堆叠仍然会产生一个 *global* 感受野。

C现在考虑一个由两个卷积层组成的序列 rs:

$$H = \text{Conv}(\text{Conv}(X))$$

因为卷积是一种线性操作（参见上一节），这相当于使用更大核大小的单个卷积（如上所述）。我们可以通过以类似全连接层的方式，通过激活函数交错它们来避免这种“折叠”：

$$H = (\phi \circ \text{Conv} \circ \ldots \circ \phi \circ \text{Conv})(X) \tag{E.7.8}$$

为了继续我们的设计，我们注意到在（E.7.8）中，通道维度将被每一层卷积层修改，而空间维度将保持相同的形状（或者如果我们避免零填充，将略微减小）。然而，如果我们的目标是图像分类等，在实践中最终减少这个维度性可能是有利的。

再次考虑一匹马出现在两张不同图像中的两个不同区域的情况。卷积层的平移等变性属性保证，在第一张图像中的区域1中找到的每个特征，将相应地在第二张图像的区域2中找到。然而，如果我们的目标是“马分类”，我们最终需要在图像本身中有一个或多个神经元激活马 *irrespective of where it is found*：如果

我们只考虑平移，这个性质称为平移不变性。

许多在空间维度上减少的操作对平移是平凡的，例如：

$$H' = \sum_{i,j} H_{ij} \text{ or } H' = \max_{i,j}(H_{ij})$$

在CNN的上下文中，这被称为全局池化。然而，这会破坏图像中存在的所有空间信息。我们可以通过部分减少，称为最大池化，获得一个稍微更有效的解决方案。

定义 D.7.6（最大池化层）

Given a tensor $X \sim (h, w, c)$*, a max-pooling layer, denoted as* $MaxPool(X) \sim (\frac{h}{2}, \frac{w}{2}, c)$*, is defined element-wise as:*

$$[\text{MaxPool}(X)]_{ijc} = \max\left([X]_{2i-1:2i,2j-1:2j,c} \right)$$

2×2 *image patch*

因此，我们取输入的2×2个窗口，并独立地计算每个通道的最大值（这可以简单地推广到更大的窗口）。最大池化有效地将空间分辨率减半，同时保持通道数不变。一个例子在图F.7.2中显示。

我们可以通过堆叠多个卷积层并使用最大池化操作来构建一个卷积“块”（见图 F.7.3）：

$$\text{ConvBlock}(X) = (\text{MaxPool} \circ \phi \circ \text{Conv} \circ \ldots \circ \phi \circ \text{Conv})(X)$$

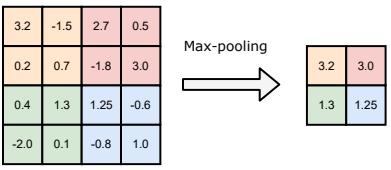


图 F.7.2:
Visualization of 2x2 max-pooling on a (4,4,1) image. For multiple channels, the operation is applied independently on each channel.

通过迭代进行，我们通过堆叠多个此类块来定义一个更复杂的网络：

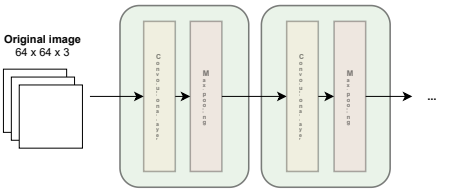
$$H = (\text{ConvBlock} \circ \text{ConvBlock} \circ ... \circ \text{ConvBlock})(X) \quad \text{(E.7.9)}$$

此设计具有大量超参数：每层的输出通道、每层的核大小等。通过做出一些简化的假设，通常可以大幅度减少设计搜索空间。例如，VGG设计[SLJ+15]普及了在每个层中保持滤波器大小恒定的想法（例如，$k = 3$），同时在每个块中保持通道数恒定，并在每个块之间加倍。

一种降低维度替代方法是下采样卷积层的输出：这被称为卷积的步长。例如，步长为1的卷积是正常卷积，而步长为2的卷积将每2个像素计算一个输出像素，步长为3的卷积将每3个像素计算一个输出，依此类推。大步长和最大池化也可以根据整个模型的设计组合在一起。

图 F.7.3:

Abstracting away from "layers" to "blocks" to simplify the design of differentiable models.



不变性和等变性

非正式地，如果 T 是从某个集合（例如所有可能的平移）到 x 的变换，我们称函数 f 是等变如果 $f(Tx) = Tf(x)$，不变如果 $f(Tx) = f(x)$。所有变换的空间形成一个群 [BBL+17]，对应特定变换的矩阵称为该群的表示。卷积层（大致上）按设计对平移是等变的，但还可以找到其他策略来处理更一般的对称形式，例如对群元素进行平均（框架平均，[PABH+21]）。我们将在第10章和第12章看到其他类型层的等变性。

7.2.2 设计完整模型

我们现在可以完成我们模型的设计。通过将多个卷积块堆叠，如（E.7.9）中所示，输出 H 将具有形状 (h', w', c')，其中 w' 和 h' 取决于最大池化操作的次数（或卷积层的步长），而 c' 将仅取决于序列中最后一个卷积层的超参数。请注意，每个元素 H_{ij} 将对应于原始图像中的“宏区域”，例如，如果 $h', w' = 2$，H_{11} 将对应于原始图像中的“左上”象限。我们可以

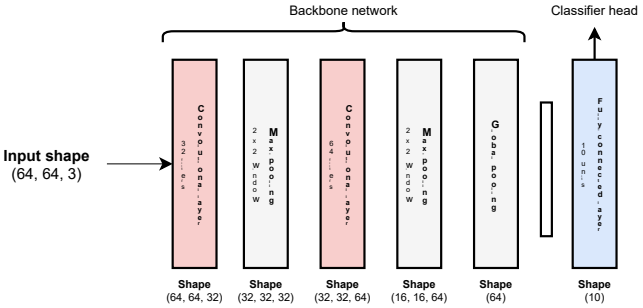


图 F.7.4：*Worked-out design of a very simple CNN for image classification (assuming 10 output classes). We show the output shape for each layer on the bottom. The global pooling operation can be replaced with a flattening operation. The last (**latent**) representation before the classification head is very useful when fine-tuning large-scale pre-trained models – it is an **embedding** of the image in the sense of Section 3.1.1.*

通过在分类之前执行最终的全球池化操作来消除这种空间依赖性。

完整的模型可以分解为三个主要组件：一系列卷积块、全局平均池化和用于分类的最终块。

$$H = (\text{ConvBlock} \circ \ldots \circ \text{ConvBlock})(X) \tag{E.7.10}$$

$$\mathbf{h} = \frac{1}{h'w'} \sum_{i,j} H_{ij} \tag{E.7.11}$$

$$y = \text{MLP}(\mathbf{h}) \tag{E.7.12}$$

MLP(h) 是一个通用的全连接层序列（也可以使用全局池化操作代替展平操作）。这是一个 CNN 的典型示例。参见图 F.7.4 以获取一个具体示例。

这个设计有几个有趣的特性，我们在此列出：

1. 它可以像第4章和第5章中描述的模型一样进行训练。例如，对于分类，我们可以在输出上包裹softmax并通过对交叉熵进行最小化来训练所有参数。第6章中描述的反向传播规则同样适用于此处。

2. 由于全局池化操作，它不依赖于特定的输入分辨率。然而，在训练和推理过程中通常将其固定，以简化小批量处理（下一章将详细介绍可变长度输入）。

3. (E.7.11) 可以被视为一个“特征提取”块，而 (E.7.12) 则是“分类块”。这种解释在我们考虑下一卷中的迁移学习时将非常有用。我们将特征提取块称为模型的骨干，将分类块称为模型的头部。

显著的卷积类型

我们通过提及两种在实践中常见的卷积层实例来结束这一章节。

首先，考虑一个具有 $k = 0$ 的卷积层，即所谓的 1×1 卷积。这对应于通过其通道的加权求和来更新每个像素的嵌入，而忽略所有其他像素：

$$H_{ijz} = \sum_{t=1}^{c} W_{zt} X_{ijt}$$

这是一个有用的操作，例如修改通道维度（我们将在处理时看到一个示例）

残差连接在第9章中）。在这种情况下，参数可以紧凑地表示为矩阵W ∼ (c'，c)。这相当于对每个像素独立应用的全连接层。

其次，考虑一个“正交”变体到1 × 1卷积，其中我们将小邻域内的像素组合起来，但忽略除一个通道之外的所有通道：

$$H_{ijc} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} W_{i',j',c} X_{i'+t(i),j'+t(j),c}$$

在 $t($ 处 $\bullet)$ 这是在（E.7.6）中定义的偏移量。在这种情况下，我们有一个形状为 (s、s、c) 的3秩权重矩阵 W，并且每个输出通道 $H_{:,:,c}$ 仅通过考虑相应的输入通道 $X_{:,:,c}$ 来更新。这被称为深度卷积，可以通过考虑通道组来推广，在这种情况下，它被称为组卷积（深度卷积是组大小等于1的极端情况）。

我们也可以将这两个想法结合起来，得到一个由交替的1 × 1 卷积（以混合通道）和深度卷积（以混合像素）组成的卷积块。这被称为深度可分离卷积，它在针对低功耗设备的CNN中很常见 [HZC+17]。请注意，在这种情况下，单个块（与标准卷积相比）的参数数量从 $sscc'$ 减少到 $ssc + cc'$。我们将在第10章中看到，这些分解，其中输入在单独的轴上交替处理，对于其他类型的架构，如transformers，是基本的。

从理论到实践

所有本章中引入的层（卷积、最大池化）都在torch.nn模块中实现。torchvision库提供了加载数据集和函数，以及应用于图像的接口，这些在下一章中将非常有用.[3]

在继续之前，我建议您遵循并重新实现torchvision中关于图像分类的许多在线教程之一，现在应该相对容易理解。[4] 玩具图像数据集众多，包括MNIST（数字分类）和CIFAR-10（通用图像分类）。将torchvision加载器与Equinox中的层结合使用，您可以在JAX中复制相同的教程，例如：

https://docs.kidger.site/equinox/例子/mnist/.

实现从头开始的卷积也是一个有趣的练习，其复杂度取决于抽象级别。一种可能性是使用PyTorch中的fold/unfold操作来提取补丁。预制的卷积核始终会快得多，这使得这成为一个纯粹的教学练习。

如果您有一些信号处理背景，您可能知道卷积也可以实现为

[3] https://pytorch.org/vision/stable/transforms.html

[4] As an example from the official documentation: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

[5] See for example: https://github.com/loeweX/Custom-ConvLayers-Pytorch

乘法转换为频域。这对于我们倾向于使用的小核来说不切实际，但对于非常大的（也称为 *long*）卷积来说很有用，例如：

https://github.com/fkodom/fft-conv-pytorch

PyTorch还提供了一个可微分的快速傅里叶变换，您可以用作起点。

8 | 超越图像的卷积

关于本章

卷积模型是许多应用中的强大基线模型，远远超出了图像分类。在本章中，我们概述了几个此类扩展，包括用于1D和3D数据的卷积层、文本建模和自回归生成。我们介绍的一些概念（例如，掩码、标记化）在本书的其余部分以及理解现代LLMs中是基本的。

8.1 1D和3D数据的卷积

8.1.1 超越图像：时间序列、音频、视频、文本

在前一章中，我们专注于图像。然而，许多其他类型的数据具有相似的特征，即一个或多个表示时间或空间的“有序”维度，以及一个维度

表示特征（图像情况中的通道）。让我们考虑一些例子：

1. 时间序列是一组一个或多个过程（例如，股票价格、传感器值、能源流动）的测量。我们可以将时间序列表示为一个矩阵 $\mathrm{X} \sim (t, c)$，其中 t 是时间序列的长度，$\mathrm{X}_i \sim (c)$ 是在时间 t (的 c 测量，例如 c 个来自脑电图扫描的传感器，或 c 个股票价格)。每个时间瞬间相当于一个像素，每个测量相当于一个通道。

2. 音频文件（语音、音乐）也可以用一个矩阵 $\mathrm{X} \sim (t、c)$ 来描述，其中 t 是音频信号的长度，而 c 是录音的通道（单声道音频为1，立体声信号为2等）。

> **频率分析**
>
> 音频也可以通过频率分析（例如，在小窗口中提取MFCC系数）转换为类似图像的格式，在这种情况下，生成的*time-frequency*图像表示信号频率内容的变化 - 请参阅图F.8.1以获取示例。通过这种预处理，我们可以使用标准的卷积模型来处理它们。

3. 视频可以用一个四阶张量 $X \sim (t、h、w、c)$ 来描述，其中 t 是视频的 *frames* 数量，每个帧是一个形状为 $(h、w、c)$ 的图像。另一个例子是医学中的体部扫描，在这种情况下 t 是体积深度。

时间序列、音频信号和视频可以通过它们的采样率来描述，这表示有多少个样本

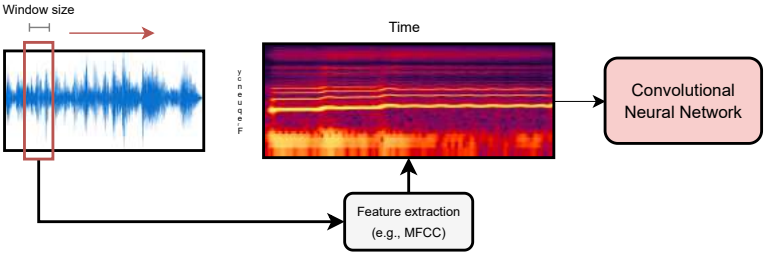


图 F.8.1：*Audio can be represented as either a 1D sequence (left), or a 2D image in a time-frequency domain (middle). In the second case, we can apply the same techniques described in the previous chapter.*

每单位时间获取，有时以每秒样本数或赫兹（Hz）表示。例如，经典脑电图单元以240 Hz的频率获取信号，意味着每秒240个样本。股票可以每分钟检查一次，相当于1/60 Hz。相比之下，音频以非常高的频率获取以确保保真度：例如，音乐可以以$44.1e^3$ Hz（或44.1 kHz）获取。视频的典型获取帧率约为每秒24帧，以确保对人类眼睛的平滑度。

图像分辨率、音频采样率和视频帧率都在确定信号获取精度方面发挥着类似的作用。对于图像，我们可以事先假设一个固定的分辨率（例如，1024 × 1024像素）。这是合理的，因为图像总是可以被重塑到给定的分辨率，同时保持足够的连贯性，除非分辨率非常小。相比之下，音频和视频的持续时间可以从输入到输入而变化（例如，一首30秒的歌曲与一首5分钟的歌曲），它们不能被重塑到共同的维度，[1]这意味着我们的

[1] In the sense of having the same duration *and* resolution.

数据集将由变长数据组成。此外，音频分辨率可以很容易地变得非常大：以44.1 kHz的采样率，3分钟音频将有$\approx 8M$个样本。

我们同样注意到，这些例子中的维度可以大致分为“空间维度”（例如，图像）或“时间维度”（例如，音频分辨率）。虽然图像可以在其空间轴上被认为是对称的（在许多情况下，沿宽度翻转的图像是另一个有效图像），时间则是 *asymmetric*：音频样本沿其时间轴反转通常无效，反转的时间序列表示一个从未来向过去演化的序列。除了在模型设计（因果性）中利用这一方面之外，我们还可以对信号的 *predicting* 未来值感兴趣：这被称为预测。

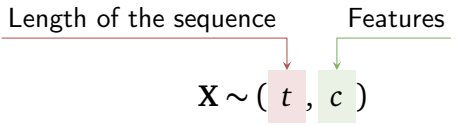
最后，考虑一个文本句子，例如“*the cat is on the table*”。有多种方法可以将这个句子分割成片段。例如，我们可以考虑它的单个音节：“[”*the*”，“*cat*”，“*i*”，“*s*”，“*on*”，“*the*”，“*ta*”，*ble*”]”。这是序列的另一个例子，只不过序列的每个元素现在是一个分类值（音节）而不是数值编码。因此，我们需要一种方法将这些值编码成模型可以处理的特征：将文本序列分割成组件称为分词，而将每个标记转换为向量称为嵌入标记。

在接下来的章节中，我们依次考虑所有这些方面（变长输入、因果关系、预测、分词和嵌入），以了解我们如何构建卷积模型来应对这些问题。我们介绍的一些技术，如掩码，非常通用，也适用于其他类型的模型，例如

作为变压器。其他技术，如扩张卷积，则是特定于卷积模型的。

8.1.2 1D和3D卷积层

让我们考虑如何为1D信号（例如时间序列、音频）定义卷积及其扩展到3D信号（例如视频）。请注意，维数仅指我们卷积的维度数（空间或时间），不包括通道维度。回想一下，在1D情况下，我们可以将输入表示为一个单矩阵：



我们现在复制第7章的推导过程。给定一个补丁大小 $s = 2k + 1$，我们定义 $P_k(i) \sim (s, c)$ 为 X 中距离 i (至多 k 的行的子集，忽略可以使用零填充的边界元素)。一个 1D 卷积层 H = Conv1D(X) 输出一个矩阵 H $\sim (t, c')$，其中 c' 是一个超参数，它定义了输出维度性，按行定义如下：

$$[\text{Conv1D}(X)]_i = \phi(\mathbf{W} \cdot \text{vect}(P_k(i)) + \mathbf{b}) \qquad \text{(E.8.1)}$$

具有可训练参数 W $\sim (c'、sc)$ 和 b $\sim (c')$。类似于二维情况，此层是局部的（对于局部性的适当修改定义）并且对序列的平移是等变的。

在二维情况下，我们还讨论了一种具有所有指标显式求和的替代记法：

$$H_{ijz} = \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',j',z,d} [X]_{i'+t(i),j'+t(j),d} \tag{E.8.2}$$

在 $t(i) = i + k - 1$ 如 (E.7.6) 中所示。回忆一下，我们使用 t 来分别索引两个张量 i' 和 j'：对于 W 从 1 到 $2k + 1$，对于 X 从 $i - k$ 到 $i + k$。通过移除一个求和索引，可以简单地得到 (E.8.1) 的等效变体：

$$H_{iz} = \sum_{i'=1}^{2k+1} \sum_{d=1}^{c} [W]_{i',z,d} [X]_{i'+t(i),d} \tag{E.8.3}$$

在参数 $W \sim (s、c'、c)$ 现已组织成一个秩为3的张量的情况下。相比之下，通过在第三维添加一个新的求和运算，并使用索引 p，得到3D变体：

$$H_{pijz} = \sum_{p'=1}^{2k+1} \sum_{i'=1}^{2k+1} \sum_{j'=1}^{2k+1}, \sum_{d=1}^{c} [W]_{p',i',j',z,d} [X]_{p'+t(p),i'+t(i),j'+t(j),d}$$

我们假设核大小在所有维度上相同，以简化问题。基于类似的原因，我们可以推导出卷积的3D向量化变体，以及1D和3D的最大池化变体。

8.2 1D和3D卷积模型

我们现在考虑一维情况下的卷积模型设计，重点关注如何处理可变长度输入以及如何处理文本序列。我们提出的几个想法对于所有可微模型来说相当通用。

8.2.1 处理变长输入

考虑两个音频文件（或两个时间序列，或两个文本），由它们相应的输入矩阵 $\mathrm{X}_1 \sim (t_1、c)$ 和 $\mathrm{X}_2 \sim (t_2、c)$ 描述。这两个输入共享相同数量的通道 c (，例如传感器的数量)，但它们的长度不同，t_1 和 t_2。记住我们在第 7.1 节中的讨论，卷积可以处理（原则上）这种变长输入。实际上，用 g 表示一个通用的 1D 卷积和最大池化操作的组合，对应于模型的特征提取部分。该模块的输出是两个矩阵：

$$\mathbf{H}_1 = g(\mathbf{X}_1), \mathbf{H}_2 = g(\mathbf{X}_2)$$

具有相同数量的列但不同数量的行（取决于对输入应用的最大池化操作或步幅卷积的数量）。全局平均池化后，对长度的依赖消失：

$$\mathbf{h}_1 = \sum_i \mathbf{H}_{1i}, \mathbf{h}_2 = \sum_i \mathbf{H}_{2i}$$

并且我们可以对向量 h_1 和 h_2 进行最终分类。然而，虽然这在模型层面上不是问题，但在实践中却是一个问题，因为无法从不同维度的矩阵中构建小批量，因此操作无法轻易向量化。这可以通过将结果小批量填充到序列长度跨度的最大维度来处理。例如，假设没有缺乏一般性的情况，$t_1 > t_2$，我们可以构建一个“填充”的小批量，如下所示：

```
# Sequences with variable length
# (3, 5, 2, respectively)
X1, X2, X3 = torch.randn(3, 8),
             torch.randn(5, 8),
             torch.randn(2, 8)

# Pad into a single mini-batch
X = torch.nn.utils.rnn.pad_sequence(
                    [X1, X2, X3],
                    batch_first=True)
print(X.shape)
# [Out]: torch.Size([3, 5, 8])
```

盒子 C.8.1: *A padded mini-batch from three sequences of variable length (with* $c = 8$*). When using a* DataLoader*, padding can be achieved by overwriting the default* collate_fn*, which describes how the loader concatenates the individual samples.*

$$X = \text{stack}\left(\mathbf{X}_1, \begin{bmatrix}\mathbf{X}_2 \\ \mathbf{0}\end{bmatrix}\right)$$

在堆栈操作新的前导维度时，生成的张量 X 的形状为 $(2, t_1, c)$。我们可以通过考虑与所有小批量元素相关的最大长度来推广到任何小批量。对于卷积，这与零填充没有太大区别，并且对填充输入的操作不会产生显著影响（例如，在音频中，零填充相当于在末尾添加静音）。参见盒 C.8.1 以获取构建填充小批量的示例。

另一种方法是构建一个描述 mini-batched 张量中有效和无效索引的掩码矩阵：

$$\mathbf{M} = \begin{bmatrix} \mathbf{1}_{t_1} \\ \mathbf{1}_{t_2} \ \mathbf{0}_{t_1 - t_2} \end{bmatrix}$$

在索引表示向量大小的地方。这些掩码矩阵有助于避免在输入张量上执行无效操作。

8.2.2 用于文本数据的CNNs

让我们现在考虑处理文本数据的问题。正如我们之前提到的，处理文本的第一步是分词，即将文本（一个字符串）划分为一系列已知符号（在这个上下文中也称为标记）。存在多种类型的标记化器：

1. 字符标记器：每个字符成为一个符号。

2. 单词分词器：每个（允许的）单词成为一个符号。

3. 子词分词器：介于字符分词器和词分词器之间，每个符号可能大于字符但小于单词。

这是在图F.8.2中示意出来的。在所有三种情况下，用户都必须定义一个允许的标记（词汇表），例如字符标记器中的所有ASCII字符。在实践中，可以选择字典的所需大小，然后查看文本中最频繁的标记来填充它，其他所有符号都进入一个特殊的“词汇表外”（OOV）标记。子词标记器为此有许多专门的算法，例如字节对编码（BPE）[SKF+99].[2]

[2]This is a short exposition focused on differentiable models, and we are ignoring many preprocessing operations that can be applied to text, such as removing stop words, punctuation, “stemming”, and so on. As the size of the models has grown, these operations have become

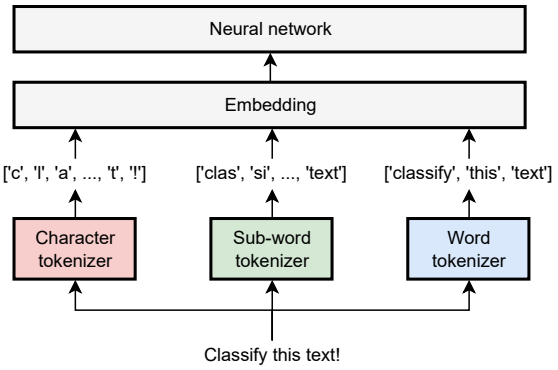


图 F.8.2: *Starting from a text, multiple types of tokenizers are possible. In all cases, symbols are then embedded as vectors and processed by a generic 1D model.*

因为大量文本可以具有很大的可变性，预训练的子词分词器现在是标准选择。作为一个具体例子，OpenAI 发布了其自己的分词器的开源版本，[3]，这是一个由大约 100k 个子词组成的子词模型（在撰写本文时）。例如，考虑使用此分词器对“*This is perplexing!*”进行编码，如图 F.8.3 所示。一些标记对应整个单词（例如，“*This*”），一些对应单词的一部分（例如，“*perplex*”），而另一些对应标点符号。该序列可以等价地表示为整数序列：

$$[2028, 374, 74252, 287, 0] \tag{E.8.4}$$

每个整数介于0和词汇表大小（在这种情况下，大约100k）之间，并且它唯一地标识了该词汇表中的标记。在实践中，没有任何东西阻止我们向序列中添加“特殊”标记，例如表示句子开头的标记

less common.

[3] https://github.com/openai/tiktoken

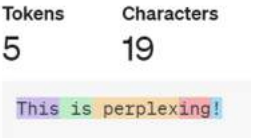


图 F.8.3：*Example of applying the tiktoken tokenizer to a sentence.*

(有时表示为 [BOS])，OOV 标记或其他任何内容。在下一节中，[BOS] 标记将具有特殊意义。

子词分词与非常大的词典有时可能难以理解：例如，常见的数字如52有自己的唯一标记，而像2512这样的数字可以拆分为“251”标记和“2”标记，因此可视化分词过程始终对调试模型行为很重要。[4]鉴于分词步骤的重要性，这是一个非常活跃的研究主题——例如，我们在这里提到字节级分词器[PPR+24]和可以端到端训练的分词器[HWG25]。

在标记化步骤之后，必须将标记嵌入到向量中，以便用作CNN的输入。在这里使用简单的独热编码策略效果不佳，因为词汇表很大，生成的向量将非常稀疏。相反，我们有两种替代策略：第一种是使用为我们执行嵌入的*pretrained*网络；我们将在介绍transformers时考虑这个选项。为了对它有一些直观的了解，我们在这里考虑第二种替代方案，*training*将嵌入与网络的其余部分一起考虑。

假设我们固定一个嵌入维度 e 作为超参数。由于字典的大小 n 也已固定，我们可以初始化一个嵌入矩阵 $\mathrm{E} \sim (n,\ e)$。我们现在

[4]For applications where processing numbers is important, specialized numerical tokenizers can be applied [GPE+23].

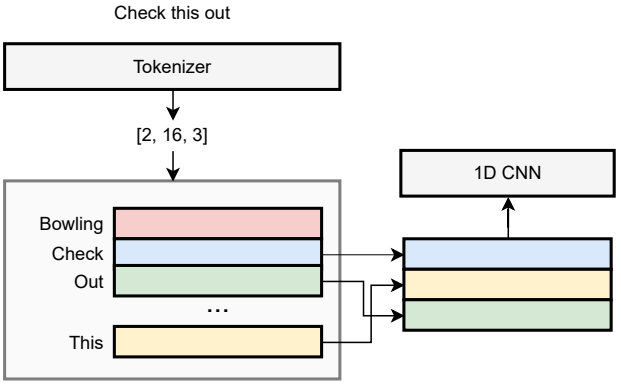


图 F.8.4：*A lookup table to convert a sequence of tokens' IDs to their corresponding embeddings: the input is a list, the output is a matrix. The embeddings (shown inside the box) can be trained together with all the other parameters via gradient descent. We assume the size of the vocabulary is* $n = 16$.

定义一个查找操作，将每个整数替换为 E 中的对应行。用 x 表示我们拥有的 ID 序列：

$$\text{LookUp}(x) = \mathbf{X} = \begin{bmatrix} \mathbf{E}_{x_1} \\ \mathbf{E}_{x_2} \\ \vdots \\ \mathbf{E}_{x_m} \end{bmatrix}$$

Row x_1 in the embedding matrix

结果输入矩阵X的形状将为(m, e)，其中m是序列的长度。现在我们可以应用通用的1D卷积模型，例如用于对文本序列进行分类：

$$\hat{y} = \text{CNN}(\mathbf{X})$$

此模型可以根据以下方式以标准方式进行训练

```
class TextCNN(nn.Module): def __init__(self, n, e):  super()._
_init__()  self.emb = nn.Embedding(n, e)  self.conv1 = nn.C
onv1d(e, 32, 5, padding='same')  self.conv2 = nn.Conv1d(32,
64, 5, padding='same')  self.head = nn.Linear(64, 10) def forw
ard(self, x):  # (*, m) x = self.emb(x)  # (*, m, e) x = x.trans
pose(1, 2)  # (*, e, m) x = relu(self.conv1(x))  # (*, 32, m) x
= max_pool1d(x, 2)  # (*, 32, m/2) x = relu(self.conv2(x))  #
(*, 64, m/2) x = x.mean(2)  return self.head(x) # (*, 10)
```

盒子C.8.2: *A 1D CNN with trainable embeddings. n is the size of the dictionary, e is the size of each embedding. We use two convolutional layers with* 32 *and* 64 *output channels. The shape of the output for each operation in the forward pass is shown as a comment.*

任务，除了梯度下降将同时进行模型参数和嵌入矩阵E的优化。这在图F.8.4中有所展示，模型定义的示例在框C.8.2中给出。

这个想法非常强大，尤其是在许多情况下我们发现，结果嵌入可以作为向量进行代数操作，例如，通过在欧几里得意义上查看最近的嵌入来找到“语义相似”的单词或句子。这个想法是许多需要检索或搜索文档的领域的可微模型使用的核心。

可微模型和嵌入

再次强调，嵌入的概念非常通用：任何将对象转换为具有代数特性的向量的过程都是嵌入。例如，经过全局池化后训练好的CNN的骨干网络的输出可以理解为输入图像的高级嵌入，并且可以通过与其他所有嵌入进行比较来检索“相似”的图像。

8.2.3 处理长序列

许多之前描述的序列可能非常长。在这种情况下，卷积层的局部性可能是一个缺点，因为我们需要线性增加的层数来处理越来越大感受野。我们将在下一章中看到，其他类别的模型（例如，转换器）可以设计来解决此问题。现在我们仍然处于卷积的领域，我们展示了一种有趣的方法，称为扩张（或法语中的*à trous*稀疏）卷积，在语音生成WaveNet模型[ODZ+16]中得到了普及。

我们引入一个称为膨胀率的额外超参数。膨胀率为1的卷积是标准卷积。对于膨胀率为2的情况，我们修改卷积操作，通过在序列中跳过一个元素来选择我们的补丁元素。同样，对于膨胀率为4的情况，我们在四个元素中跳过三个，等等。我们按指数级增加的膨胀率堆叠卷积层，如图F.8.5所示。参数数量不变，因为无论膨胀率如何，邻居的数量都保持不变。然而，很容易证明

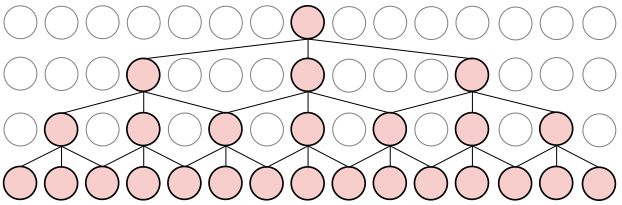


图 F.8.5：*Convolutional layers with increasing dilation rates. Elements selected for the convolution are in red, the others are greyed out. We show the receptive field for a single output element.*

结果感受野在此情况下在层数上增长 *exponentially fast*。

8.3 预测和因果模型

8.3.1 预测序列

一个重要的方面是，在处理序列时，我们可以构建一个模型来预测未来的元素，例如能源价格、湍流流动、呼叫中心占用等。预测标记也是大型语言模型和其他最近突破的基本构建块。在非常广泛的意义上，目前围绕神经网络的大部分兴奋都集中在这样一个问题上：一个模型可以从大量文本的下一个标记预测中推断出多少，以及这种设置可以在不同的模态（例如视频）和动态[WFD+23]中复制多少。在统计学和时间序列分析中，预测序列的下一个元素被称为预测。从现在起，为了与现代文献保持一致，我们将使用通用的术语标记来指代序列的每个元素，无论我们是在处理嵌入文本标记还是通用向量值输入。

预测是一个重要问题，原因在于我们只需访问一组序列即可训练预测模型，无需额外的目标标签：用现代术语来说，这也被称为自监督学习任务，因为目标可以从输入中自动提取。

平稳性与预测

就像文本处理一样，预测现实世界的时间序列有许多相关的问题（例如，时间序列的可能非平稳性、趋势和季节性）在这里我们没有考虑。[a]在实践中，音频、文本以及许多其他感兴趣的序列可以被认为是平稳的，不需要特殊的预处理。对于文本来说，对于非常大的预测数据集和相应的大型模型，预处理的影响往往会减弱 [AST+24]。

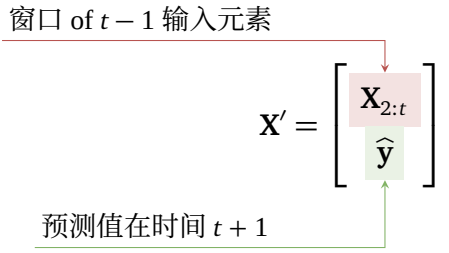
[a] https://filippomb.github.io/python-time-series-handbook/

为此，假设我们固定一个用户定义的长度 t，并从数据集中提取所有可能的长度为 t 的子序列（例如，使用 $t = 12$，所有连续的12个元素窗口，或由12个标记组成的所有句子等）。在LLMs的上下文中，输入序列的大小被称为模型的上下文。我们将每个子序列与一个目标值关联，该值是序列中的下一个元素。因此，我们构建了一组对 (X, y)，$\mathrm{X} \sim (t, c)$，$y \sim (c)$，并且我们的预测模型在这个数据集上以监督方式进行训练：

$$f(\mathbf{X}) \approx \mathbf{y}$$

请注意，标准1D卷积模型可以用作预测模型，使用均方误差进行训练。

错误（对于连续时间序列）或交叉熵（对于分类序列，如文本）。虽然模型被训练来预测单步预测，但我们可以通过所谓的自回归方法轻松地生成我们想要的任何步骤数，这意味着模型正在预测（{v*}）其自身的输出。假设我们预测一个单步，$\hat{y} = f(X)$，并通过将我们的预测值添加到输入中（移除第一个元素以避免超过 t 个元素）创建一个“平移”输入：



(E.8.5)

预测离散序列

对于连续时间序列，这很简单。对于具有离散值的时间序列，f将返回一个可能的值（即可能的标记）的概率向量，我们可以通过取其arg max获得$\hat{y}$，即与最高概率相关的标记。或者，我们可以按预测概率成比例地采样一个标记：参见第8.4.1节。

我们现在可以运行 $f(X')$ 来生成序列中的下一个输入值，以此类推，通过始终以先进先出（FIFO）的方式更新我们的缓冲输入。这种方法非常强大，但它要求我们事先固定输入序列的长度，这限制了其适用性。为了克服这一限制，我们只需要对我们的模型进行微小的修改。

8.3.2 因果模型

假设我们只有4个元素组成的短序列，收集到一个矩阵X $\sim (4, c)$，但我们使用$t = 6$的更长序列训练了一个预测模型。为了在较短序列上运行模型，我们可以在序列开头用两个零向量0进行零填充，但除非我们屏蔽其操作，否则这些将被模型解释为时间序列的实际值。幸运的是，存在一种更简单、更优雅的方法，即因果模型。

定义 D.8.1（因果层）

A layer $\mathrm{H} = f(\mathrm{X})$ *is* ***causal*** *if* $\mathrm{H}_i = f(\mathrm{X}_{:i})$*, i.e., the output value corresponding to the i-th element of the sequence depends only on elements "from its past".*

一个仅由因果层组成的模型当然本身也是因果的。例如，核大小为1的卷积层是因果的，因为每个元素只考虑自身进行处理。然而，核大小为3的卷积层不是因果的，因为它除了考虑自身外，还要考虑左侧和右侧的一个元素。我们可以通过部分零掩码非因果连接对应的权重，将任何卷积转换为因果变体：

$$\mathbf{h}_i = \phi\left(\left[\ \mathbf{W} \odot \mathbf{M}\ \right] \mathrm{vect}(P_k(i)) + \mathbf{b}\right)$$

Masked weight matrix

在 $M_{ij} =$ 为 0 如果权重对应于输入中的一个元素，使得 $j > i$，否则为 1。因果 1D 卷积可以与膨胀核结合使用

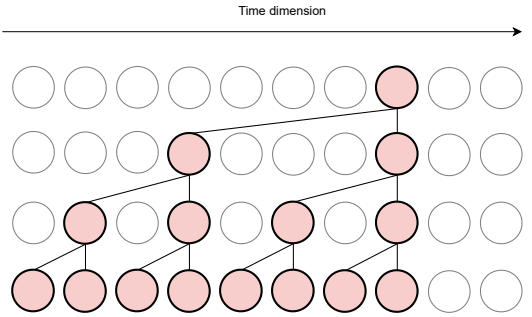


图 F.8.6：*Overview of a 1D causal convolutional layer with (original) kernel size of* 3 *and exponentially increasing dilation rates. Zeroed out connections are removed, and we show the receptive field for a single output element.*

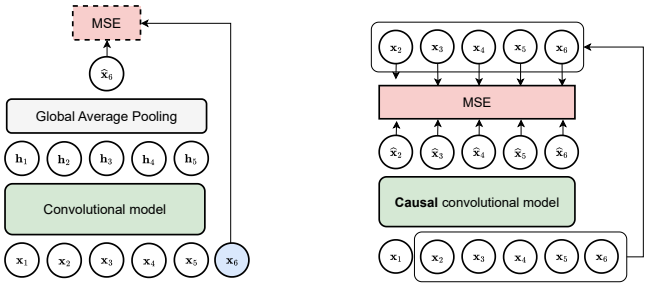
获取音频自回归模型，例如WaveNet模型 [ODZ+16] - 有关示例，请参阅图F.8.6。

掩码在单通道的情况下更容易理解，在这种情况下，M 只是一个下三角二进制矩阵。掩码操作有效地将参数数量从 $(\hat{}2k+1)cc'$ 减少到 $(k+1)cc'$。

通过堆叠多个因果卷积层，我们可以获得一个因果1D模型变体。假设我们将它应用于我们的输入序列，使用一个没有最大池化操作的模型。在这种情况下，输出序列的长度与输入序列相同：

$$\widehat{\mathbf{Y}} = f_{\text{causal}}(\mathbf{X})$$

除了，输出中的任何元素只依赖于相同位置或其之前的输入元素。因此，我们可以通过预测一个值
for each element of the input sequence 来定义一个更复杂的预测模型。



(a) *Non-causal model* **(b)** *Causal model*

图 F.8.7：*Comparison between (a) a non-causal model for forecasting (predicting only a single element for the entire input sequence) and (b) a causal model trained to predict one output element for each input element in the sequence.*

实际上，现在考虑一个定义为的矩阵输出：

$$\mathbf{Y} = \begin{bmatrix} \mathbf{X}_{2:t} \\ \mathbf{y} \end{bmatrix}$$

这与（E.8.5）中的平移输入相似，但我们在序列的最后一个元素添加了真实值。我们可以通过最小化所有元素上的损失来训练此模型，例如，均方误差：

$$l(\widehat{\mathbf{Y}}, \mathbf{Y}) = \|\widehat{\mathbf{Y}} - \mathbf{Y}\|^2 = \sum_{i=1}^{t} \|\widehat{\mathbf{Y}}_i - \mathbf{Y}_i\|^2 \tag{E.8.6}$$

Loss when predicting $\mathbf{X}_{i+1}$

我们同时根据第一个元素预测第二个元素，根据前两个元素预测第三个元素，等等。对于单个输入窗口，我们有 t 个独立的损失项，极大地增强了梯度传播。两种方法之间的比较如图 F.8.7 所示：在

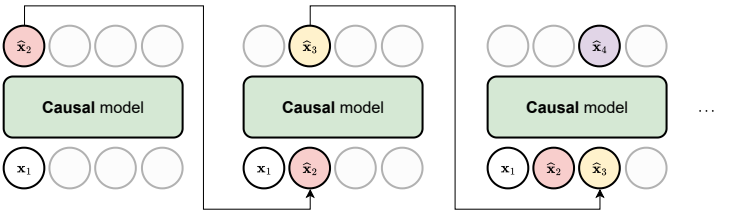


图 F.8.8：*Inference with a causal CNN, generating a sequence step-by-step in an autoregressive way. Unused input tokens are greyed out. Generated tokens are shown with different colors to distinguish them.*

图F.8.7a展示了训练用于预测序列中下一个元素的因果卷积模型，而图F.8.7b展示了根据（E.8.6）训练的因果模型。

更重要的是，我们现在可以使用模型以自回归方式处理任意长度序列，最长可达 t 的最大长度。这可以通过一个例子轻松看出。假设我们有 $t = 4$，并且我们观察到了两个值 x_1 和 x_2。我们通过将序列进行零填充来调用模型一次，以生成第三个标记：

$$\begin{bmatrix} - \\ \widehat{\mathbf{x}}_3 \\ - \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}\right)$$

我们忽略除了第二个输出值之外的所有输出值（实际上，第三个和第四个输出值由于零填充而无效）。我们将 $\widehat{}\mathrm{x}_3$ 添加到序列中，并继续自回归地调用模型（我们用颜色显示预测值）：

$$\begin{bmatrix} - \\ - \\ \widehat{\mathbf{x}}_4 \\ - \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \mathbf{0} \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_5 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \end{bmatrix}\right), \begin{bmatrix} - \\ - \\ - \\ \widehat{\mathbf{x}}_6 \end{bmatrix} = f\left(\begin{bmatrix} \mathbf{x}_2 \\ \widehat{\mathbf{x}}_3 \\ \widehat{\mathbf{x}}_4 \\ \widehat{\mathbf{x}}_5 \end{bmatrix}\right) \cdots$$

在最后一步中，我们移除了一个原始输入以保持对输入大小的约束。这也在图F.8.8中显示。请注意，该模型仅在真实值上训练，而不是在其自己的预测上：这被称为教师强制。教师强制的变体是在训练过程中，随着模型变得更加准确，逐步用模型预测的值替换迷你批次中的某些值。

因果自回归模型在文本序列的情况下特别有趣（因为我们只有一个通道，即标记的索引），因为我们可以从代表序列开始的单个 [BOS] 标记开始，从头生成文本句子，或者 *condition* 根据用户添加到 [BOS] 标记的特定提示进行生成。类似的推理可以应用于音频模型以生成语音或音乐 [ODZ+16]。

8.4 生成模型

8.4.1 一个概率公式

自回归模型是生成模型的一个简单例子。[5] 我们将详细讨论其他

[5] Remember from Chapter 3 that we assume our supervised pairs (x, y) come from some unknown probability distribution $p(x, y)$. By the product rule of probability we can decompose it equivalently as $p(y \mid x)p(x)$, or $p(x \mid y)p(y)$. Any model which approximates $p(x)$

下一卷中介绍生成模型的类型。目前，我们提供一些针对自回归算法的特定见解。我们考虑具有单个通道和离散值（如文本）的序列。基于文本标记的自回归模型是LLMs的基础，它们可以用作多模态架构的基础（第11章）。

生成模型在概率的背景下更为自然，因此我们首先用概率形式重新表述之前的讨论。用 $\mathscr{X}$ 表示所有可能的序列空间（例如，所有可能的文本标记组合）。一般来说，这些序列中的许多将是无效的，例如英语中的序列 [“*tt*”，“*tt*”]。然而，即使在非常大的文本语料库中，非常不常见的序列也可能至少出现一两次（想象一个角色大喊“*Scotttt!*”）。

我们可以通过考虑所有可能序列 $x \in \mathscr{X}$ 上的概率分布 $p(x)$ 来推广这一点。在文本的上下文中，这也被称为语言模型。生成建模是学习从该分布中高效采样的任务：[6]

$$x \sim p(x)$$

要看到这与我们之前的讨论如何相关，请注意，根据概率乘法法则，我们总是可以

or $p(x \mid y)$ is called **generative**, because you can use it to sample new input points. By contrast, a model that only approximates $p(y \mid x)$, like we did in the previous chapters, is called **discriminative**.

[6] In this section $\sim$ is used to denote sampling from a probability distribution instead of the shape of a tensor.

重写 $p(x)$ 为：

$$p(x) = \prod_i p(x_i \mid x_{:i}) \tag{E.8.7}$$

在何处我们将每个值 x_i 条件化到所有前面的值。如果我们假设我们的模型输入长度足够大，可以容纳所有可能的序列，我们可以使用因果预测模型来参数化（E.8.7）中的概率分布：

$$p(x_i \mid x_{:i}) = \text{Categorical}(x_i \mid f(x_{:i}))$$

在所有时间步长中使用单个共享模型。在此模型上的最大似然等价于最小化预测概率上的交叉熵损失，如第4.2.2节所述。

8.4.2 自回归模型中的采样

一般来说，从概率分布中进行采样是非平凡的。然而，对于自回归模型，我们可以利用（E.8.7）中的乘积分解来设计一种简单的迭代策略：

1. 示例 $x_1 \sim p(x_1)$。这相当于在空集 $p(x_1 \mid \{\})$ 上进行条件化。在实践中，我们总是对初始固定标记进行条件化，例如 [BOS] 标记，这样我们的输入永远不会为空。

2. 通过再次运行我们在步骤（1）中采样的值来采样示例 $x_2 \sim p(x_2 \mid x_1)$，如图 F.8.8 所示。

3. 样本 $x_3 \sim p(x_3 \mid x_1, x_2)$。

4. 继续直到达到所需的序列长度

或者直到我们到达一个句子结束标记。

我们之前隐式地这样做，总是采样最高概率的元素：

$$x_i = \arg\max_i f(x_{:i})$$

然而，我们也可以通过根据f预测的概率采样一个值来推广这一点。记住（第4.2.1节），softmax可以通过考虑一个额外的温度参数来推广。通过在推理过程中改变这个参数，我们可以在始终取argmax值（非常低的温度）和在标记上具有几乎均匀分布（非常高的温度）之间平滑地变化。

在概率建模的背景下，从这个模型类中这样采样被称为祖先采样，而在语言建模的背景下，我们有时使用贪婪解码这个术语。使用“贪婪”这个术语以及这次简短的讨论足以突出这种方法的潜在缺点：虽然$p(x)$的乘积分解是精确的，但贪婪解码并不能保证提供与$p(x)$高值相对应的样本。

要看到这一点，请注意，f提供了对单个标记的概率估计，但序列的概率是由许多此类项的乘积给出的。因此，在序列的开始处采样具有高（局部）概率的标记可能并不对应于具有大（全局）概率的句子。如果你想象第一个标记的选择让解码阶段“卡”在低概率路径上，这很容易理解。

一种常见的缓解方法是束搜索（或束解码）。在束搜索中，第一步我们采样 k 个不同的元素（称为束，其中 k 是用户定义的参数）。第二步，对于我们的 k 个束中的每一个，我们采样 k 个可能的延续。在这些 k^2 对中，我们只保留按其乘积概率 $p(x_1)p(x_2 \mid x_1)$ (或，等价地，其对数概率) 排名前 k 的值。我们以这种方式迭代进行，直到序列结束。

从这一角度出发，从我们的自回归模型中采样最可能的序列是一个组合搜索问题（想想一棵树，其中对于每个标记，我们扩展到所有可能的下一个标记，依此类推）。从计算机编程的角度来看，波束搜索是这种树上的广度优先搜索的一个例子。在某种程度上，波束搜索是在简单的训练过程和更昂贵的推理阶段之间进行权衡——为此存在许多其他技术，包括引导解码以满足外部奖励函数 [WBF+24] 的可能性。

8.4.3 条件建模

如我们之前提到的，通常我们可能不太感兴趣从头开始生成序列，而是在生成已知序列的延续，例如用户的提问或交互。这可以通过考虑形式为 $p(x \mid c)$ 的 *conditional* 概率分布来形式化，其中 c 是条件论据，例如用户的提示。我们之前的讨论几乎可以直接扩展到这种情况。例如，乘积分解现在可以写成：

$$p(x \mid c) = \prod_i p(x_i \mid x_{:i}, c)$$

在给定先前输入 *and* 的用户上下文的情况下。采样和解码以类似方式扩展。

为了执行条件生成，我们将 $p(x_i \mid x_{:i}, c)$ 用神经网络 $f(x_{:i}, c)$ 参数化，使得：

$$p(x_i \mid x_{:i}, c) \approx \text{Categorical}(x_i \mid f(x_{:i}, c))$$

因此，与无条件情况的主要区别在于我们需要一个具有两个输入参数的函数 $f(x, c)$，并且该函数在第一个参数中满足因果性。当与自回归模型一起工作时，如果 x 和 c 都是文本，我们可以通过将 c 视为输入序列的一部分，并使用单个连接输入 $x' = [c \| x]$ 来轻松完成。例如，对于用户的提示“*The capital of France*”，为了简单起见，使用一个单词分词器，我们可能会有：[7]

$$f_{\text{causal}}([\text{The}, \text{capital}, \text{of}, \text{France}]) = \text{is}$$
$$f_{\text{causal}}([\text{The}, \text{capital}, \text{of}, \text{France}, \text{is}]) = \text{Paris}$$

因此，我们可以使用单个模型同时处理无条件建模和条件建模.[8] 在下一卷中，我们将看到其他条件建模的例子。

[7] We ignore the presence of an end-of-sequence token (EOS) to stop the autoregressive generation.

[8] We will see in Chapter 11 that almost any type of data can be converted into a sequence of tokens. Suppose we are generating a text sequence conditioned on an image prompt (e.g., **image captioning**). If both text and images are converted to tokens having the same embedding size, we can apply an autoregressive model by concatenating the tokens from the two input types (also called **modalities** in this context), where we view the image tokens as the conditioning set c.

生成模型中需要更复杂的策略。我们还将在此主题上扩展，当我们在第11章讨论仅解码器Transformer模型时。

从理论到实践

与文本数据相比，图像分类更复杂，因为涉及许多与分词、数据格式化、奇怪字符和可变长度序列相关的细微差别。PyTorch拥有自己的文本库torchtext，在撰写本文时，它相对较少

文档化的比主库更少，并依赖于另一个beta库（torchdata）来处理数据管道。因此，我们在这里忽略它，但邀请您自己检查它。

Hugging Face 数据集可能是这种情况中最通用的工具，因为它提供了一系列庞大的数据集和预训练的标记器，可以立即导出到 PyTorch.[9] 在进行练习之前，先熟悉一下这个库。

1. 选择一个文本分类数据集，例如经典的IMDB数据集.[10]

2. 将其分词以获得形式为 (x, y) 的数据集，其中 x 是类似于（E.8.4）中的整数列表，y 是文本标签。

3. 构建并训练一个类似于盒C.8.2的1D CNN模型。稍微实验一下模型设计，看看它对最终准确率的影响。

PyTorch没有快速将1D卷积设置为因果卷积的方法，因此我们将推迟我们的自回归

[9] See this tutorial for a guide: `https://huggingface.co/docs/datasets/use_dataset`.

[10] `https://huggingface.co/datasets/stanfordnlp/imdb`

实验，当我们引入变压器时.[11] 训练自己的分词器是一个非常棒的教学习练，尽管它远远超出了本书的范围。作为一个介绍，您可以查看这个极简的BPE实现：

https://github.com/karpathy/minbpe。

[11] If you want to try, you can emulate a causal convolution with proper padding; see Lecture 10.2 here: https://fleuret.org/dlc/. The entire course is really good if you are looking for streamed lectures.

9 | 模型扩展

关于本章

我们现在转向设计具有数十（或数百）层的可微分模型的任务。正如我们所见，卷积模型的感觉野随着层数的增加而线性增长，这促使我们采用具有如此深度架构。这可以通过使用从数据增强到隐藏状态归一化的众多方法来正确稳定训练来实现。

9.1 ImageNet挑战

让我们再次考虑图像分类的任务，这在实践中和历史上都对神经网络具有强烈的兴趣。事实上，2012-2018年期间对这些模型的研究兴趣在很大程度上可以与ImageNet大规模视觉识别挑战[1]（（后文简称ImageNet）联系起来。ImageNet是一个年度的

[1] https://image-net.org/challenges/LSVRC/

挑战从2010年到2017年进行，以评估图像分类的最新模型。该挑战在ImageNet数据集的一个子集上运行，该子集包含大约100万张标签为1k个类别的图像。

有指导意义的是看看挑战的早期版本。在2010[2]年和2011[3]年，获胜者是由专门图像描述符和核的组合构建的线性核方法，2010年的前5%错误率为28%，2011年为26%。尽管有许多有希望的结果[4]，但使用梯度下降训练的卷积模型在计算机视觉中仍然是一个小众话题。然后，在2012年，获胜模型（AlexNet，[KSH12]）实现了前5%错误率为15.3%，比所有（非神经）竞争对手低10%。

这随后在该领域引发了一场真正的“哥白尼革命”（向哥白尼道歉），因为仅仅几年时间，几乎所有提交的论文都转向了卷积模型，整体准确率以史无前例的速度增长，超过95%（导致2017年挑战结束），如图F.9.1所示。在5年的时间跨度内，使用梯度下降训练的卷积模型成为了计算机视觉的领先范式，包括我们在此处未提及的其他子领域，从目标检测到语义分割和深度估计。

AlexNet是一个相对简单的模型，由5个卷积层和3个全连接层组成，总参数量约为60M，而图F.9.1中表现最好的模型需要多达数百层。这是缩放定律的基本示例（第1章）：添加

[2] https://image-net.org/challenges/LSVRC/2010/

[3] https://image-net.org/challenges/LSVRC/2011/

[4] https://people.idsia.ch/~juergen/computer-vision-contests-won-by-gpu-cnns.html

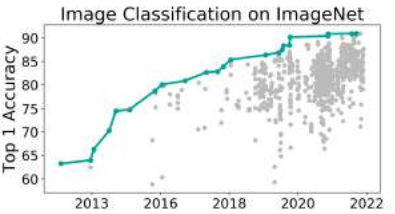


图 F.9.1：*Top- 1 accuracy on the ImageNet dataset. Reproduced from Papers With Code.*

层和计算能力与模型精度成正比，直到由数据集给出的饱和点。然而，将卷积模型扩展到几层以上是非平凡的，因为它会遇到从慢优化到梯度问题和数值不稳定性的一系列问题。因此，在2012-2017年间开发了一系列技术来稳定超大型模型的训练。

在这一章中，我们概述了一些这些技术。我们关注至今仍具有根本性的想法和方法，即使对于其他架构（例如，变换器）。我们首先介绍三种在机器学习中广为人知的改进训练的技术：权重正则化、数据增强和早期停止。然后，我们描述了在2012-2017年间流行起来的三种最具影响力的技术：dropout、批量归一化和残差连接，大致按照引入的时间顺序。对于每种方法，我们描述了基本算法以及一些在实践中表现良好的变体（例如，层归一化）。

9.2 数据和训练策略

9.2.1 权重正则化

一种可能的改进训练的方法是惩罚那些看似不合理的解决方案，例如拥有一个或两个非常大的权重。用w表示我们模型的所有参数的向量，用L(w、$\mathscr{S}_n$)表示我们数据集上的损失函数（例如平均交叉熵）。我们可以通过定义一个所谓的正则化项R(w)来形式化前面的想法，该正则化项根据我们的偏好对解决方案进行评分，并通过将正则化项添加到原始损失函数中来惩罚损失：

$$L_{\text{reg}} = L(\mathbf{w}, \mathscr{S}_n) + \lambda R(\mathbf{w})$$

在本文中，我们假设R(w)的值越高，对应的解越差，而$\lambda \geq 0$是一个权重两个项的标量。对于$\lambda = 0$，正则化项没有效果，而对于$\lambda \to \infty$，我们仅根据先验知识简单地选择最佳函数。

这也可以被解释为基于权重 p(w) 的先验分布组合和我们对数据的标准似然函数（第3.3节）进行的最大先验（而不是最大似然）推理：

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} \{\log p(\mathscr{S}_n \mid \mathbf{w}) + \log p(\mathbf{w})\} \qquad \text{(E.9.1)}$$

在具有正则化项对应于非均匀先验分布 p(w) 的情况下。我们已经在第 4.1.5 节中看到了正则化的一个例子，即权重的 ℓ_2 范数：

$$R(\mathbf{w}) = \|\mathbf{w}\|^2 = \sum_i w_i^2$$

对于相同的未正则化损失，惩罚 ℓ_2 范数将有利于具有较低权重幅度的解，对应于输入微小偏差下输出“不那么突然”的变化.[5] 现在考虑正则化项对梯度项的影响：

$$\nabla L_{\text{reg}} = \nabla L(\mathbf{w}, \mathscr{S}_n) + 2\lambda\mathbf{w}$$

以这种形式写出，有时被称为权重衰减，因为如果没有第一个项，其净效果是通过一个小的比例因子 λ (使权重指数快速衰减到0，在迭代次数上如果 $\nabla L(\mathrm{w}, \mathscr{S}_n) = 0$)。对于（S）GD，$\ell_2$ 正则化和权重衰减是一致的。然而，对于其他类型的优化算法（例如，基于动量的 SGD、Adam），通常会在梯度上应用后处理。用 $g(\nabla L(\mathrm{w}, \mathscr{S}_n))$ 表示（未正则化的）损失的经过后处理的梯度，我们可以写出一个广义权重衰减公式（忽略常数

t term 2) as:

Unregularized gradient

$$\mathbf{w}_t = \mathbf{w}_{t-1} - g(\nabla L(\mathbf{w}_{t-1}, \mathscr{S}_n)) - \lambda\mathbf{w}_{t-1}$$

Weight decay term

这与纯 ℓ_2 正则化不同，在这种情况下，正则化项的梯度将位于 $g($ 内
•)这对于像Adam这样的算法尤为重要，

[5] With respect to (E.9.1), ℓ_2 regularization is equivalent to choosing a Gaussian prior on the weights with diagonal $\sigma^2\mathbf{I}$ covariance.

对于其中，权重衰减公式（称为AdamW [LH19]）可以更好地工作。

我们还可以考虑其他类型的正则化项。例如，ℓ_1范数：

$$R(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |x_i|$$

可以优先选择具有高零值百分比的 *sparse* 解决方案（这相当于对权重施加拉普拉斯先验）。这也可以推广到将稀疏变体分组以对神经元 [SCHU17].[6] 强制结构稀疏。然而，与其他机器学习模型相比，稀疏 ℓ_1 惩罚较少见，因为它与优化问题的强非凸性和梯度下降 [ZW23] 的使用相互作用不佳。但是，可以通过增加更大的内存占用成本来重新参数化优化问题以减轻此问题。特别是，[ZW23] 表明我们可以用两个形状等效的向量 a 和 b 替换 w，并且：

$$\mathbf{w} = \mathbf{a} \odot \mathbf{b}\,,\ \|\mathbf{w}\|_1 \approx \|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 \qquad \text{(E.9.2)}$$

在 $\approx$ 中表示，在非常一般的条件下，可以证明这两个问题几乎是等价的 [ZW23]。

我们可以通过考虑一个凸损失函数 {v*} 来获得一些几何洞察，了解正则化为什么（以及如何）起作用，例如最小二乘法 {v*}，在这种情况下，正则化的

[6] Training sparse models is a huge topic with many connections also to efficient hardware execution. See [BJMO12] for a review on sparse penalties in the context of convex models, and [HABN+21] for an overview of sparsity and pruning in general differentiable models.

问题可以重写为显式约束形式：{v*}

$$\begin{array}{ll} \arg\min & L(\mathbf{w}, \mathscr{S}_n) \\ \text{subject to} & R(\mathbf{w}) \leq \mu \end{array} \tag{E.9.3}$$

在 μ 与 λ 成比例依赖的情况下，通过将 (E.9.3) 用拉格朗日乘子重新表述得到无约束公式。在这种情况下，ℓ_2 正则化对应于将解约束在以原点为中心的圆内，而 ℓ_1 正则化对应于解位于以原点为中心的正多边形内部（或顶点上），稀疏解位于与轴相交的顶点上。

9.2.2 早期停止

从优化的角度来看，最小化函数 $L(\mathrm{w})$ 的任务是尽可能快地找到一个驻点，即一个满足 $\nabla L(\mathrm{w}_t) \approx 0$ 的点 w_t：

$$\|L(\mathbf{w}_t) - L(\mathbf{w}_{t-1})\|^2 \leq \varepsilon$$

对于某些公差 $\varepsilon > 0$。然而，这并不一定对应于我们在优化模型时想要的结果。特别是，在低数据状态下训练时间过长可能导致过拟合，而且一般来说，任何提高泛化能力的做法都是好的，无论其对 $L($ 的净效应如何。

•) 或者下降方向（例如，权重衰减）。

早期停止是纯优化与学习之间差异的一个简单例子。假设我们有一个小型的监督数据集，它独立于训练和测试数据集，我们称之为验证集

数据集。在每个epoch结束时，我们在验证数据集上跟踪一个感兴趣的指标，例如准确率或F1分数。我们用a_t表示t个epoch时的分数。早期停止的想法是检查这个指标是否持续改进：如果不改进，我们可能正在进入过拟合阶段，我们应该停止训练。因为准确率可能会因为随机波动而略有波动，所以我们通过考虑一个k个epoch的窗口（耐心期）来稳健地处理这个问题。

$$\text{If } a_t \leq a_i, \quad \forall i = t-1, t-2, \ldots, t-k \quad \rightarrow \text{Stop training}$$

Wait for k epochs

对于耐心超参数 k 的高值，算法将等待更长时间，但我们将对可能的振荡更加稳健。如果我们有存储模型权重（检查点）的机制，我们还可以将权重回滚到最后一个显示改进的epoch，对应于epoch编号 $t-k$。

早期停止可以被视为一种简单的模型选择形式，其中我们根据给定的指标选择最优的epoch数量。与模型优化不同，我们在这里可以为任何感兴趣的指标进行优化，例如F1分数，即使它不可微。

有趣的是，对于大型过参数化模型，早期停止并不总是有益的，因为训练轮数与验证误差之间的关系可能是非单调的，存在多个上升和下降阶段（称为多次下降 [RM22]）以及长期停滞后的损失突然下降 [PBE+22]。因此，早期停止主要在优化小型数据集时是有用的。

9.2.3 数据增强

一般来说，提高模型性能的最有效方法是增加可用数据量。然而，标注数据可能成本高昂且耗时，而人工生成数据（例如，借助大型语言模型）需要定制化的管道才能有效工作 [PRCB24]。

在许多情况下，通过根据某些预先指定的（语义保持的）变换将它们进行转换，可以部分缓解这个问题通过 *virtually* 增加可用数据量。作为一个简单的例子，考虑一个向量输入 x 和通过添加高斯噪声引起的变换：

$$\mathbf{x}' = \mathbf{x} + \varepsilon,\ \varepsilon \sim \mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I})$$

这创建了一个包含在以x为中心的小球中的几乎无限量的数据。此外，这些数据不能存储在磁盘上，并且可以通过在每次选择新的迷你批次时在运行时应用转换来模拟这个过程。事实上，已知以这种方式训练可以使模型更加鲁棒，并且它与ℓ_2正则化[Bis95]相关。然而，向量数据是无结构的，添加过高方差噪声可能会生成无效的点。

对于图像，我们可以通过注意到通常存在大量可以改变图像同时保持其语义的变换来做得更好：缩放、旋转、亮度修改、对比度变化等。用 $T(x\,;c)$ 表示这样的一个变换（例如，旋转），该变换由某些参数 c（参数化，例如旋转角度）。大多数变换将基本图像作为特殊情况（在这种情况下，例如，旋转）

角度 $c = 0$）。数据增强是在训练过程中根据这些变换之一或多个变换来转换图像的过程：

$$x' = T(x; c),\ c \sim p(c) \tag{E.9.4}$$

$p(c)$表示所有有效参数的分布（例如，$-20°$和$+20°$之间的旋转角度）。在训练过程中，数据集的每个元素在每个epoch中采样一次，并且每次可以应用不同的变换（E.9.4），从而创建一个（实际上）无限的独特数据点流。

数据增强在图像（或类似数据，如音频和视频）中非常常见，但它需要做出一些设计选择：包括哪些变换，考虑哪些参数，以及如何组合这些变换。一种简单的策略称为RandAugment [CZSL20]，它考虑了广泛的变换，并为每个小批量样本选择少量（例如，2或3）变换，以相同的幅度依次应用。然而，用户必须验证变换的有效性（例如，如果识别文本，水平翻转可能会使生成的图像无效）。从实际角度来看，数据增强可以包含在数据加载组件中（参见框C.9.1），或者包含在模型中。

数据增强管道和方法可能比简单的直观变换更复杂。即使是更复杂类型，直觉仍然是，只要模型能够在复杂场景中解决任务（例如，在所有亮度条件下识别物体），它就应该在现实、温和的场景中表现得更好。此外，数据增强可以通过避免多次重复相同的输入来防止过拟合。

```
# Image tensor (b, c, h, w)
img = torch.randint(0, 256,
            size=(32, 3, 256, 256))

# Data augmentation pipeline
from torchvision.transforms import v2
transforms = v2.Compose([
    v2.RandomHorizontalFlip(p=0.5),
    v2.RandomRotation(10),
])

# Applying the data augmentation pipeline:
# each function call returns a different
# mini-batch starting from the same
# input tensor.
img = transforms(img)
```

盒子 C.9.1: *Data augmentation pipeline with two transformations applied in sequence, taken from the* torchvision *package. In PyTorch, augmentations can be passed to the data loaders or used independently. In other frameworks, such as TensorFlow and Keras, data augmentation can also be included natively as layers inside the model.*

作为一个更复杂方法的例子，我们描述了用于向量的mixup [ZCDLP17]及其用于图像的扩展cutmix [YHO+19]。对于前者，假设我们采样两个示例，(x_1, y_1) 和 (x_2, y_2)。mixup的想法是创建一个新的、虚拟的示例，它由它们的凸组合给出：

$$\mathbf{x} = \lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \tag{E.9.5}$$

$$y = \lambda y_1 + (1-\lambda) y_2 \tag{E.9.6}$$

在区间 $[0, 1]$ 中随机选择 λ。此过程应推动模型在两个示例之间具有简单的（线性）输出，避免突然

输出变化。从几何角度来看，对于两个接近的点，我们可以将（E.9.6）视为在数据流形上缓慢移动，通过跟随连接两个点的线，当λ从0到1变化时。

Mixup可能不适用于图像，因为逐像素线性插值两个图像会产生模糊的图像。在cutmix中，我们而是在第一张图像上采样一个固定形状的小块（例如，32 × 32）。用M表示与图像相同形状的二进制掩码，其中块内的像素为1，块外的像素为0。在cutmix中，我们通过“缝合”第一张图像的一部分到第二张图像的顶部来组合两个图像x_1和x_2:

$$x = \mathbf{M} \odot x_1 + (1 - \mathbf{M}) \odot x_2$$

当标签仍然以前的方式以随机系数 λ 进行线性插值。参见图 F.9.2 以了解使用旋转和 cutmix 进行数据增强的示例。

9.3 Dropout 和归一化

我们之前章节中描述的策略非常通用，从意义上讲，它们意味着对优化算法或数据集本身的修改，并且可以应用于广泛的算法。

相反，我们现在关注在2012年至2016年期间流行起来的三个想法，主要是在ImageNet挑战的背景下。这三个想法都针对可微模型，因为它们可以作为模型中的附加层或连接来实现，从而简化了非常深层的模型的训练。我们按大致的时间顺序列出这些方法。正如我们将看到的

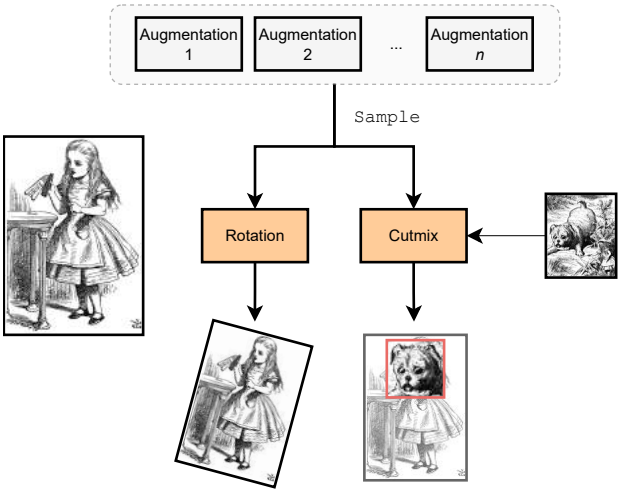


图 F.9.2: *High-level overview of data augmentation. For every mini-batch, a set of data augmentations are randomly sampled from a base set, and they are applied to the images of the mini-batch. Here, we show an example of rotation and an example of cutmix. Illustrations by John Tenniel, reproduced from Wikimedia.*

在以下章节中，这些方法在卷积模型之外也仍然是基本的。

9.3.1 通过dropout进行正则化

当讨论数据增强时，我们提到一个见解是，增强迫使网络在一个更困难的设置中学习，因此其在更简单环境中的性能可以从准确性和鲁棒性方面得到提高。Dropout [SHK+14] 将这一想法扩展到模型的内部嵌入：通过在训练过程中人工向模型的中间输出引入噪声，该解决方案可以改进。

有许多可能的噪声类型选择：例如，在训练中使用少量高斯噪声

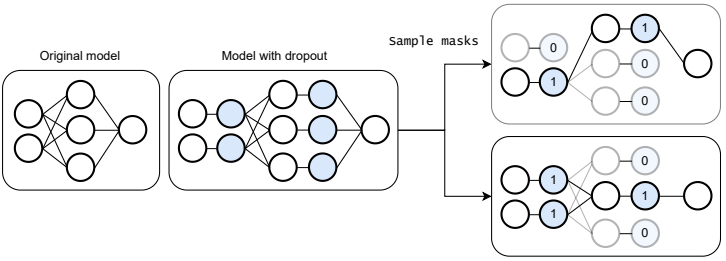


图 F.9.3：*Schematic overview of dropout: starting from a base model, we add additional units after each layer of interest, shown in blue. At training time, each dropout unit is randomly assigned a binary value, masking part of the preceding layers. Hence, we select one out of exponentially many possible models having a subset of active hidden units every time a forward pass is made. Dropout can also be applied at the input level, by randomly removing some input features.*

激活在循环模型文献中始终是一种流行的替代方案。正如其名所示，dropout 的想法是在计算过程中随机移除某些单元（神经元），减少对任何单个内部特征的依赖，并（希望）导致具有良好冗余的训练鲁棒层。

我们定义了全连接层中的dropout，这是其最常见的情况。

定义 D.9.1（Dropout 层）

Denote by $X \sim (n, c)$ *a mini-batch of internal activations of the model (e.g., the output of some intermediate fully-connected layer) with n elements in the mini-batch and c features. In a dropout layer, we first sample a binary matrix* $M \sim Binary(n, c)$ *of the same size, whose elements*

> *are drawn from a Bernoulli distribution with probability* p *(where* $p \in [0, 1\{v^2\}$

$$M_{ij} \sim Bern(p) \tag{E.9.7}$$

> *The output of the layer is obtained by masking the input:*

$$\text{Dropout}(\mathbf{X}) = \mathbf{M} \odot \mathbf{X}$$

> *The layer has a single hyper-parameter,* p*, and no trainable parameters.*

[a]伯尔尼(p)的样本以概率p为1，以概率$1-p$为0。

我们称$1-p$为下降概率。因此，对于迷你批中的任何元素，将有随机数量的单元（大约$(1-p)$%）被设置为0，从而有效地移除它们。这如图F.9.3所示，其中额外的dropout单元用蓝色表示。采样掩码是层的前向传递的一部分：对于两次不同的前向传递，输出将不同，因为不同的元素将被掩码，如图F.9.3右侧所示。

如图所示，我们可以将dropout实现为一个层，该层插入到我们想要正则化的每一层之后。例如，考虑图F.9.3中所示的两个层的全连接模型：

$$y = (\text{FC} \circ \text{FC})(\mathbf{x})$$

在输入和输入上添加dropout正则化

```
模型 = nn.Sequential( nn.Dropout(0.3), nn.Li
near(2, 3), nn.ReLU(), nn.Dropout(0.3), nn.Li
near(3, 1) )
```

C.9.2： *The model in Figure F.9.3 implemented as a sequence of four layers in PyTorch. During training, the output of the model will be stochastic due to the presence of the two dropout layers.*

第一层输出的新模型具有 *four* 层：

$$y = (\text{FC} \circ \text{Dropout} \circ \text{FC} \circ \text{Dropout})(\mathbf{x})$$

查看框C.9.2以获取PyTorch中的实现。

虽然dropout可以提高性能，但输出y现在是一个关于dropout层内部不同掩码采样的随机变量，这在训练后是不理想的。例如，网络的两次正向传递可以返回两个不同的输出，并且某些抽取（例如，包含大量零的抽取）可能是次优的。因此，我们需要一些策略来用确定性操作替换正向传递。

假设我们有一个 {v*} dropout 层。让我们用 M_i 表示第 i 个 dropout 层的掩码，用 $p(\text{M}_1, \ldots, \text{M}_m) = \prod_{i=1}^{m} p(\text{M}_i)$ 表示掩码并集上的概率分布，用 $f(\text{x}; \text{M})$ 表示一旦选择一组给定的掩码 $\text{M} \sim p(\text{M})$，就得到的确定性输出。一个选择是在推理期间用其期望值替换 dropout 影响：

$$f(\mathbf{x}) = \begin{cases} f(\mathbf{x}; \mathbf{M}),\ \mathbf{M} \sim p(\mathbf{M}) & \texttt{[training]} \\ \mathbb{E}_{p(\mathbf{M})}[f(\mathbf{x}; \mathbf{M})] & \texttt{[inference]} \end{cases}$$

我们可以通过蒙特卡洛抽样（附录A）来近似期望值，通过重复抽样掩码值并取平均值：

$$\mathbf{E}_{p(\mathbf{M})}[f(\mathbf{x}; \mathbf{M})] \approx \frac{1}{k} \sum_{i=1}^{k} f(\mathbf{x}; \mathbf{Z}_i),\ \mathbf{Z}_i \sim p(\mathbf{M})$$

这是 k 前向传递的平均值。这被称为蒙特卡洛dropout [GG16]。输出仍然是随机的，但通过适当选择 k，可以控制方差。此外，不同前向传递的输出可以提供对预测不确定性的度量。

然而，执行多次正向传递可能很昂贵。一个更简单（也更常见）的选项是替换随机变量 *layer-by-layer*，这是一个合理的近似。在这种情况下，期望值可以写成封闭形式：

$$\mathbb{E}_{p(\mathbf{M})}[\text{Dropout}(\mathbf{X})] = p\mathbf{X}$$

这是通过一个常数因子 p（放大后的输入，是采样掩码中1的概率）。这导致了一个更简单的公式，逆dropout，其中在训练期间考虑了这种校正：

```
x = torch.randn((16, 2))

# Training with dropout
model.train()
y = model(x)

# Inference with dropout
model.eval()
y = model(x)

# Monte Carlo dropout for inference
k = 10
model.train()
y = model(x[:, None, :].repeat(1, k, 1))
                       .mean(1)
```

盒子C.9.3:
Applying the model from Box C.9.2 on a mini-batch of 16 *examples. For layers like dropout, a framework requires a way to differentiate between a forward pass executed during training or during inference. In PyTorch, this is done by calling the* 训练 *and* 评估 *methods of a model, which set an internal* 训练 *flag on all layers. We also show a vectorized implementation of Monte Carlo dropout.*

$$\text{Dropout}(\mathbf{X}) = \begin{cases} \frac{\mathbf{M} \odot \mathbf{X}}{p} & \texttt{[training]} \\ \mathbf{X} & \texttt{[inference]} \end{cases}$$

在这种情况下，当在推理过程中应用时，dropout层没有效果，可以直接移除。这是大多数框架中首选的实现方式。参见盒C.9.3以了解一些比较。

如我们所述，dropout（可能具有较低的drop概率，如$p = 0.8$或$p = 0.9$）在全连接层中很常见。它也常见于注意力图（将在下一章中介绍）。它不太常见

对于卷积层，其中丢弃输入张量的单个元素会导致过于无结构的稀疏模式。已经设计出考虑图像特定结构的dropout变体：例如，空间dropout [TGJ+15] 丢弃整个张量通道，而cutout [DT17] 丢弃单个通道的空间块。

其他替代方案也是可能的。例如，DropConnect [WZZ+13] 丢弃全连接层的单个权重：

$$\text{DropConnect}(\mathbf{x}) = (\mathbf{M} \odot \mathbf{W})\mathbf{x} + \mathbf{b}$$

DropConnect 在推理中也可以通过矩匹配 [WZZ+13] 高效地近似。然而，这些在实践中较少见，接下来描述的技术更受欢迎。

9.3.2 批量（和层）归一化

处理表格数据时，我们尚未讨论的一种常见预处理操作是归一化，即确保所有特征（输入矩阵的所有列）具有相似的取值范围和统计特性。例如，我们可以预处理数据，将所有列压缩到 $[0, 1]$ 的范围内（最小-最大归一化），或者确保每列具有零均值和单位方差（称为标准缩放、正常缩放或z分数缩放）。

批标准化（BN，[IS15]）复制了这些想法，但针对模型的中间嵌入。这并不简单，因为一个单元（例如，其均值）的统计量在每次梯度下降更新后都会从迭代到迭代发生变化。因此，为了计算一个单元的均值，我们应该在整个训练上执行前向传递

数据集在每个迭代中，这是不可行的。正如其名所示，BN的核心思想是仅使用数据*in the mini-batch itself*来近似这些统计量。

再次考虑任何全连接层$\mathbf{X} \sim (n, c)$的输出，其中n是小批量大小。我们很快就会看到如何将这个想法扩展到图像和其他类型的数据。在BN中，我们仅基于小批量对每个特征（$\mathbf{X}$的每一列）进行归一化，使其具有零均值和单位方差。为此，我们首先计算经验列均值$\mu \sim (c)$和方差$\sigma^2 \sim (c)$：

$$\text{Mean of column } j: \quad \mu_j = \frac{1}{n}\sum_i X_{ij} \tag{E.9.8}$$

$$\text{Variance of column } j: \quad \sigma_j^2 = \frac{1}{n}\sum_i (X_{ij} - \mu_j)^2 \tag{E.9.9}$$

然后我们继续对列进行归一化：

Set the column mean to 0

$$\mathbf{X}' = \frac{\mathbf{X} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

Set the column variance to 1

在考虑标准广播规则（μ和σ^2在第一维上广播），并且$\varepsilon > 0$是添加的小正项以避免除以零的情况下。与表格数据的归一化不同，在归一化中，此操作在训练之前一次性应用于整个数据集，而在BN中，此操作必须在每次前向传递期间为每个小批量重新计算。

选择零均值和单位方差只是一个

约定，不一定是最好的。为了推广它，我们可以让优化算法选择最佳选择，以参数为代价的小额开销。考虑两个可训练参数 $\alpha \sim (c)$ 和 $\beta \sim (c)$ (，我们可以分别初始化为 1 和 0)，我们执行：

$$\mathbf{X}'' = \alpha \mathbf{X}' + \beta$$

与上述类似的广播规则。得到的矩阵将具有第 i 列的均值 β_i 和方差 α_i。BN 层被定义为这两个操作的组合。

> 定义 D.9.2（批量归一化层）
>
> 
>
>
> *Given an input matrix* $\mathrm{X} \sim (n, c)$*, a* ***batch normalization*** *(BN) layer applies the following normalization:*
>
> $$\mathrm{BN}(\mathbf{X}) = \alpha \left(\frac{\mathbf{X} - \mu}{\sqrt{\sigma^2 + \varepsilon}} \right) + \beta$$
>
> *where* μ *and* $\sigma 2$ *are computed according to* (E.9.8) *and* (E.9.9)*, while* $\alpha \sim (c)$ *and* $\beta \sim (c)$ *are trainable parameters. The layer has no hyper-parameters. During inference,* μ *and* σ^2 *are fixed as described next.*

该层只有2c可训练参数，并且当插入到每个块中时，可以显著简化复杂模型的训练。特别是，通常考虑将BN放置在模型的线性组件和非线性组件之间：

$$\mathbf{H} = (\mathrm{ReLU} \circ \mathrm{BN} \circ \mathrm{Linear})(\mathbf{X})$$

在ReLU之前对数据进行居中可以提高

利用其负（稀疏）象限。此外，这种设置使得线性层的偏差变得冗余（因为它与β参数混淆），从而可以将其移除。最后，双线性操作可以很容易地通过标准编译器在大多数框架中进行优化。

BN非常有效，以至于导致了大量关于理解为什么[BGSW18]的文献。原始推导考虑了一个被称为内部协变量偏移的问题，即从单层视角来看，它接收到的输入的统计信息将在优化过程中由于前一层权重的变化而发生变化。然而，当前文献一致认为，BN在优化本身中的效果更为明显，无论是在稳定性方面还是在使用更高学习率的可能性方面，这都是由于缩放和中心化对梯度的综合影响[BGSW18]。[7]

扩展BN到表格数据之外很简单。例如，考虑一个图像嵌入的小批量$X \sim (n, h, w, c)$。我们可以通过将前三个维度一起考虑来对每个通道应用BN，即我们计算一个通道均值，如下：

$$\mu_z = \underbrace{\frac{1}{nhw}}_{\text{Mean of channel } z \text{ (all pixels)}} \sum_{i,j,k} X_{ijkz}$$

[7] See also https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/ for a nice entry point into this literature (and the corresponding literature on developing **normalizer-free** models.

推理过程中的批量归一化

BN在输入预测和它所在的迷你批次之间引入了依赖关系，这在推理过程中是不必要的（换句话说，将图像从一个迷你批次移动到另一个迷你批次将修改其预测）。然而，我们可以利用模型参数在训练后不改变的事实，并将均值和方差冻结到预设值。为此有两种可能性：

1. 训练后，我们对整个训练集进行另一次正向传播，以计算关于数据集 [WJ21] 的经验均值和方差。

2. 更常见的是，我们可以在训练过程中，在每次模型前向传递后更新一组统计数据，并在训练后使用这些数据。为了简化，仅考虑均值，假设我们初始化另一个向量 $\widehat{\mu} = 0$，对应于“均值的滚动均值”。在计算 μ 如 (E. 9.8) 所示后，我们使用指数移动平均更新滚动均值：

$$\widehat{\mu} \leftarrow \lambda\widehat{\mu} + (1-\lambda)\mu$$

在 λ 被设置为一个小值时，例如 $\lambda = 0.01$。假设训练收敛，滚动平均值也将收敛到由选项（1）给出的平均值的近似。因此，在训练后，我们可以通过用（预先计算的）$\widehat{\mu}$ 替换 μ 来使用 BN，对于方差也是如此.[8]

[8] $\widehat{\mu}$ is the first example of a layer's tensor which is part of the layer's state, is adapted during training, but is not needed for gradient descent. In PyTorch, these are referred to as *buffers*.

批归一化的变体

尽管BN在经验上表现良好，但它有几个重要的缺点。我们已经提到了对mini-batch的依赖，这还有其他含义：例如，在训练过程中，μ的方差会因mini-batch较小而变得很大，对于非常小的mini-batch大小，训练可能不可行。此外，在分布式环境中（每个GPU持有mini-batch的不同部分）训练可能很困难。最后，在训练后用不同的值替换μ会在训练和推理之间产生不希望的不匹配。

BN的变体已被提出以解决这些问题。一个常见的想法是保持层的整体结构，但修改执行归一化的轴。例如，层归一化 [BKH16] 计算矩阵在 *the rows* 上的经验均值和方差，即对于每个输入独立地：

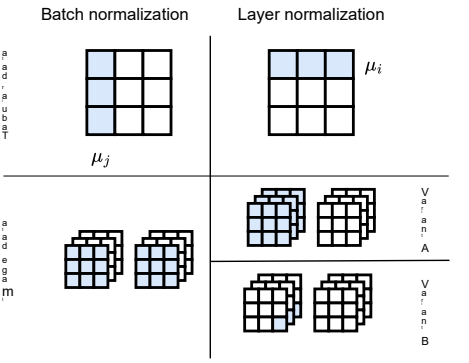
$$\text{Mean of row } i: \quad \mu_i = \frac{1}{c}\sum_j X_{ji} \tag{E.9.10}$$

$$\text{Variance of row } i: \quad \sigma_i^2 = \frac{1}{c}\sum_j (X_{ji} - \mu_i)^2 \tag{E.9.11}$$

考虑图F.9.4，其中我们展示了BN和LN在表格和图像数据上的比较。特别是，我们用蓝色标出所有用于计算单个均值和方差的样本。对于层归一化，我们可以同时计算h、w、c的统计量（变体A）或者分别对每个空间位置进行计算（变体B）。后者在

图 F.9.4:

Comparison between BN and LN for tabular and image data. Blue regions show the sets over which we compute means and variances. For LN we have two variants, discussed better in the main text.



transformer 模型，在下一章中讨论。其他变体也是可能的，例如，组归一化将操作限制为通道的子集，单通道的情况称为实例归一化.[9]

在BN中，我们在(E.9.8)和(E.9.9)中计算统计量的轴与应用可训练参数的轴相同。在LN中，这两者是解耦的。例如，考虑一个应用于维度为(b、3、32、32)的PyTorch LN层：

```
nn.LayerNorm(normalized_shape=[3, 32, 32])
```

这对应于图F.9.4中的变体A。在这种情况下，α和β将与我们在其上计算归一化的轴具有相同的形状，即α、$\beta \sim$ (3、32、32)，总共有$2 \times 3 \times 32 \times 32 = 6144$个可训练参数。必须针对每个框架和模型检查LN和BN的具体实现。

我们最后提到层的另一种常见变体

[9] See https://iclr-blog-track.github.io/2022/03/25/unnormalized-resnets/ for a nicer variant of Figure F.9.4.

归一化，称为均方根归一化（RMSNorm）[ZS19]。它通过去除均值中心和移位简化了对数归一化，对于一个单个输入向量x $\sim (c)$，可以表示为：

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{c}\sum_i x_i^2}} \odot \alpha \tag{E.9.12}$$

当 $\beta =$ 为 0 且数据已经零中心化时，对数归一化（LN）和均方根归一化（RMSNorm）是相同的。

9.4 残差连接

9.4.1 残差连接和残差网络

所有在前一节中看到的技术的组合足以显著增加我们模型中的层数，但仅限于某个上限。考虑三个通用层 f_1、f_2 和 f_3，以及两个模型 g_1、g_2，其中 g_1 是 g_2 的子集：

$$g_1(x) = (f_3 \circ f_1)(x)$$
$$g_2(x) = (f_3 \circ f_2 \circ f_1)(x)$$

直观上，根据万能逼近定理，中间部分f_2应该总是可以逼近恒等函数$f_2(x) \approx x$，在这种情况下$g_2(x) \approx g_1(x)$。因此，总存在一种参数设置，其中第二个（更深）模型应该至少与第一个（较浅）模型表现一样好。然而，如图F.9.5所示，在实践中并未观察到这一点。

我们可以通过偏置网络中的块来解决此问题

图 F.9.5:

Bigger models do not always improve monotonically in training error, despite representing larger classes of functions. Reproduced from [HZRS16].

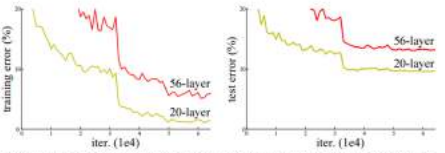


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

朝向恒等函数。这可以通过用所谓的残差（跳过）连接 [HZRS16] 重写一个块 $f(x)$ 来轻松实现：

$$r(x) = f(x) + x$$

因此，我们使用该块来建模与恒等函数的偏差，$f(x) = r(x) - x$，而不是建模与零函数的偏差。这个小小的技巧本身就能帮助训练多达数百层的模型。我们将$f(x)$称为残差路径，$r(x)$称为残差块，由残差块组成的卷积模型称为残差网络（简称ResNet）。

残差连接与残差路径上的批量归一化配合良好，这可以进一步使模型在训练初期偏向于恒等映射 [DS20]。然而，只有在 $f(x)$ 的输入和输出维度相同的情况下才能添加残差连接。否则，可以在残差连接中添加一些缩放。例如，如果 x 是一个图像，而 $f(x)$ 修改了通道数，我们可以添加一个 1×1 卷积：

$$r(x) = f(x) + \text{Conv2D}_{1\times1}(x)$$

残差块的好处也可以从其反向传播的角度来理解。考虑残差块的VJP：

$$\text{vjp}_r(\mathbf{v}) = \text{vjp}_f(\mathbf{v}) + \mathbf{v}^\top \mathbf{I} = \underbrace{\text{vjp}_f(\mathbf{v})}_{\text{VJP of } f} + \underbrace{\mathbf{v}^\top}_{\text{VJP of the skip connection}}$$

因此，前向传播允许输入 x 在跳跃连接上未修改地通过，而反向传播将未修改的回传梯度 v 添加到原始 VJP 中，这有助于缓解梯度不稳定性。

关于残差块的设计

如何设计块 $f(x)$？考虑之前引入的批量归一化块：

$$h = \underbrace{(\text{ReLU} \circ \text{BN} \circ \text{Conv2D})(x)}_{=f(x)} + x$$

因为ReLU的输出始终为正，所以我们有$h \geq x$ (逐元素)。因此，这种形式的残差块堆叠只能增加输入张量的值，或将它设置为零。因此，[HZRS16]中提出的原始设计考虑了类似的块堆叠，除了最后一个激活函数。例如，对于两个块，我们得到以下设计：

$$h = (\text{BN} \circ \text{Conv2D} \circ \text{ReLU} \circ \text{BN} \circ \text{Conv2D})(x) + x$$

一系列这种形式的块可以由一个具有非残差连接的小组件 precede，以降低图像维度，有时称为茎。

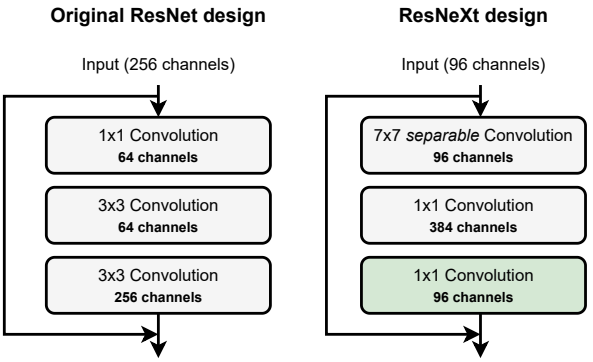


图 F.9.6：*The original ResNet block [HZRS16], and the more recent ResNeXt [LMW+22] block. As can be seen, the design has shifted from an early channel reduction to a later compression (**bottleneck**). Additional details (not shown) are the switch from BN to LN and the use of GELU activation functions. Adapted from [LMW+22].*

该块的超参数特定选择在近年来发生了显著变化。

原始ResNet块提出在第一个操作中对通道数进行压缩，随后是标准的3 × 3卷积，最后在通道数上进行上采样。最近，如图F.9.6)右侧所示的类似ResNeXt块[LMW+22] (的瓶颈层变得流行。为了增加卷积的感受野，初始层被替换为深度卷积。为了利用减少的参数数量，在最后的1 × 1卷积之前，通道数通过一个给定的因子（例如，3×，4×）进行减少。

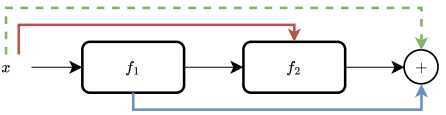


图 F.9.7：*Residual paths: the black, red, and blue paths are implemented explicitly; the green path is only implicit.*

9.4.2 关于残差连接的额外视角

我们通过讨论两个关于残差连接使用的有趣视角来结束这一章节，这两个视角在当前研究中都得到了深入探讨。首先，考虑由两个残差块组成的网络：

$$h_1 = f_1(x) + x \tag{E.9.13}$$
$$h_2 = f_2(h_1) + h_1 \tag{E.9.14}$$

如果我们展开计算：

$$h_2 = f_2(f_1(x) + x) + f_1(x) + x$$

这对应于网络中多个 *paths* 的总和，其中输入要么未修改，要么只经过一次变换（f_1 或 f_2），或者通过它们的组合。

应清楚，此类路径的数量随着残差块数量的指数增长。因此，深度残差模型可以看作是大量较小模型的组合（一个集成），通过权重共享实现。这种观点可以通过测试来证明，例如，ResNets往往对它们元素的小删除或修改具有鲁棒性 [VWB16]。这在图F.9.7中进行了直观展示。

其次，考虑以下用连续参数 t（代表时间）表示的微分方程：

$$\partial_t x_t = f(x, t)$$

我们使用具有参数 x 和 t (a 矢量) 的神经网络来参数化某些函数的时间导数。这被称为常微分方程（ODE）。常微分方程的一个常见问题是，从已知的起始值 x_0 积分到某个指定的时间点 T：

$$x_T = x_0 + \int_{t=0}^{T} f(x, t)dt$$

欧拉方法[10]用于计算x_T，通过选择一个小的步长h并迭代计算一阶离散化：

$$x_t = x_{t-1} + hf(x_{t-1}, t)$$

将 h 合并到 f 中，这对应于一种残差模型的限制形式，其中所有残差块共享相同的权重，每一层对应一个离散的时间瞬间，而 x_T 是网络的输出。从这种观点来看，我们可以直接处理原始的连续时间方程，并通过与现代常微分方程求解器积分来计算输出。这被称为神经常微分方程 [CRBD18]。可以推导出连续时间变体的反向传播，其形式为另一个常微分方程问题。我们将在下一卷中看到神经常微分方程与一类称为正态流 [PNR+21] 的生成模型之间有趣的联系。

[10] https://en.wikipedia.org/wiki/Euler_method

从理论到实践

所有本章讨论的层（批量归一化、dropout等）已在PyTorch、Equinox以及几乎所有其他框架中实现。至于我们描述的其他技术，这取决于框架：对于

示例，权重衰减在所有PyTorch优化器中本地实现，数据增强可以在torchvision（以及其他相应库）中找到作为变换，而早期停止必须手动实现.[11]

1. 在进入下一章之前，我建议您尝试仅使用标准线性代数例程实现dropout或批量归一化作为一层，并将结果与内置层进行比较。

2. 在第7章中，你应该已经实现了一个简单的卷积模型用于图像分类。尝试逐步增加其大小，根据需要添加归一化、dropout或残差连接。

3. 选择一个标准架构，例如ResNet [HZRS16]，或ResNeXt [LMW+22]。尝试按照原始论文中的建议实现整个模型。在

[11] In PyTorch, a common alternative is to use an external library such as PyTorch Lightning to handle the training process. Modifications to the training procedure, such as early stopping, are pre-implemented in the form of *callback* functions.

ImageNet-like datasets在消费者的GPU上可能具有挑战性 - 如果您无法访问良好的硬件或云GPU小时数，您可以继续关注更简单的数据集，例如CIFAR-10。

4. 在这一点上，你可能已经意识到在较小的数据集上从头开始训练非常大的模型（例如，ResNet-50）实际上是不可能的。一种解决方案是使用在线存储库中的权重初始化模型，例如使用在ImageNet上训练的模型的权重，并通过修改最后一层来微调模型，这对应于分类头。到本书的这一部分，这应该相对容易——我建议使用在torchvision或Hugging Face Hub上可用的许多预训练模型之一.[12] 我们将在下一卷中更深入地介绍微调。

[12]For an example tutorial: `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`.

第三部分 洞穴之旅

“It would be so nice if something made sense for a change.”

《爱丽丝梦游仙境》，1951年电影

10 | 变压器模型

关于本章

卷积模型是强大的基线，尤其是在图像和序列中局部关系占主导地位的情况下，但它们在处理非常长的序列或序列元素之间的非局部依赖关系方面存在局限性。在本章中，我们介绍另一类模型，称为转换器，这些模型旨在克服这些挑战。

10.1 长卷积和非局部模型

在上一章讨论的2012-2016年关键发展之后，2016-2017年随着可微分模型设计的下一个重要突破，即变压器 [VSP+17] 的普及，这一架构被设计用来高效处理自然语言处理中的长距离依赖。由于其强大的扩展定律，该架构随后

扩展到其他类型的数据，从图像到时间序列和图，由于在大量数据上训练时具有非常好的缩放规律，它今天在许多领域都是最先进的模型，[KMH+20，BPA+24]。

我们将会看到，变压器的一个有趣方面是数据类型（通过使用适当的标记化器）与架构之间的解耦，其中大部分保持数据无关。这开辟了几个有趣的方向，例如简单的多模态架构和迁移学习策略。我们首先阐述变压器的核心组件，即多头注意力（MHA）层。我们将把对原始变压器模型[VSP+17]的讨论推迟到下一章。

一点历史

历史上，本章的顺序有误：2015年，对于文本而言，循环神经网络（RNNs）是CNNs最常用的替代品。MHA作为RNNs的一个独立组件被引入[BCB15]，之后被用作Transformer模型的核心组件。我们在第13章中介绍了RNNs及其现代形式，线性化RNNs。最近，RNNs已成为语言建模中与Transformer有吸引力的竞争对手。

10.1.1 处理长距离和稀疏依赖

考虑以下两个句子：

> “*The cat is on the table*”

并且一个更长的：

> “
> *The cat, who belongs to my mother, is on the table*”。

为了被可微模型处理，句子必须进行分词，并将标记嵌入为向量（第8章）。从语义角度来看，属于红色单词（猫）和绿色单词（桌子）的标记在这两个句子中都共享类似的依赖关系。然而，这两种情况下的相对偏移量不同，它们的距离可以变得任意大。因此，文本中的依赖关系可以是长程和输入相关的。

用$\mathrm{X} \sim (n,\ e)$ 表示嵌入在 e 维向量中的 n 个标记的句子，用x_i 表示第 i 个标记。我们可以将标记 i 上的大小为 k 的 1D 卷积重写如下：

$$\mathbf{h}_i = \sum_{j=1}^{2k+1} \mathbf{W}_j \mathbf{x}_{i+k+1-j} \tag{E.10.1}$$

每个感受野内的标记都使用一个固定的权重矩阵 W_i 进行处理，该矩阵仅依赖于特定的偏移量 i。在层内建模长距离依赖关系需要我们增加层的感受野，这将在感受野中线性增加参数数量。

一种解决这个问题的可能性如下：而不是

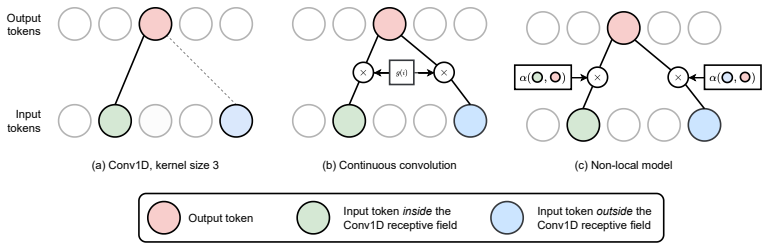


图 F.10.1：*Comparison between different types of convolution for a 1D sequence. We show how one output token (in red) interacts with two tokens, one inside the receptive field of the convolution (in green), and one outside (in blue). (a) In a standard convolution, the blue token is ignored because it is outside of the receptive field of the filter. (b) For a continuous convolution, both tokens are considered, and the resulting weight matrices are given by $g(-1)$ and $g(2)$ respectively. (c) In the non-local case, the weight matrices depend on a pairwise comparison between the tokens themselves.*

显式学习矩阵 W_1、W_2、...，我们可以通过定义一个独立的神经网络块 $g(i)$：$\mathbb{R} \to \mathbb{R}^{e\times e}$ 来定义它们 *implicitly*，该神经网络块输出基于相对偏移 i 的所有权重矩阵。因此，我们将 (E.10.1) 重写为：

$$\mathbf{h}_i = \sum_{j=1}^{n} g(i-j)\mathbf{x}_j$$

The sum is now on *all* tokens

这是一个长卷积，因为卷积跨越整个输入矩阵X。它也被称为连续卷积 [RKG+22]，因为我们可以使用 $g($

•) 对中间位置或变量分辨率进行参数化 [RKG+22]。在这种情况下，参数的数量仅取决于 g 的参数，而它确实

不依赖于 n，序列的长度。定义 g 是非平凡的，因为它需要输出整个权重矩阵。我们可以轻松恢复标准卷积：

$$g(i,j) = \begin{cases} \mathbf{W}_{i-j} & \text{if } |i-j| \leq k \\ 0 & \text{otherwise} \end{cases} \tag{E.10.2}$$

这部分的解决了长距离依赖问题，但它没有解决依赖于输入的依赖问题，因为分配给一个标记的权重仅取决于与索引 i 的相对偏移。然而，这种公式提供了一种简单的方法来解决这个问题，通过让训练函数 g 依赖于标记的 *content* 而不是它们的位位置：

$$\mathbf{h}_i = \sum_{j=1}^{n} g(\mathbf{x}_i, \mathbf{x}_j)\mathbf{x}_j \tag{E.10.3}$$

在计算机视觉的背景下，这些模型也被称为非局部网络 [WGGH18]。我们在图F.10.1中提供了标准卷积、连续卷积和非局部卷积的比较。

10.1.2 注意力层

MHA 层是 (E.10.3) 的简化。首先，处理具有矩阵输出的函数是困难的，所以我们限制层只与标量权重一起工作。特别是，标记之间相似度的一个简单度量是它们的内积（点积）：

$$g(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j$$

我们将会看到，这导致了一个可以轻松并行化的整个序列的算法。对于以下内容，我们考虑点积的归一化版本：

$$g(\mathbf{x}_i, \mathbf{x}_j) = \frac{1}{\sqrt{e}} \mathbf{x}_i^\top \mathbf{x}_j$$

这可以如下解释：如果我们假设 $\mathrm{x}_i \sim \mathcal{N}(0,\ \sigma^2 \mathrm{I})$，$\mathrm{x}_i^\top \mathrm{x}_j$ 的每个元素的方差是 σ^4，因此这些元素的幅度可以很容易地变得非常大。缩放因子确保点积的方差保持在 σ^2。

因为我们对可能可变的标记数量 n 进行求和，因此包括一个归一化操作，例如 softmax：[1] 也很有帮助

$$\mathbf{h}_i = \sum_{j=1}^{n} \mathrm{softmax}_j(g(\mathbf{x}_i, \mathbf{x}_j))\mathbf{x}_j \tag{E.10.4}$$

在此上下文中，我们指的是 $g(\quad \bullet, \bullet)$ 作为注意力评分函数，以及将softmax的输出作为注意力分数。由于softmax的归一化特性，我们可以想象每个标记i可以分配一定量的“注意力”给其他标记：通过增加一个标记的预算，由于softmax中的分母，其他标记的注意力必然会减少。

[1] The notation $\mathrm{softmax}_j$ in (E.10.4) means we are applying the softmax normalization to the set $\left\{g(\mathbf{x}_i, \mathbf{x}_j)\right\}_{j=1}^{n}$, independently for each i. This is easier to see in the vectorized case, described below.

如果我们使用“点积注意力”，我们的g没有可训练参数。注意力层的想法是通过在计算前一个方程之前向输入添加可训练投影来恢复它们。为此，我们定义了三个可训练矩阵$\mathrm{W}_k \sim (e、k)$、$\mathrm{W}_v \sim (e、v)$、$\mathrm{W}_q \sim (e、k)$，其中k和v是超参数。每个标记都使用这三个矩阵进行投影，总共获得$3n$个标记：

$$\text{Key tokens:} \quad \mathbf{k}_i = \mathbf{W}_k^\top \mathbf{x}_i \tag{E.10.5}$$

$$\text{Value tokens:} \quad \mathbf{v}_i = \mathbf{W}_v^\top \mathbf{x}_i \tag{E.10.6}$$

$$\text{Query tokens:} \quad \mathbf{q}_i = \mathbf{W}_q^\top \mathbf{x}_i \tag{E.10.7}$$

这些处理过的标记被称为键、值和查询（现在可以忽略术语的选择；我们将在本节末尾回到这一点）。通过将三个投影（E.10.5）-（E.10.6）-（E.10.7）与（E.10.4）相结合，得到自注意力（SA）层：

$$\mathbf{h}_i = \sum_{j=1}^{n} \mathrm{softmax}_j(g(\mathbf{q}_i, \mathbf{k}_j))\mathbf{v_j}$$

因此，我们通过比较其查询与所有可能的键来计算标记 i 的更新表示，并使用归一化权重来组合相应的值标记。请注意，键和查询的维度必须相同，而值的维度可以不同。

如果我们使用点积，我们可以紧凑地重写所有标记的SA层操作。为此，我们

定义由所有可能的键、查询和值组成的堆栈的三个矩阵：

$$\mathbf{K} = \mathbf{X}\mathbf{W}_k \tag{E.10.8}$$
$$\mathbf{V} = \mathbf{X}\mathbf{W}_v \tag{E.10.9}$$
$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \tag{E.10.10}$$

三个导出矩阵 K、V、Q 的形状分别为 $(n、k)$、$(n、v)$ 和 $(n、k)$，分别。作为旁注，我们还可以将它们实现为单次矩阵乘法，其输出分为三部分：

$$[\mathbf{K} \| \mathbf{V} \| \mathbf{Q}] = \mathbf{X}\left[\mathbf{W}_k \| \mathbf{W}_v \| \mathbf{W}_q\right]$$

$\|$ 表示连接。然后SA层可表示为：

$$\mathrm{SA}(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{k}}\right)\mathbf{V}$$

在本文中，我们假设softmax是按行应用的。我们还可以将投影明确化，如下所示。

定义 D.10.1（自注意力层）

*The **self-attention** (SA) layer is defined for an input* $\mathrm{X} \sim (n, e)$ *as:*

$$\mathrm{SA}(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\right)\mathbf{X}\mathbf{W}_v \tag{E.10.11}$$

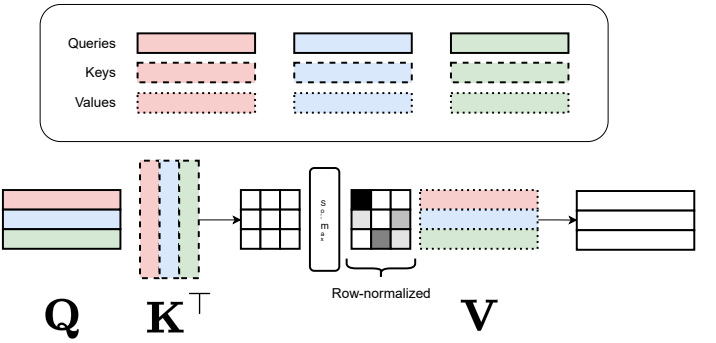


图 F.10.2：*Visualization of the main operations of the SA layer (excluding projections).*

> *The trainable parameters are* $\mathrm{W}_q \sim (k,\ e)$, $\mathrm{W}_k \sim (k,\ e)$ *and* $\mathrm{W}_v \sim (v,\ e)$, *where* k *and* v *are hyper-parameters. Hence, there are* $2ke + ve$ *trainable parameters, independent of* n.

我们在图F.10.2中直观地展示了层的操作。

10.1.3 多头注意力

前一层也称为单头注意力操作。它允许以高灵活性建模跨标记的成对依赖关系。然而，在某些情况下，我们可能需要考虑多个依赖集：再次以“*the cat, which belongs to my mother, is on the table*”为例，"v16"和"v17"之间的依赖关系与"v18"和"v19"之间的依赖关系不同，我们可能希望层能够分别建模它们。[2]

一个多头层通过并行运行多个注意力操作来实现，每个操作都有自己的集合

[2]And everything depends on the cat, of course.

可训练参数，在通过某些池化操作聚合结果之前。为此，我们定义了一个新的超参数 h，我们称之为层的头数。我们对标记实例化了 h 个单独的投影，总共 $3hn$ 个标记（每个“头”$3n$ 个）：

$$\mathbf{K}_e = \mathbf{X}\mathbf{W}_{k,e} \tag{E.10.12}$$
$$\mathbf{V}_e = \mathbf{X}\mathbf{W}_{v,e} \tag{E.10.13}$$
$$\mathbf{Q}_e = \mathbf{X}\mathbf{W}_{q,e} \tag{E.10.14}$$

$\mathrm{W}_{k,e}$ 表示第 e 个头的键投影，其他量同理。多头注意力（MHA）层执行 h 个独立的 SA 操作，堆叠结果输出嵌入，并将它们最终投影到所需的维度：

$$\mathrm{MHA}(\mathbf{X}) = \Big[\underbrace{\mathrm{SA}_1(\mathbf{X})}_{\text{Individual SA layer}} \parallel \ldots \parallel \mathrm{SA}_h(\mathbf{X})\Big] \underbrace{\mathbf{W}_o}_{\text{Output projection}} \tag{E.10.15}$$

其中：

$$\mathrm{SA}_i(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{Q}_i\mathbf{K}_i^\top}{\sqrt{k}}\right)\mathbf{V}_i$$

每个SA操作返回一个形状为$(n、v)$的矩阵。这些h矩阵在第二维上连接，以获得一个形状为$(n、hv)$的矩阵，然后使用矩阵$\mathrm{W}_o \sim (hv、o)$进行投影，其中o是一个额外的超参数，允许在输出维度选择上具有灵活性。

头部和电路

我们将很快看到MHA层总是与残差连接结合使用（第9.4节）。在这种情况下，我们可以将其输出写为i-th标记的：

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \underbrace{\sum_e}_{\text{Sum over heads}} \underbrace{\sum_j}_{\text{Sum over tokens}} \alpha_e(\mathbf{x}_i, \mathbf{x}_j) \mathbf{W}_e^\top \mathbf{x}_j \tag{E.10.16}$$

在 $\alpha_e(\mathrm{x}_i, \mathrm{x}_j)$ 是头 e 中标记 i 和 j 之间的注意力分数，W_e 结合了第 e 个头的值投影与输出投影中的第 e 块。标记嵌入有时被称为模型的残差流。因此，头可以理解为“读取”残差流（通过 W_e 的投影和通过注意力分数的选择），并线性地“写入”流。

[a]这已经在机制可解释性的背景下被普及，它试图逆向工程层的行为以找到可解释的组件称为 *circuits*：`https://transformer-circuits.pub`。流的线性对于分析是基本的。

术语解释

为了理解为什么这三个令牌被称为查询、键和值，我们考虑将SA层与标准的Python字典进行类比，如图C.10.1所示。

形式上，字典是一组形如（键，{v*}）的键值对集合。

```
d = dict()
d["Alice"] = 2
d["Alice"]      # Returns 2
d["Alce"]       # Returns an error
```

盒子 C.10.1: *A dictionary in Python: a value is returned only if a perfect key-query match is found. Otherwise, we get an error.*

值），其中键作为唯一的ID来检索相应的值。例如，在C.10.1框的第三行和第四行中，我们使用两个不同的字符串（“*Alice*”和“*Alce*”）查询字典：字典将查询字符串与存储在其内的所有键进行比较，如果找到完全匹配，则返回相应的值，否则返回错误。

给定一个键对的相似度度量，我们可以考虑一种标准字典的变体，该变体总是返回字典中找到的最近键对应的值。如果键、查询和值都是向量，那么这种字典变体在将softmax操作替换为对标记的argmax操作后，与我们的SA层等价，如图F.10.3所示。

这个“硬”注意力变体难以实现，因为argmax操作的梯度几乎在所有地方都是零（我们将在下一卷中介绍离散采样以及用离散松弛近似argmax操作）。因此，我们可以将SA层解释为一种软近似，其中每个标记都通过基于相应键/查询相似度的所有值的加权组合进行更新。

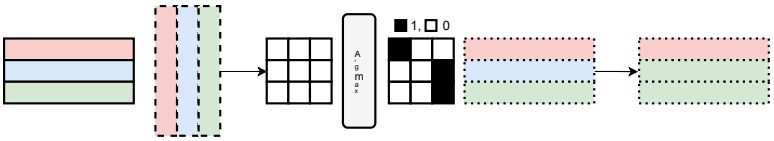


图 F.10.3：*SA with a "hard" attention is equivalent to a vector-valued dictionary.*

10.2 位置嵌入

有了MHA层在手，我们考虑完整Transformer模型的设计，这需要另一个组件，即位置嵌入。

10.2.1 排列等变性

考虑当标记顺序重新排列时MHA层的输出会发生什么是有趣的（*permuted*）。为了形式化这一点，我们引入了置换矩阵的概念。

> 定义 D.10.2（排列矩阵）
>
> *A **permutation matrix** of size n is a square binary matrix $\mathrm{P} \sim Binary(n, n)$ such that only a single 1 is present on each row or column:*
>
> $$\mathbf{1}^\top \mathbf{P} = \mathbf{1}, \ \mathbf{P}\mathbf{1} = \mathbf{1}$$
>
> *If we remove the requirement for the matrix to have binary entries and we only constrain the entries to be non-negative, we obtain the set of **doubly stochastic** matrices (matrices whose rows and columns sum to one).*

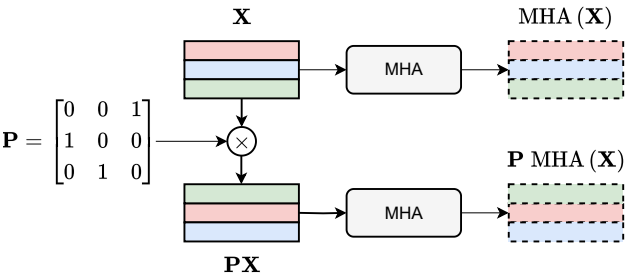


图 F.10.4：*The output of a MHA layer after permuting the ordering of the tokens is trivially the permutation of the original outputs.*

矩阵应用置换矩阵的效果是对矩阵的相应行 / 列进行重新排列。例如，考虑以下置换：

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

查看行，我们发现通过其应用，第二和第三个元素被交换了：

$$\mathbf{P} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_3 \\ \mathbf{x}_2 \end{bmatrix}$$

有趣的是，将置换矩阵应用于MHA层的输入的唯一效果是以等效方式重新排列层的输出：

$$\text{MHA}(\mathbf{PX}) = \mathbf{P} \cdot \text{MHA}(\mathbf{X})$$

这是立即可以证明的。我们关注单头变体，因为多头变体的过程类似。首先，softmax重新规范化矩阵的列元素，因此它在行和列上都是显然的排列等变：

$$\text{softmax}(\mathbf{PXP}^\top) = \mathbf{P}\,[\text{softmax}(\mathbf{X})]\,\mathbf{P}^\top$$

从这我们可以立即得出SA的位置不变性：

$$\text{SA}(\mathbf{PX}) = \text{softmax}\left(\mathbf{P}\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\mathbf{P}^\top\right)\mathbf{PX}\mathbf{W}_v \quad \text{(E.10.17)}$$

$$= \mathbf{P}\cdot\text{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{X}^\top}{\sqrt{k}}\right)\mathbf{X}\mathbf{W}_v = \mathbf{P}\cdot\text{SA}(\mathbf{X}) \quad \text{(E.10.18)}$$

在任意排列矩阵中，我们利用了$\mathrm{P}^\top\mathrm{P} = \mathrm{I}$的事实。这也可以通过为每个标记的SA层进行推理来看到：输出由元素的和给出，每个元素都由成对比较加权。因此，对于给定的标记，操作是对称不变的。相反，对于整个输入矩阵，操作是对称等变的。

翻译等效性是卷积层的一个理想属性，但排列等效性在这里至少是*undesirable* (，因为它丢弃了输入序列的宝贵顺序。例如，处理一个其标记已反转的文本的唯一效果就是反转层的输出，尽管结果的反转输入可能无效。形式上，SA和MHA层被设置为

函数，不是序列函数.[3]

而不是修改层或添加非排列等变层，Transformer通过引入新的位置嵌入概念来操作，这些嵌入是辅助标记，仅取决于标记在序列中的位置（绝对位置嵌入）或两个标记的偏移量（相对位置嵌入）。我们依次描述这两个概念。

10.2.2 绝对位置嵌入

每个输入矩阵X中的标记 $\sim (n, e)$ 代表特定文本片段（例如，一个子词）的 *content*。假设我们将任何序列的最大长度固定为 m 个标记。为了克服位置等变性，我们引入了一个额外的位置嵌入集S $\sim (m, e)$，其中向量S_i唯一编码了“位于位置 i”的概念。因此，输入矩阵与S的第一行的和：

$$\mathbf{X}' = \mathbf{X} + \mathbf{S}_{1:n}$$

是这样一个，$[X']_i$ 表示“位置 i 中的标记 X_i”。因为对位置嵌入进行排列没有意义（因为它们只依赖于位置），所以结果层不再是对称的：

$$\text{MHA}(\mathbf{PX} + \mathbf{S}) \neq \mathbf{P} \cdot \text{MHA}(\mathbf{X} + \mathbf{S})$$

查看图 F.10.5 以了解此想法的直观表示。

我们应该如何构建位置嵌入？最简单的策略是将 S 作为模型参数的一部分考虑，

[3] To be even more pedantic, they are *multiset* functions since tokens can be repeated.

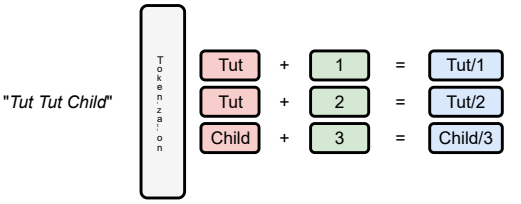


图 F.10.5：*Positional embeddings (green) added to the tokens' embeddings (red). The same token in different positions has different outputs (blue).*

与所有可训练参数一起训练，类似于标记嵌入。当标记数量相对稳定时，这种策略效果很好；我们将在下一章的计算机视觉背景下看到一个例子。

另一种方法是定义一个从标记位置集合到给定向量的确定性函数，该向量唯一标识位置。一些策略显然是糟糕的选择，例如：

1. 我们可以将一个标量 $p = i/m$ 与每个位置关联，该标量与位置线性增加。然而，将单个标量添加到标记嵌入中只有轻微的影响。

2. 我们可以将位置一热编码为一个大小为 m 的二进制向量，但生成的向量将非常稀疏和高维。

一种在原始Transformer论文[VSP+17]中引入的可能性是正弦嵌入。要理解它们，考虑一个正弦函数：

$$y = \sin(x)$$

正弦函数将一个唯一的值分配给范围 $[0, 2\pi]$ 内的任何输入 x。我们还可以改变正弦的频率：

$$y = \sin(\omega x)$$

这根据频率 ω 大小不同而或多或少快速振荡，并为范围 $[0, \frac{2\pi}{\omega}]$ 内的任何输入分配一个唯一值。

与（类比）时钟有相似之处：秒针以$\frac{1}{60}$ Hz的频率（每分钟一次）完成一次完整旋转。因此，可以通过观察指针来区分一分钟内的“时间点”，但通常只能通过模60秒来识别两个时间瞬间。我们在时钟中通过添加一个单独的指针（分针）来克服这一点，该指针以更慢的$\frac{1}{3600}$ Hz频率旋转。因此，通过观察坐标对（秒，分）（时间的“嵌入”）我们可以区分一小时内的任何一点。通过添加另一个以更慢频率旋转的指针（时针），我们可以区分一天内的任何一点。这可以推广：我们可以设计频率更低或更高的时钟来区分月份、年份或毫秒。

一个类似的策略可以在这里应用：我们可以通过一组e正弦波（其中e是一个超参数）对每个位置i进行编码，这些正弦波的频率逐渐增加：

$$\mathbf{S}_i = [\sin(\omega_1 i), \sin(\omega_2 i), \ldots, \sin(\omega_e i)]$$

在实践中，[VSP+17]的原始提案仅使用$e/2$个可能的频率，但添加了正弦和余弦函数：

$$\mathbf{S}_i = \left[\sin(\omega_1 i), \cos(\omega_1 i), \ldots, \sin(\omega_{e/2} i), \cos(\omega_{e/2} i)\right]$$

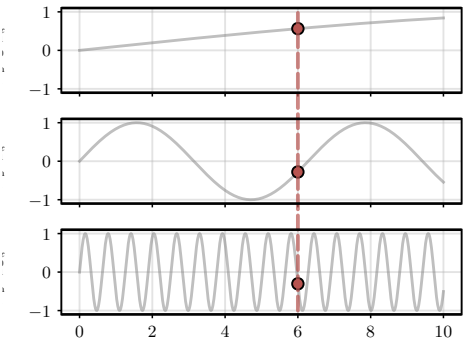


图 F.10.6: *We show three* 正弦 *functions with* 0.1, *and* 1, 10. *The* 6 *embedding for* 4 *is given by the* 2 *corresponding* 5 *values (red circles).*

这可以通过指出，在这个嵌入中，两个位置通过一个简单的线性变换、一个旋转相关联，这个旋转只依赖于两个位置的相对偏移量来证明.[4]只要它们足够大并且以超线性速率增加，任何频率的选择都是有效的。从 [VSP+17] 的选择是一个几何级数：

$$\omega_i = \frac{1}{10000^{i/e}}$$

该值从 $\omega_0 = 1$ 变化到 $\omega_e = \frac{1}{10000}$。参见图 F.10.6 以获取可视化。

10.2.3 相对位置嵌入 s

可训练位置嵌入和正弦位置嵌入是绝对嵌入的例子，因为它们编码序列中的特定位置。对于非常长的序列，相对位置嵌入已成为一种常见的替代方案。在这种情况下，我们不是向标记添加位置编码，而是修改注意力函数，使其依赖于

[4] See https://kazemnejad.com/blog/transformer_architecture_positional_encoding/ for a worked-out computation.

任意两个标记之间的偏移量:

$$g(\mathbf{x}_i, \mathbf{x}_j) \rightarrow g(\mathbf{x}_i, \mathbf{x}_j, i-j)$$

这是我们在本章开头介绍的两个想法的结合（图F.10.1）。请注意，虽然绝对嵌入只添加一次（在输入处），但相对嵌入必须在每次使用MHA层时添加。例如，我们可以添加一个可训练的偏置矩阵$\mathbf{B} \sim (m, m)$，并用偏置相关的偏移量重写点积:

$$g(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^\top \mathbf{x}_j + B_{ij}$$

一个更简单的变体，线性偏差注意力（ALiBi）[PSL22]，考虑每个头部的单个可训练标量，该标量乘以一个偏移矩阵。更高级的策略，如旋转位置嵌入（RoPE），也是可能的 [SAL+24]。

10.3 构建Transformer模型

10.3.1 变压器块和模型

一个模型原则上可以由多个MHA层堆叠而成（softmax提供必要的非线性，以避免多个线性投影的崩溃）。然而，经验上发现，当MHA与一个独立操作的完全连接块交织使用时效果最佳，该块对每个标记独立操作。这两个操作可以理解为混合标记（MHA）和混合通道（MLP），类似于深度可分离卷积模型。

特别是，对于MLP块，通常选择一个

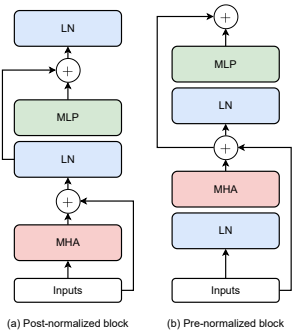


图 F.10.7：*Schematic view of pre-normalized and post-normalized transformer blocks. In the post-normalized variant the LN block is applied after the MHA or MLP operation, while in the pre-normalized one before each layer.*

瓶颈架构由两个形式为 {v*} 的全连接层组成：

$$\text{MLP}(\mathbf{x}) = \mathbf{W}_2\phi\left(\mathbf{W}_1\mathbf{x}\right)$$

在 $\mathrm{x} \sim (e)$ 是一个标记，$\mathrm{W}_1 \sim (p,\ e)$，其中 p 被选为 e 的整数倍，例如 $p = 3e$ 或 $p = 4e$），并且 $\mathrm{W}_2 \sim (e,\ p)$ 重新投影回原始嵌入维度。由于增加的隐藏维度提供了足够的自由度，因此通常移除偏差。

为确保深度模型的效率训练，我们还需要一些额外的正则化策略。特别是，对于MHA和MLP块，通常包括两个层归一化步骤和两个残差连接。根据层归一化应用的位置，我们获得两种基本Transformer块变体，有时称为预归一化和后归一化。这些在图F.10.7中显示。

虽然归一化后的版本对应于原始的Transformer块，但通常发现预归一化的变体更稳定且训练速度更快 [XYH+20]。图F.10.7中块的设计，从根本上讲，是一种经验选择，文献中已经提出了许多变体并进行了测试。我们

稍后在本节11.3中回顾这些内容。

我们现在可以完成基本变压器模型的描述：

1. 对原始输入序列进行分词并嵌入到矩阵$\mathbf{X} \sim (n, e)$中
。2. 如果使用绝对位置嵌入，将其添加到输入矩阵中。
3. 应用上述讨论的1个或多个块。

4. 根据任务包括一个最终的头部。

步骤（3）的输出是一组处理过的标记 $\mathbf{H} \sim (n, e)$，其中 n 和 e 都没有被变换器模型改变（前者因为我们没有在集合上执行局部池化操作，后者因为块中的残差连接）。例如，考虑一个分类任务，我们可以通过对标记进行池化并使用全连接块来应用标准的分类头：

$$y = \text{softmax}\left(\text{MLP}\left(\frac{1}{n}\sum_i \mathbf{H}_i\right)\right)$$

这部分与其对应的CNN设计相同。然而，变换器具有许多有趣的属性，这主要源于它将输入作为一个（多重）集合进行操作，在整个架构中不改变其维度。我们接下来研究一个简单的例子。

10.3.2 类令牌和寄存器令牌

虽然到目前为止我们假设每个标记对应于我们输入序列的一部分，但没有任何东西阻止我们向transformer的输入添加*additional*个标记。这严格取决于其特定的架构：例如，CNN需要其输入精确排序，并且不清楚我们如何向图像或序列添加额外的标记。这是一个非常强大的想法，我们在这里只考虑两种特定的实现。

首先，我们考虑使用一个类别标记 [DBK+21]，这是一个显式添加的额外标记，用于分类以替换上面的全局池化操作。假设我们初始化一个单个可训练标记 $\mathrm{c} \sim (e)$，并将其添加到输入矩阵中：

$$\mathbf{X} \leftarrow \begin{bmatrix} \mathbf{X} \\ \mathbf{c}^\top \end{bmatrix}$$

新矩阵的形状为 $(n+1, e)$。对于迷你批次的序列，类标记是相同的。在上面的步骤（3）之后，变换器输出一个矩阵 $\mathrm{H} \sim (n+1, e)$，其中包含所有标记的更新表示，包括类标记。想法是，而不是对标记进行池化，模型应该能够将有关分类任务的所有信息“压缩”到类标记中，我们可以通过简单地丢弃所有其他标记来重写分类头：[5]

$$y = \mathrm{softmax}\left(\mathrm{MLP}\left(\mathbf{H}_{n+1}\right)\right)$$

[5] In the language of circuits and heads from Section 10.1.3, we could say equivalently that the model must learn to move all information related to the task in the residual stream of the class token.

额外的可训练标记即使不明确使用也可能很有用。例如，[DOMB24] 已表明添加一些额外的标记（在此情况下称为寄存器）可以通过为模型提供使用寄存器“存储”不依赖于特定位置的辅助信息的机会来提高注意力图的质量。

从理论到实践

我们将介绍许多与变压器相关的重要概念，下一章中。因此，对于这一章，我建议一个稍微不寻常的练习，结合卷积骨干和类似transformer的头部 - 如图F.10.8所示。

您在第7章和第9章中开发的卷积模型被应用于一个 *single* 图像。然而，有时我们有可用的同一识别对象的 *set* 张图像——例如，在一个监控系统，我们可能有多个可疑人员的截图。这在文献中被称为多视角系统，每个图像被称为对象的视角。多视角模型应该为整个视角集提供单个预测，同时对输入视角的顺序保持不变。在这个练习中，我们将实现一个简单的多视角模型——见图F.10.8。

1. 使用任何图像分类数据集，您可以通过对输入（图F.10.8中的灰色块）应用固定数量的数据变换来模拟一个多视图模型。忽略批量维度，对于每个形状为$x \sim (h、w、c)$ (高度、宽度、通道)的输入图像，您获得一个

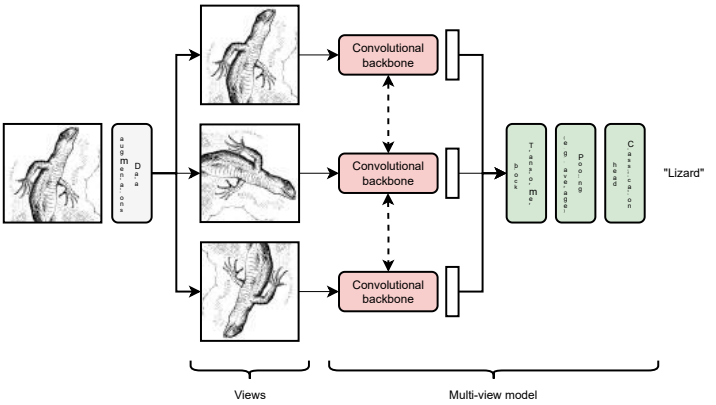


图 F.10.8：*Multi-view model to be implemented in this chapter. The image is augmented through a set of random data augmentation strategies to obtain a set of* ***views*** *of the input (* gray *). Each view is processed by the same convolutional backbone to obtain a fixed-sized dimensional embedding (* red *). The set of embeddings are processed by a transformer block before the final classification (* green *). Illustration by John Tenniel.*

> *multi-view* 输入形状为 $x' \sim (v,\ h,\ w,\ c)$，其中 v 是视图数。一个单独的标签 y 与此张量相关联——原始图像的标签。视图数也可以在不同的小批量之间不同，因为没有模型的任何部分被限制在预指定的视图数。

2. 多视角模型由三个组件组成。用 $g(x)$ 表示一个将单个视图处理为固定维度的嵌入的模型——例如，这可以是您为之前的练习训练的任何卷积骨干。全模型的第一个部分（图 F.10.8 中的红色部分）将 g 并行应用于所有视图，$\mathrm{h}_i = g(x_i) \sim (e)$，其中 e 是

超参数（骨干网络的输出大小）。

3. 在连接视图的嵌入后，我们得到一个矩阵 $\mathbf{H} \sim (v, e)$。为了使整个模型具有排列不变性，对 H 应用任何组件都必须是排列等变的.[6] 为了完成这个练习，根据第10.3.1节，实现并应用一个单个的Transformer块。你可以使用基本的PyTorch实现MHA，或者尝试使用einops进行更高级的实现.[7] 你还可以与torch.nn中预实现的版本进行比较。

4. 变换器块不修改输入形状。为了完成模型，对视图（在本场景中代表标记）进行平均，并应用最终的分类头。您还可以尝试添加一个类别标记（第10.3.2节）。很容易证明以这种方式构建的模型在视图方面是排列不变的。

[6] An average operation over the views is the simplest example of permutation invariant layer. Hence, removing the MHA block from Figure F.10.8 is also a valid baseline. Alternatively, deep sets [ZKR+17] characterize the full spectrum of linear, permutation invariant layers.

[7] See https://einops.rocks/pytorch-examples.html.

11 | 变换器在实际应用中

关于本章

我们现在考虑基本变换器模型的一些变体，包括编码器-解码器架构、因果MHA层以及图像和音频领域的应用。

11.1 编码器-解码器转换器

第10章中我们描述的模型可以用于对给定序列进行回归或分类。然而，原始的Transformer [VSP+17]是一个更复杂的模型，旨在执行所谓的序列到序列（seq2seq）任务。在seq2seq任务中，输入和输出都是序列，它们之间的标记没有简单的对应关系。一个显著的例子是机器翻译，其输出是在不同语言中对输入序列的翻译。

一种为seq2seq任务构建可微分模型的可能性是编码器-解码器（ED）设计 [SVL14]。ED模型由两个模块组成：一个编码器，它将输入序列处理为转换后的表示（可能为固定维度），以及一个解码器，它根据编码器的输出自回归地生成输出序列。我们之前描述的转换器模型可用于构建编码器：这种类型的转换器用于分类时被称为仅编码器转换器。为了构建解码器，我们需要两个额外的组件：一种使模型因果（以执行自回归）的方法，以及一种将计算条件化到单独输入（编码器的输出）的方法。

11.1.1 因果多头注意力

让我们首先考虑使Transformer块具有因果性的问题。唯一交互的组件是MHA块。因此，拥有因果性的MHA变体就足以使整个模型具有因果性。记住，对于卷积，我们通过适当地掩蔽卷积滤波器中的权重来设计因果性变体。对于MHA，我们可以掩蔽所有不满足因果性属性的标记之间的交互：

$$\text{Masked-SA}(\mathbf{X}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top \odot \mathbf{M}}{\sqrt{k}}\right)\mathbf{V}$$

执行softmax内部的掩码是必要的。考虑以下（错误）的变体：

$$\textbf{Wrong}: \left(\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{k}}\right) \odot \mathbf{M}\right)\mathbf{V}$$

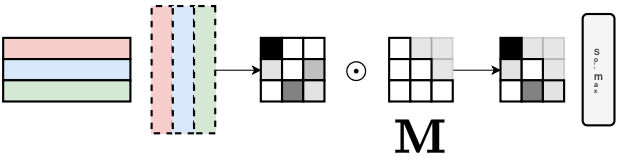


图 F.11.1：*Visual depiction of causal attention implemented with attention masking.*

因为softmax中的分母，所有标记都参与每个标记的计算，无论后续的掩码如何。此外，请注意，对于非因果链接，将M_{ij} =设置为0不起作用，因为$\exp(0) = 1$。因此，MHA掩码变体的正确实现是选择一个上三角矩阵，其上部分为$-\infty$，因为$\exp(-\infty) = 0$如所期望：

$$M_{ij} = \begin{cases} -\infty & \text{if } i > j \\ 1 & \text{otherwise} \end{cases}$$

实际上，值可以设置为一个非常大的负数（例如，-10^9）。

11.1.2 交叉注意力

其次，让我们考虑将MHA层的输出条件化为独立输入块的问题。为此，让我们通过明确分离输入矩阵的三种出现来重写MHA操作：

$$\text{SA}(\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3) = \text{softmax}\left(\frac{\mathbf{X}_1 \mathbf{W}_q \mathbf{W}_k^\top \mathbf{X}_2^\top}{\sqrt{k}}\right) \mathbf{X}_3 \mathbf{W}_v$$

SA层对应于$\mathrm{X}_1 = \mathrm{X}_2 = \mathrm{X}_3 = \mathrm{X}$（巧合的是，这也解释了我们给它取的名字）。然而，

该公式也适用于考虑属于不同集合的键、值和查询的情况。一个重要的情况是交叉注意力（CA），其中我们假设键和值是从第二个矩阵 $\mathbf{Z} \sim (m, e)$ 计算得出的：

$$\mathrm{CA}(\mathbf{X}, \mathbf{Z}) = \mathrm{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{W}_k^\top\mathbf{Z}^\top}{\sqrt{k}}\right)\mathbf{Z}\mathbf{W}_v \qquad \text{(E.11.1)}$$

Cross-attention between $\mathbf{X}$ and $\mathbf{Z}$

如此，$\mathrm{CA}(\mathrm{X}, \mathrm{Z}) = \mathrm{SA}(\mathrm{X}, \mathrm{Z}, \mathrm{Z})$。其解释是，X的嵌入根据其与Z提供的（键，值）对集合的相似性进行更新：我们称X在Z上为*cross-attending*。请注意，这种公式与两个输入标记集的连接非常相似，随后是对注意力矩阵的适当掩码。

与前馈层比较

考虑(E.11.1)中交叉注意力操作的简化变体，其中我们显式地参数化键和值矩阵：[1]

$$\mathrm{NeuralMemory}(\mathbf{X}) = \mathrm{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q\mathbf{K}}{\sqrt{k}}\right)\mathbf{V} \qquad \text{(E.11.2)}$$

层现在由查询投影矩阵 W_q 和两个矩阵 K 和 V 参数化。（E.11.2）被称为记忆层 [SWF+15]，在行上

[1] See also the discussion on the perceiver network in Section 11.2.1.

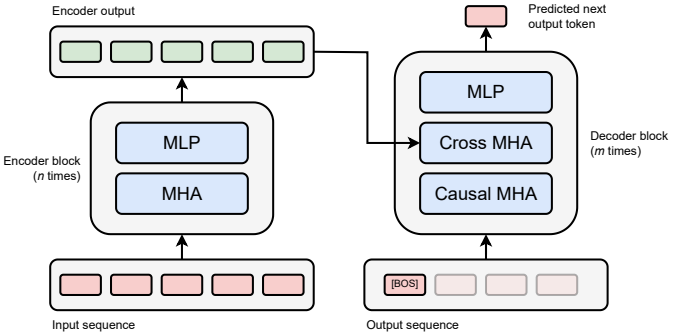


图 F.11.2：*Encoder-decoder architecture, adapted from [VSP+17]. Padded tokens in the decoder are greyed out.*

键值矩阵被模型用于存储有趣的模式，这些模式可以通过类似注意力的操作动态检索。如果我们进一步通过设置$\mathbf{W}_q = \mathbf{I}$简化层，忽略$\sqrt{k}$的归一化，并用通用激活函数$\phi$替换softmax，我们得到一个两层MLP：

$$\text{MLP}(\mathbf{X}) = \phi\left(\mathbf{XK}\right)\mathbf{V} \tag{E.11.3}$$

因此，在Transformer网络中，MLP可以看作是对可训练键和值的注意力操作的近似。可视化训练数据中最接近的标记显示了人类可理解的模式 [GSBL20]。

11.1.3 编码器-解码器变换器

使用这两个组件，我们准备讨论原始的Transformer模型，如图F.11.2所示。首先，输入序列X经过一个

[2]A pedantic note: technically, Transformer (upper-cased) is a proper noun in [VSP+17]. In the book, I use transformer (lower-cased) to

标准变换器模型（称为编码器），提供更新后的嵌入序列H。接下来，输出序列由另一个变换器模型（称为解码器）自回归地预测。与编码器不同，解码器每个块有三个组件：

1. MHA层的掩码变体（以确保自回归是可能的）。

2. 一个查询由输入序列嵌入H给出的交叉注意力层。3. 一个标准的基于标记的MLP。

解码器仅用模型也是可能的，在这种情况下，解码器的第二个块被移除，只使用掩码MHA和MLP。大多数现代LLM都是通过解码器仅用模型训练来自动回归生成文本标记[RWC+19]来构建的，如下文所述。实际上，随着意识到许多seq2seq任务可以通过将输入序列连接到生成的输出序列直接使用解码器仅用模型来解决，编码器-解码器模型已经变得不那么常见，如第8.4.3节所述。

11.2 计算考虑事项

11.2.1 时间复杂度和线性时间变换器

MHA性能并非无代价：由于每个标记必须关注所有其他标记，其复杂度高于简单的卷积

refer to any model composed primarily of attention layers.

```
def self_attention(Q: Float[Array, "n k"],
                   K: Float[Array, "n k"],
                   V: Float[Array, "n v"]
                  ) -> Float[Array, "n v"]:
    return nn.softmax(Q @ K.T) @ V
```

C.11.1: *Simple implementation of the SA layer, explicitly parameterized in terms of the query, key, and value matrices.*

操作。为了理解这一点，我们从两个角度来考察其复杂性：内存和时间。我们使用SA层的朴素实现作为参考，如图C.11.1所示。

让我们首先看看时间复杂度。softmax内的操作以$\mathcal{O}(n^2k)$的规模增长，因为它需要计算n^2个点积（每个标记对一个）。将其与仅按序列长度线性增长的1D卷积层进行比较。*Theoretically*，这种复杂性的二次增长对于非常长的序列可能是个问题，这在LLMs中很常见。

这导致了加速自回归生成的几种策略的发展（例如，投机解码 [LKM23]），以及线性或亚二次变体的Transformer。例如，我们可以用具有 *trainable* 个标记集Z的交叉注意力层替换SA层，其中标记的数量可以作为超参数选择并由用户控制。这种策略由Perceiver架构 [JGB+21]普及，以将原始标记集蒸馏成更小的潜在瓶颈。设计线性化Transformer有许多替代策略：我们在第11.3节和第13章讨论了几种变体。

重要的是，如框中所示的一种实现

C.11.1 可以证明在现代硬件上高度依赖于内存 [DFE+22]，这意味着其计算成本主要由内存和I/O操作主导。因此，线性时间注意力变体的理论收益与硬件上的实际加速并不相关。结合可能的性能降低，这使得它们不如MHA的强优化实现有吸引力，例如下文所述的一个。

11.2.2 在线softmax

在内存方面，Box C.11.1中的实现也有一个二次 n^2 复杂度因子，因为在计算过程中，注意力矩阵$QK^\top$被完全实现。然而，这是不必要的，并且可以通过将计算分块并仅在最后执行softmax归一化来将这种复杂性显著降低到线性因子 [RS21]。

为了理解这一点，考虑一个单个查询向量 q，并假设我们将我们的键和值分成两个块，这些块依次加载到内存中：

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_1 \\ \mathbf{K}_2 \end{bmatrix}, \quad \mathbf{V} = \begin{bmatrix} \mathbf{V}_1 \\ \mathbf{V}_2 \end{bmatrix} \tag{E.11.4}$$

如果我们忽略softmax中的分母，我们可以分解SA操作，依次计算每个块的结果：

$$\mathrm{SA}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \frac{1}{L_1 + L_2} [\mathbf{h}_1 + \mathbf{h}_2] \tag{E.11.5}$$

在对于两个块 $i = 1, 2$ 我们定义了两个

辅助量：

$$\mathbf{h}_i = \exp(\mathbf{K}_i \mathbf{q}) \mathbf{V}_i \tag{E.11.6}$$

$$L_i = \sum_j [\exp(\mathbf{K}_i \mathbf{q})]_j \tag{E.11.7}$$

记住我们分别将块加载到内存中，因此对于块1，我们计算h_1和L_1；然后我们卸载前一个块，并计算块2的h_2和L_2。请注意，除非我们跟踪所需的额外统计信息L_i （，否则该操作不是完全可分解的，这些统计信息用于计算softmax操作的归一化系数)。更一般地，对于多个块$i = 1, \ldots, m$，我们将有：

$$\mathrm{SA}(\mathbf{q}, \mathbf{K}, \mathbf{V}) = \frac{1}{\sum_{i=1}^m L_i} \left[\sum_{i=1}^m \mathbf{h}_i \right] \tag{E.11.8}$$

因此，我们可以设计一个简单的迭代算法，其中对于加载到内存中的每个键值块，我们只更新并存储(E.11.8)中的分子和分母的累积和，只在最后进行归一化。这个技巧（有时称为*online* softmax），结合IO感知实现和内核融合，导致了高度内存和计算高效的注意力实现，如FlashAttention-2.[3]通过将查询组分配给不同的设备并将键和查询的块在设备之间旋转，也可以设计出注意力的分布式实现（例如，RingAttention [LZA23]）。针对特定硬件优化操作可能导致一些反直觉的行为，例如*increased*速度对更长的序列

[3] https://github.com/Dao-AILab/flash-attention

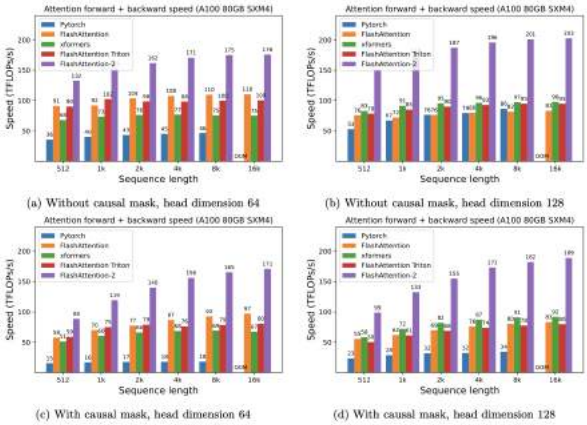


图 F.11.3：*Official benchmark of FlashAttention and FlashAttention-2 on an NVIDIA A100 GPU card, reproduced from* https://github.com/Dao-AILab/flash-attention.

长度 - 见图 F.11.3。

11.2.3 KV缓存

MHA的重要实现方面发生在处理仅解码器模型中的自回归生成时。对于要生成的每个新标记，只需计算注意力矩阵的新行和一个值标记，这意味着可以存储在内存中的先前键和值，如图F.11.4所示。这被称为KV缓存，并且是大多数MHA优化实现的标准。

KV缓存的大小随序列长度线性增加。再次强调，您可以将其与因果卷积层的等效实现进行比较，其中内存大小由感受野的大小限制。在自回归生成中设计具有固定内存成本的表示层是一个有吸引力的

图 F.11.4：*To compute masked self-attention on a new token, most of the previous computation can be reused (in gray). This is called the **KV cache**.*

因子，第13章。

11.2.4 图像和音频的Transformer

Transformer最初是为文本开发的，很快成为语言建模的默认选择。特别是，流行的GPT-2模型[RWC+19] (及其后续变体)是一种仅包含解码器的架构，通过预测文本序列中的标记进行预训练。大多数开源LLM，如LLaMa [TLI+23]，遵循类似的架构。相比之下，BERT [DCLT18]是另一个基于仅编码器架构的流行预训练词嵌入家族，该架构经过训练以预测掩码标记（掩码语言建模）。与GPT类模型不同，BERT类模型不能用于生成文本，而只能用于文本嵌入或作为微调架构的第一部分。用于语言建模的编码器-解码器模型也存在（例如，T5家族[RSR+20]），但它们已经变得不那么受欢迎了.[4]

从高层次的角度来看，一个Transformer由三个组件组成：一个标记化/嵌入步骤，

[4] **Diffusion language models** [YTL+25] (DLMs) are a recent alternative to autoregressive LLMs, and they are trained with a denoising objective vaguely reminiscent of BERT models. All types of LLMs undergo several *post-training* steps beyond token prediction (e.g., instruction tuning), which we do not have space to cover here.

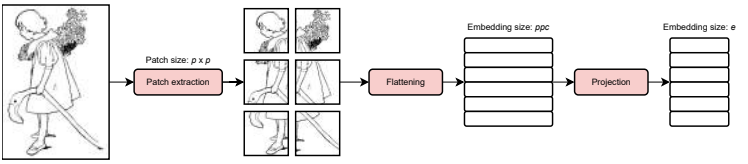


图 F.11.5：*Image tokenization: the image is split into non-overlapping patches of shape $p \times p$ (with p an hyper-parameter). Then, each patch is flattened and undergoes a further linear projection to a user-defined embedding size e. c is the number of channels of the input image.*

将原始输入转换为一系列向量；位置嵌入以编码原始序列的顺序信息；以及变压器块本身。因此，可以通过定义适当的标记化程序和位置嵌入来设计用于其他类型数据的变压器。

让我们首先考虑计算机视觉。在像素级别对图像进行标记化过于昂贵，因为随着序列长度的二次增长，复杂度也在增加。视觉Transformer（ViTs，[DBK+21]）的核心思想是将原始输入分割成固定长度的非重叠补丁，然后将这些补丁展平并投影到预定义大小的嵌入中，如图F.11.5所示。

嵌入步骤如图 F.11.5 所示，可以使用步长等于内核大小的卷积层实现。或者，像 einops[5] 这样的库将 einsum 操作（第 2.1 节）扩展到允许将元素分组为预定义形状的块。一个示例在框 C.11.2 中显示。

[5] http://einops.rocks

```
from einops import rearrange
# A batch of images
xb = torch.randn((32, 3, 64, 64))

# Define the operation: differently from
# standard einsum, we can split the output
# in blocks using brackets
op = 'b c (h ph) (w pw) \
                -> b (h w) (ph pw c)'

# Run the operation with a given patch size
patches = rearrange(xb, op, ph=8, pw=8)
print(patches.shape) # [Out]: (32, 64, 192)
```

C.11.2框：*einops can be used to decompose an image into patches with a simple extension of the einsum syntax.*

原始ViT使用了可训练的位置嵌入以及一个额外的类别标记来执行图像分类。ViT还可以通过预测行主序或列主序的补丁来用于图像生成。在这种情况下，我们可以训练一个独立的模块，将每个补丁转换为离散的标记集，例如使用向量量化变分自编码器 [CZJ+22]，或者我们可以直接处理连续输出 [TEM23]。然而，对于图像生成，通常更倾向于使用其他非自回归方法，如扩散模型和流匹配；我们将在下一卷中介绍它们。

通过开发适当的标记化机制和位置嵌入，变压器也被用于音频，特别是语音识别。在这种情况下，通常有一个小的1D卷积模型（带有池化）作为标记化块 [BZMA20，RKX+23]。例如，Wav2Vec [BZMA20] 是一个仅编码器模型，其输出使用扩展的交叉熵损失进行训练，称为连接主义

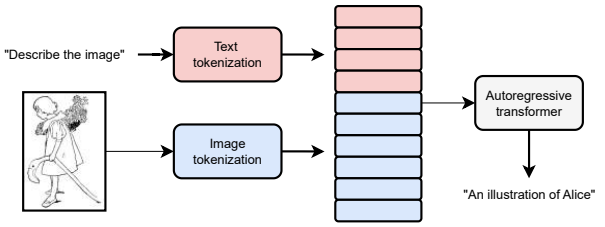


图 F.11.6：*An example of a **bimodal** transformer that operates on both images and text: the outputs of the two tokenizers are concatenated and sent to the model.*

时间分类损失 [GFGS06]，以对齐输出嵌入到转录。由于具有精确对齐的标记数据稀缺，Wav2Vec 模型使用掩码语言建模损失的变体在大量未标记音频上进行预训练。相比之下，Whisper [RKX+23] 是一个编码器-解码器模型，其中解码器被训练来自动回归生成转录。这为模型提供了更多灵活性，并减少了强烈标记数据的需求，但以转录阶段可能出现的 *hallucinations* 为代价。神经音频编解码器也可以训练成将音频压缩成一系列离散标记 [DCSA23]，这些标记反过来又成为文本到语音生成等生成应用的基

变换器也可以定义用于时间序列 [AST+24]，图（将在下一章中介绍），以及其他类型的数据。数据与架构之间的解耦也是多模态变体的基础，这些变体可以接受（或提供）不同模态的输入（或输出）。这是通过使用其各自的标记器对每个模态（图像、音频、...）进行标记化，并将不同的标记连接成一个单一序列 [BPA+24] 来实现的。我们在图 F.11.6 中展示了图像-文本模型的示例。

11.3 变种变压器

我们通过讨论基本变换器块的一些有趣变体来结束这一章。首先，为非常大的变换器设计了几种变体，以略微减少计算时间或参数数量。例如，并行块 [DDM+23] 并行执行 MLP 和 MHA 操作：

$$\mathbf{H} = \mathbf{H} + \text{MLP}(\mathbf{H}) + \text{MHA}(\mathbf{H})$$

这种方式，MLP和MHA层中的初始和最终线性投影可以融合，以实现更高效的实现。作为另一个例子，多查询MHA [Sha19]为每个头共享相同的键和值投影矩阵，仅查询不同。

更一般地，我们可以在保持transformer块整体结构的情况下，用更简单的（序列长度线性复杂度）操作替换MHA层，即交替进行标记和通道混合，并使用层归一化和残差连接。例如，假设序列长度是固定的（例如，在计算机视觉中，补丁的数量可以事先固定）。在这种情况下，MHA层可以被一个操作在单个输入通道上的MLP所替换，对应于嵌入的一个维度。这种类型的模型被称为混合模型 [THK+21] - 见图F.11.7。忽略归一化操作，这可以写成交替的MLP在输入矩阵的转置上：

$$\mathbf{H} = \text{MLP}(\mathbf{H}) + \mathbf{H} \tag{E.11.9}$$

$$\mathbf{H} = \left[\text{MLP}(\mathbf{H}^\top) + \mathbf{H}^\top\right]^\top \tag{E.11.10}$$

其他混合器模型的变体也使用{v*}可能。

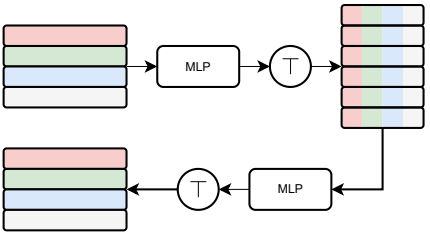


图 F.11.7：*Mixer block, composed of alternating MLPs on the rows and columns of the input matrix.*

例如，1D卷积、傅里叶变换或池化。特别是，在S2-MLP [YLC+22]模型中，令牌混合操作被替换为一个对其输入的平移版本应用的一个更简单的MLP。这类模型的一般类别被[YLZ+22]称为MetaFormers。

门控（乘法）交互也可以用于块的组合。在这种情况下，多个块并行执行，但它们的输出通过Hadamard乘法组合。我们可以将一个通用的门控单元表示为：

$$f(\mathbf{X}) = \phi_1(\mathbf{X}) \odot \phi_2(\mathbf{X}) \tag{E.11.11}$$

在 ϕ_1 和 ϕ_2 是可训练块的情况下。例如，使用 $\phi_1(\mathrm{X}) = \sigma(\mathrm{XA})$ 和 $\phi_2(\mathrm{X}) = \mathrm{XB}$，我们获得第 5.4 节中描述的门控线性单元（GLU）。

作为少数代表性示例，gMLP模型[LDSL21]在混音模型中使用门控单元而不是通道混合块；LLaMa系列模型[TLI+23]使用类似于GLU的单元而不是标准的MLP块；而门控注意力单元（GAU）[HDLL22]使用一个具有单个头部的更简单的类似注意力模型，用于ϕ_1和一个线性投影用于ϕ_2。这些设计在最近的一些循环模型变体中特别受欢迎，将在第13章中进一步讨论。

为了进一步简化设计，多线性算子网络（MONet）移除了所有激活函数以定义

一个仅由线性投影和逐元素乘法组成的块 [CCGC24]：

$$\mathbf{H} = \mathbf{E}(\mathbf{AX} \odot \mathbf{BX} + \mathbf{DX})$$

E类似于变换器块中的输出投影，DX充当残差连接，B通过低秩分解实现以减少参数数量[CCGC24]。为了引入标记混合，在模型的全部奇数块中实现了标记移位操作。

从理论到实践

有许多有趣的练习可以在这一点上进行——你几乎成了设计可微模型的专家！首先，使用任何图像分类数据集，你可以尝试从头实现如第11.2.4节所述的视觉Transformer，遵循[DBK+21]来选择超参数。在小型数据集上从头训练ViT相当具有挑战性[LLS21，SKZ+21]，除非你有足够的计算能力来考虑百万规模的数据集，否则要做好失望的准备。你也可以尝试一个更简单的变体，例如第11.3节中描述的Mixer模型。所有这些练习都应该相对简单。

1. 对于图像分词，您可以使用Einops，如Box C.11.2中所述，或采用其他策略（例如，使用大步长的卷积）。对于小图像，您还可以尝试使用每个像素作为标记。

2. 对于位置嵌入，所有在 中描述的策略

第10.2节是有效的。对于ViT来说，最简单的一个方法是初始化一个*trainable*嵌入矩阵，但我建议你尝试正弦和相对嵌入作为练习。

您也可以尝试实现一个类似GPT的小型模型。在文本数据的标记化中有很多复杂之处我们没有涉及。然而，minGPT仓库[6]是一个出色的教学实现，您可以将其作为起点使用。

[6] https://github.com/karpathy/minGPT

12 | 图模型

关于本章

在这一章中，我们考虑图结构数据，即由一组（已知）关系连接的节点。图在现实世界中无处不在，从蛋白质到交通网络、社交网络和推荐系统。我们引入了专门用于处理图的层，这些层被广泛归类为消息传递层或图变换器架构。

12.1 基于图数据的机器学习

12.1.1 图及其在图上的特征

目前我们已考虑的数据要么是完全非结构化的（以向量表示的表格数据），要么是以简单方式结构化的，包括集合、序列和图像等网格。然而，许多类型的数据由其组成部分之间更复杂的依赖关系定义。例如，分子由仅通过化学键稀疏连接的原子组成。许多种类的网络（社交网络、交通网络、能源网络）

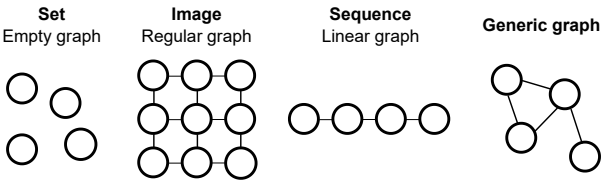


图 F.12.1：*Graphs generalize many types of data: sets can be seen as empty graphs (or graphs having only self-loops), images as regular graphs, and sequences as linear graphs. In this chapter we look at more general graph structures.*

网络）由数百万个单元（人、产品、用户）组成，这些单元仅通过一小部分连接进行交互，例如道路、反馈或友谊。这些在图论语言中更自然地定义。本章的目的是介绍可微模型来处理以这种方式定义的数据。

在其最简单形式中，一个图可以通过一对集合 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ 来描述，其中 $\mathcal{V} = \{1, \ldots, n\}$ 是节点（顶点）集合，而：

$$\mathcal{E} = \left\{(i, j) \mid \underbrace{i, j \in \mathcal{N}}_{\text{Two nodes of the graph}}\right\}$$

图中的边集。在大多数数据集中，节点数 {v*} 和边数 {v*} 可以从图到图变化。

图泛化了许多我们已经看到的概念：例如，只包含形式为 $(i、i)$ 的自环的图表示一组对象，而包含所有可能边的图（全连接图）与注意力层相关联，正如我们接下来所展示的。图像可以是

表示为图，通过将每个像素与图的节点关联，并根据规则的网格状结构连接相邻像素 - 见图 F.12.1.[1]

图中的连接可以等价地表示为一个称为邻接矩阵的矩阵表示。这是一个二进制方阵 $\mathrm{A} \sim \text{Binary}(n, n)$，使得：

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathscr{E} \\ 0 & \text{otherwise} \end{cases}$$

在此格式中，一个集合由单位矩阵 $\mathrm{A} = \mathrm{I}$ 表示，一个全连接图由全为1的矩阵表示，一个图像由托普利茨矩阵表示。连接始终是双向的图（即，(i, j) 和 (j, i) 总是作为边对出现）称为无向图，我们有 $\mathrm{A}^\top = \mathrm{A}$。为了简单起见，我们将处理无向图，但方法可以很容易地扩展到有向情况。我们注意到还有其他矩阵表示方法，例如，关联矩阵 $\mathrm{B} \sim \text{二进制}(n, |\mathscr{E}|)$ 是这样的，如果节点 i 参与边 j，则 $B_{ij} = 1$，并且我们因为每条边恰好连接两个节点，所以有 $\mathrm{B1}^\top = 2$。参见图 F.12.2 中的示例。

我们将假设我们的图具有自环，即 $A_{ii} = 1$。如果邻接矩阵没有自环，我们可以通过重新分配它来添加它们：

$$\mathbf{A} \leftarrow \mathbf{A} + \mathbf{I}$$

[1] There are many variants of this basic setup, including heterogenous graphs (graphs with different types of nodes), directed graphs, signed graphs, etc. Most of them can be handled by variations of the techniques we describe next.

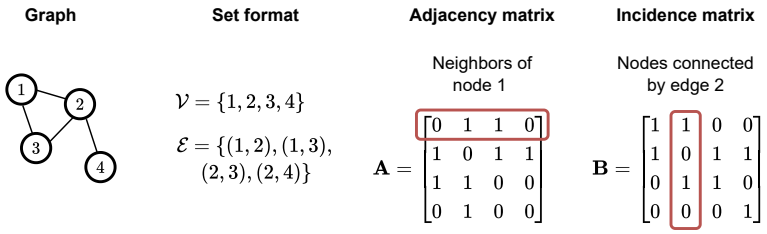


图 F.12.2：*We can represent the graph connectivity in three ways: as a set $\mathcal{E}$ of pairs (second column); as an (n, n) adjacency matrix (third column); or as an $(n, |\mathcal{E}|)$ incidence matrix (fourth column).*

12.1.2 图特征

图具有描述它们的多种可能特征。例如，分子中的原子和键可以通过表示其类型的分类特征来描述；交通网络中的道路可以具有容量和交通流量；而在社交网络中的两个朋友可以通过他们相识的年数来描述。

通常，这些特征可以分为三种类型：与每个节点关联的节点特征、与每个边关联的边特征以及与整个图关联的图特征。我们将从最简单的情况开始，即只有非结构化节点特征可访问，即每个节点 i 都关联一个向量 $\mathrm{x}_i \sim (c)$。然后，完整的图可以通过两个矩阵 $\mathrm{X} \sim (n、c)$ 来描述，我们称之为特征矩阵，以及邻接矩阵 $\mathrm{A} \sim (n、n)$。

在大多数情况下，节点的顺序无关紧要，即如果我们考虑一个排列矩阵 $\mathrm{P} \sim$ 二元$(n，n)$ (参见第10.2节)，一个图及其排列版本在本质上相同，换句话说：

$$(\mathrm{X}, \mathrm{A}) \text{ 与 } (\mathrm{PX}, \mathrm{PAP}^{\top}) \text{ 是相同的图}$$

注意，排列矩阵通过交换X中的行来作用，同时它在邻接矩阵中交换行和列。

某些特征也可以直接从图的拓扑结构中提取。例如，我们可以将一个标量值 d_i 与每个节点关联，称为度数，它描述了它与多少个节点相连：

$$d_i = \sum_j A_{ij}$$

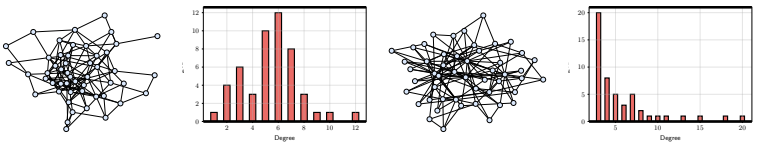
图中的度分布是图本身的一个重要特征，如图 F.12.3 所示。我们可以将度数收集到一个称为度矩阵的单个对角矩阵中：

$$\mathbf{D} = \begin{bmatrix} d_1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & d_n \end{bmatrix}$$

我们可以使用度矩阵来定义多种类型的 *weighted* 邻接矩阵。例如，行归一化的邻接矩阵定义为：

$$\mathbf{A}' \leftarrow \mathbf{D}^{-1}\mathbf{A}_{ij} \;\rightarrow\; A'_{ij} = \frac{1}{d_i} A_{ij}$$

这是在$\sum_i A'_{ij} = 1$的意义上归一化的。我们还可以定义一个列归一化的邻接矩阵为



(a) *Erdős–Rényi* **(b)** *Degree* **(c)** *Barabasi-Albert* **(d)** *Degree*

图 F.12.3：*(a) Random graph generated by drawing each edge independently from a Bernoulli distribution (**Erdős–Rényi model**). (b) These graphs show a Gaussian-like degree distribution. (c) Random graph generated by adding nodes sequentially, and for each of them drawing* 3 *connections towards existing nodes with a probability proportional to their degree (**preferential attachment process** or **Barabasi-Albert model**). (d) These graphs have a few nodes with many connections acting as hubs for the graph.*

$\mathrm{A}' = \mathrm{AD}^{-1}$。这两个矩阵都可以解释为图上的“随机游走”，即在给定一个节点 i 的情况下，归一化邻接矩阵的对应行或列表示向其任何邻居随机移动的概率分布。一个更一般的对称归一化邻接矩阵如下所示：

$$\mathbf{A}' = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$$

这是由 $A'_{ij} = \frac{A_{ij}}{\sqrt{d_i d_j}}$ 定义，根据连接的两个节点的度数给每个连接赋予权重。邻接矩阵及其加权变体都具有以下性质：当 $(i,\ j) \notin \mathcal{E}$ 时，$A_{ij} =$ 为 0。在信号处理术语中，这些被称为图移位矩阵。

稀疏矩阵

考虑一个6节点图的通用邻接矩阵（尝试作为练习绘制该图）：

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

邻接矩阵非常稀疏（许多零）。这是一个重要的属性，因为稀疏矩阵有定制的实现和操作它们的技巧，比它们的稠密版本具有更好的计算复杂度.[a]

[a]作为一个例子，在JAX中：https://jax.readthedocs.io/en/latest/jax.experimental.sparse.html。

12.1.3 图上的扩散操作

我们感兴趣的基本图操作被称为扩散，这对应于根据图拓扑对节点特征的平滑。为了理解它，考虑每个节点上的一个标量特征，我们将其收集在向量 $\mathrm{x} \sim (n)$ 中，并对以下特征进行操作：

$$\mathbf{x}' = \mathbf{A}\mathbf{x}$$

在 A 可以是邻接矩阵、归一化变体或任何加权邻接矩阵的情况下。我们可以重新编写这个

按节点操作如下：

$$x_i' = \sum_{j \in \mathcal{N}(i)} A_{ij} x_j$$

我们已定义1-hop邻域：

$$\mathcal{N}(i) = \left\{ j \mid \underbrace{(i,j) \in \mathcal{E}}_{\text{All edges with node } i \text{ as a vertex}} \right\}$$

如果我们把节点特征解释为物理量，那么通过邻接矩阵的投影可以看作是一个“扩散”过程，该过程将每个节点上的量替换为其邻域内量的加权平均值。

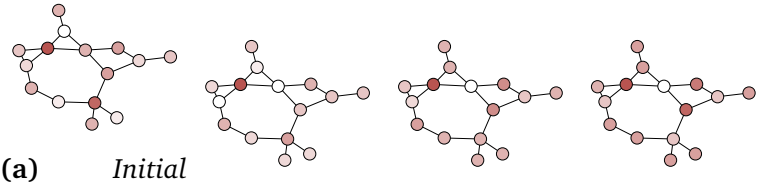
另一个在图分析中的基本矩阵是拉普拉斯矩阵：

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

无论邻接矩阵是否归一化，度矩阵都计算为 $D_{ii} = \sum_j A_{ij}$ 。拉普拉斯算子的一次扩散可以表示为：

$$[\mathbf{Lx}]_i = \sum_{(i,j) \in \mathcal{E}} A_{ij}(x_i - x_j) \tag{E.12.1}$$

我们可以从这里看出，拉普拉斯算子与图上的梯度概念密切相关，其分析是谱图理论领域的核心。作为一个



(a) *Initial graph* **(b)** *10 steps* **(c)** *20 steps* **(d)** *30 steps*

图 F.12.4：*(a) A random graph with 15 nodes and a scalar feature on each node (denoted with variable colors). (b)-(d) The result after* 10*,* 20*, and* 30 *steps of diffusion with the Laplacian matrix. The features converge to a stable state.*

示例，在（E.12.1）中，1始终是与零特征值（特别是最小的那个）相关的拉普拉斯算子的特征向量。我们在图F.12.4中展示了使用拉普拉斯矩阵的扩散示例。

12.1.4 流形正则化

从（E.12.1）我们还可以推导出一个基于拉普拉斯算子的二次型：

$$\mathbf{x}^\top \mathbf{L}\mathbf{x} = \sum_{(i,j)\in\mathcal{E}} A_{ij}(x_i - x_j)^2 \tag{E.12.2}$$

非正式地说，这是一个标量值，用于衡量图上信号的“平滑”程度，即信号在图中连接的节点对之间变化的快慢。为了了解这个概念的一个简单应用，考虑一个表格分类数据集 $\mathcal{S}_n = \{(\mathrm{x}_i,\ y_i)\}$。假设我们在这个数据集上构建一个图，其中每个节点是数据集的一个元素，邻接矩阵是基于特征之间的距离构建的：

$$A_{ij} = \begin{cases} \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2) & \text{if } \|\mathbf{x}_i - \mathbf{x}_j\|^2 < \tau \\ 0 & \text{otherwise} \end{cases} \tag{E.12.3}$$

在 τ 是一个用户定义的超参数的情况下。给定一个分类模型 $f(\mathrm{x})$，我们可能希望约束其输出对于相似的输入相似，其中相似性按比例定义为 (E.12.3)。为此，我们可以将图的特征定义为我们的模型输出：

$$\mathbf{f} = \begin{bmatrix} f(\mathbf{x}_1) \\ \vdots \\ f(\mathbf{x}_n) \end{bmatrix} \sim (n)$$

二次型（E.12.2）精确地告诉我们相似输入在预测方面的变化程度：

$$\mathbf{f}^\top \mathbf{L}\mathbf{f} = \sum_{i,j} A_{ij}(f(\mathbf{x}_i) - f(\mathbf{x}_j))^2 \tag{E.12.4}$$

通过正则化优化问题找到最优模型，其中正则化器由（E.12.4）给出：

$$f^*(\mathbf{x}) = \arg\min \left\{ \sum_{i=1}^{n} L(y_i, f(\mathbf{x})) + \lambda \ \mathbf{f}^\top \mathbf{L}\mathbf{f} \right\}$$

在 L 是一个通用损失函数且 λ 是一个标量超参数的情况下：

这是称为流形正则化 [BNS06]，它可以作为一种通用的正则化工具来强制模型

在图上保持平滑，其中邻接关系要么给出，要么由用户构建，如（E.12.3）中所述。这在半监督场景中特别有用，因为我们有一个小的标记数据集和一个来自同一分布的大规模未标记数据集，因为（E.12.4）中的正则化器不需要标签 [BNS06]。然而，模型的预测仅依赖于单个元素 x_i，并且在训练后图被丢弃。在下一节中，我们将介绍将连通性嵌入到模型本身的更自然的方法。

12.2 图卷积层

12.2.1 图层属性

为了设计预测依赖于连接性的模型，我们可以通过邻接矩阵的知识增强标准层 $f(\mathrm{X})$，即我们考虑以下形式的层：

$$\mathbf{H} = f(\mathbf{X}, \mathbf{A})$$

在之前，$\mathrm{X} \sim (n,\ c)$ (与 n 节点数和 c 每个节点的特征) 以及 $\mathrm{H} \sim (n,\ c')$，即操作不改变图的连通性，并为图中的每个节点 i 返回一个更新的嵌入 $\mathrm{H}_i \sim (c')$。对于接下来的内容，A 可以是邻接矩阵或具有相同稀疏模式的任何矩阵（一个图移位矩阵），包括加权邻接矩阵、拉普拉斯矩阵等。

由于对图中的节点进行排列不应影响最终预测，因此层不应依赖于节点的特定顺序，即对于任何排列矩阵 P，层的输出应为

排列等变：

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = \mathbf{P} \cdot f(\mathbf{X}, \mathbf{A})$$

我们可以为图层定义一个类似于图像情况下的“局部性”概念。为此，我们首先引入子图的概念。给定全图中节点集合 $\mathcal{T} \in \mathcal{V}$，我们定义由 $\mathcal{T}$ 诱导的子图为：

$$\mathcal{G}_\mathcal{T} = (\mathbf{X}_\mathcal{T}, \mathbf{A}_\mathcal{T})$$

在X$_\mathcal{T}$是一个(|$\mathcal{T}$|，c)矩阵，收集$\mathcal{T}$中节点的特征，而A ∼ (|$\mathcal{T}$|，|$\mathcal{T}$|)是全邻接矩阵的对应块。

> 定义 D.12.1（图局部性）
>
> *A graph layer* H = f(X, A) *is* ***local*** *if for every node,* H$_i$ = f(X$_{\mathcal{N}(i)}$, A$_{\mathcal{N}(i)}$)*, where* $\mathcal{N}(i)$ *is the* 1*-hop neighborhood of node* i*.*

这与考虑图像中距离为1的所有像素类似，但在此情况下，（a）$\mathcal{N}(i)$中的节点没有特定的顺序，并且（b）$\mathcal{N}(i)$的大小会根据i有很大的变化。因此，我们不能像在图像情况下那样定义卷积，因为它的定义需要这两个属性（想想卷积层中的权重张量）。

对于以下内容，请注意我们可以将局部性的定义扩展到1-hop邻居之外。例如，2-hop邻域 $\mathcal{N}^2(i)$ 被定义为距离最多为2的所有节点：

$$\mathcal{N}^2(i) = \bigcup_{j \in \mathcal{N}(i)} \mathcal{N}(j)$$

∪ 是集合的并集运算符。我们可以扩展

定义考虑高阶邻域的局部性，并设计3 × 3、5 × 5等滤波器的等效。

12.2.2 图卷积层

为了定义一个模仿卷积层的图层，我们需要它具有排列等变（而不是平移等变）和局部性。MHA层自然具有排列等变性，但它不具有局部性，并且它不显式依赖于邻接矩阵A。我们将在下一节中看到对此的可能扩展。现在，让我们关注一个更简单的全连接层：

$$f(\mathbf{X}, \_) = \phi(\mathbf{XW} + \mathbf{b})$$

在 $\mathrm{W} \sim (c,\ c')$ 和 $\mathrm{b} \sim (c')$ 处。这同样是自然排列等变的，但它不依赖于图的连通性，该连通性被忽略。为了构建一个适当的可微层，我们可以交替层的操作与扩散步骤。

定义 D.12.2（图卷积）

Given a graph represented by a node feature matrix $\mathrm{X} \sim (n,\ c)$ *and a generic graph-shift matrix* $\mathrm{A} \sim (n, n)$ *(the adjacency, the Laplacian, ...), a* ***graph convolutional*** *(GC) layer is given by [KW17]:*

$$f(\mathbf{X}, \mathbf{A}) = \phi(\mathbf{A}(\mathbf{XW} + \mathbf{b}))$$

where the trainable parameters are $\mathrm{W} \sim (c, c')$ *and*

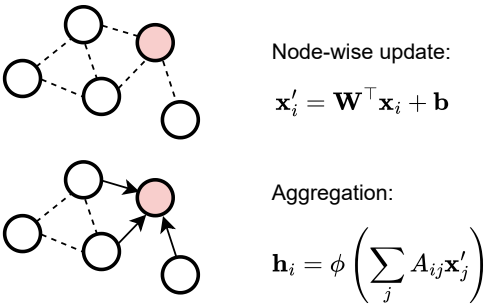


图 F.12.5：*Two stages of a GC layer: each node updates its embedding in parallel to all other nodes; the output is given by a weighted average of all updated embeddings in the node's neighbourhood.*

> b $\sim (c')$, *with* c' *an hyper-parameter.* ϕ *is a standard activation function, such as a ReLU.*

注意与标准卷积层的相似性：我们通过矩阵W执行“通道混合”操作，通过矩阵A执行“节点混合”操作，区别在于前者在这种情况下不可训练（由于节点之间再次存在变量度数以及需要使层排列等变）。参见图F.12.5以获取可视化。通过利用图信号处理的概念也可以更正式地证明这种类比，这超出了本书的范围 [BBL+17]。

忽略偏差，我们可以将此公式重写为单个节点 i 的形式：

$$\mathbf{H}_i = \phi\left(\sum_{j\in\mathcal{N}(i)} A_{ij}\mathbf{X}_j\mathbf{W}\right)$$

因此，我们首先对所有节点嵌入（通过右乘W给出）进行同时更新。然后，每个节点计算其自身及其邻居的更新节点嵌入的加权平均值。由于邻居的数量可以从节点到节点变化，使用邻接矩阵的归一化版本可以在训练中显著帮助。对于该层，证明排列等变性是显而易见的：

$$f(\mathbf{PX}, \mathbf{PAP}^\top) = \phi\left(\mathbf{PAP}^\top \mathbf{PXW}\right) = \mathbf{P} \cdot f(\mathbf{X}, \mathbf{A})$$

12.2.3 构建图卷积网络

一个GC层是局部的，但多层堆栈不是。例如，考虑一个双层GC模型：

$$f(\mathbf{X}, \mathbf{A}) = \phi(\mathbf{A}\ \underbrace{\phi\left(\mathbf{AXW}_1\right)}_{\text{First GC layer}}\ \mathbf{W}_2) \tag{E.12.5}$$

具有两个可训练的权重矩阵 W_1 和 W_2。与图像情况类似，我们可以定义一个感受野的概念。

> 定义 D.12.3（图感受野）
>
> *Given a generic graph neural network* $\mathrm{H} = f(\mathrm{X}, \mathrm{A})$*, the* ***receptive field*** *of node* i *is the smallest set of nodes* $\mathcal{V}(i) \in \mathcal{V}$ *such that* $\mathrm{H}_i = f(\mathrm{X}_{\mathcal{V}(i)}, \mathrm{A}_{\mathcal{V}(i)})$.

对于一个单层GC层，感受野是 $\mathcal{V}(i) = \mathcal{N}(i)$。对于如（E.12.5）中的双层网络，我们需要考虑邻居的邻居，以及感受野

成为 $\mathcal{V}(i) = \mathcal{N}^2(i)$。一般来说，对于 k 层的堆叠，我们将有一个 $\mathcal{V}(i) = \mathcal{N}^k(i)$ 的感受野。从图中任意两个节点移动所需的最小步数被称为图的大径。大径定义了实现所有节点全局感受野所需的最小层数。

多项式GC层

另一种方法是增加 *a single* GC 层的感受野。例如，如果我们从邻接矩阵中移除自环，我们也可以通过考虑邻接矩阵的平方，使层相对于 $\mathcal{N}^2(i)$ 而不是 $\mathcal{N}(i)$ 本地化：

$$\mathbf{H} = \phi\left(\mathbf{X}\mathbf{W}_0 + \mathbf{A}\mathbf{X}\mathbf{W}_1 + \mathbf{A}^2\mathbf{X}\mathbf{W}_2\right)$$

在其中有三个参数集 W_0、W_1 和 W_2 分别用于处理自环、邻居及其邻居。这被称为多项式 GC 层。可以通过更高的幂获得更大的感受野。通过考虑多项式 [BGLA21] 的比率，可以设计更复杂的层。

我们可以将GC层与标准归一化层、残差连接、dropout或其他任何是排列等变的操作相结合。与图像情况不同，池化更难，因为没有立即的方法来子采样图连通性。通过利用图论工具或添加额外的可训练组件，池化层仍然可以定义，但它们较少见 [GZBA22]。

表示为 $\mathrm{H} = f(\mathrm{X}, \mathrm{A})$ 提供每个节点更新嵌入的通用层组合（无

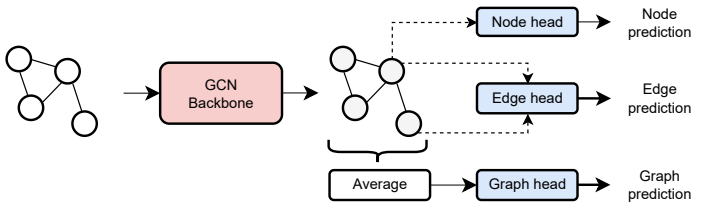


图 F.12.6：*Different types of graph heads: (a) node tasks need to process the features of a single node; (b) edge tasks require heads that are conditioned on two nodes simultaneously; (c) graph tasks can be achieved by pooling all node representations into a fixed-dimensional vector.*

修改连接性）。类似于CNN，我们称其为骨干网络。通过在这些表示的顶部添加一个小头，我们可以完成通用图卷积网络（GCN）的设计：

$$y = (g \circ f)(\mathbf{X}, \mathbf{A})$$

头部设计取决于我们试图解决的问题。最常见的任务分为三个基本类别之一：节点级任务（例如，节点分类）、边级任务（例如，边分类）或图级任务（例如，图分类）。我们依次简要考虑每个示例，见图F.12.6。

节点分类

首先，假设输入图描述了一种社交网络，其中每个用户都与一个节点相关联。对于给定的一组用户，$\mathcal{T} \subseteq \mathcal{V}$，我们知道一个标签 y_i，$i \in \mathcal{T}$，(，例如，用户是否为真实用户、机器人或另一种类型的自动化资料)。我们感兴趣的是预测所有其他节点的标签。在这种情况下，我们可以通过处理每个更新的节点嵌入来获得节点级预测，例如：

$$\hat{y}_i = g(\mathbf{H}_i) = \text{softmax}(\text{MLP}(\mathbf{H}_i))$$

运行此操作在整个矩阵 H 上，为我们提供了所有节点的预测，但我们只知道一小部分的真实标签。我们可以通过丢弃训练集外的节点来训练 GCN：

$$\arg\min \frac{1}{|\mathcal{T}|} \sum_{i \in \mathcal{T}} \text{CE}(\hat{y}_i, y_i)$$

CE是交叉熵损失。重要的是，即使我们丢弃了训练集外节点的输出预测，由于GCN内部的扩散步骤，它们的输入特征仍然参与训练过程。然后，可以在训练后运行GCN一次来对剩余节点进行分类。这种只有训练数据的一个子集被标记的情况被称为半监督问题。

边缘分类

作为第二个例子，假设我们有一个 *edges* 的子集的标签，即 $\mathcal{T}_E \subseteq \mathcal{E}$。例如，我们的图可能是一个交通网络，其中我们只知道部分道路上的交通流量。在这种情况下，我们可以通过添加一个依赖于两个连接节点的特征的头部来获得边级预测，例如通过将它们连接起来：

$$\hat{y}_{ij} = g(\mathbf{H}_i, \mathbf{H}_j) = \text{MLP}\left(\left[\mathbf{H}_i \parallel \mathbf{H}_j\right]\right)$$

对于二元分类（例如，通过0到1之间的标量值预测两个用户的亲和力）我们可以通过考虑它们之间的点积来简化这一点

两个特征：

$$\hat{y}_{ij} = \sigma(\mathbf{H}_i^\top \mathbf{H}_j)$$

与之前一样，我们可以通过最小化已知边的损失来训练网络。

图分类

最后，假设我们感兴趣的是对整个图进行分类（或回归）。例如，该图可能是一个分子，我们想要预测其某些化学性质，例如对给定化合物的反应性。我们可以通过汇总节点表示（例如，通过求和）来处理结果固定维度的嵌入：

$$y = \text{MLP}\left(\frac{1}{n}\sum_{i=1}^{n}\mathbf{H}_i\right)$$

最终池化层使网络 *invariant* 变为节点的排列。在这种情况下，我们的数据集将由多个图（例如，几个分子）组成，使其类似于标准图像分类任务。对于节点和边任务，相反，一些数据集可能由单个图组成（例如，一个大型的社交网络），而其他数据集可以包含多个图（例如，来自不同城镇的几个未连接的道路网络）。这引发了如何高效构建图的小批量的问题。

12.2.4 关于图神经网络实现

如上所述，与图工作的一个特点是几个矩阵可以非常稀疏。例如，考虑以下邻接矩阵：

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

这对应于一个有三个节点的图，节点1和节点3之间有一个双向边。我们可以通过只存储非零值的索引来更有效地存储它，例如，在代码中：

```
A = [[0,2], [2,0]]
```

这是一个坐标列表格式。对于非常稀疏的矩阵，像这样的专用格式可以减少存储，但也可以显著提高对稀疏矩阵或稀疏和稠密矩阵组合进行操作的计算时间。以pytorch-sparse[2]为例，它支持在PyTorch中实现高效的转置和几种类型的矩阵乘法。这也在层的实现中得到了体现。在PyTorch Geometric[3] (中，这是PyTorch)中用于处理图神经网络的最常见的库之一，层的正向传播通过提供图的特征和作为边坐标列表的连接性进行参数化。

[2] https://github.com/rusty1s/pytorch_sparse

[3] https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html#learning-methods-on-graphs

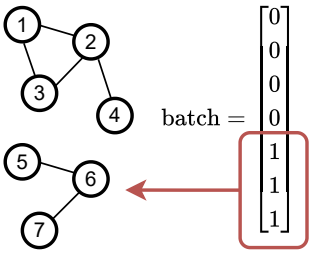


图 F.12.7：*Two graphs in a mini-batch can be seen as a single graph with two disconnected components. In order to distinguish them, we need to introduce an additional vector containing the mapping between nodes and graph IDs.*

与稀疏矩阵一起工作在迷你批次方面有另一个有趣的后果。假设我们拥有 b 个图 $(\mathrm{X}_i,\ \mathrm{A}_i)_{i=1}^{b}$。对于每个图，我们都有相同数量的节点特征 c 但节点数量不同 n_i，因此 $\mathrm{X}_i \sim (n_i,\ c)$ 和 $\mathrm{A}_i \sim$ 二元$(n_i,\ n_i)$。为了构建一个迷你批次，我们可以创建两个秩为3的张量：

$$X \sim (b, n, c) \tag{E.12.6}$$

$$A \sim \text{Binary}(b, n, n) \tag{E.12.7}$$

在 $n = \max(n_1,\ ...,\ n_b)$ 的位置，并且两个矩阵都通过填充零来填充两个张量。然而，通过注意到在 GC 层中，任何两个节点（通过边序列的路径）都不会通信，我们可以得到一个更优雅的替代方案。因此，我们可以通过简单地合并所有节点来构建一个描述整个 mini-batch 的 *single* 图。

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_b \end{bmatrix} \tag{E.12.8}$$

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{A}_b \end{bmatrix} \tag{E.12.9}$$

在X $\sim (\sum_i n_i,\ c)$ 和 A $\sim$ Binary$(\sum_i n_i,\ \sum_i n_i)$ 的位置。小批量图的邻接矩阵具有分块对角结构，其中对角块外的所有元素都是零（不同图中的节点不相连）。虽然看似浪费，但实际上这实际上增加了图的稀疏率，更好地利用稀疏矩阵操作。因此，对于许多图数据集，在处理单个图或图的小批量之间实际上没有真正的区别。

为了跟踪节点和图之间的对应关系，我们可以通过添加一个额外的向量 b $\sim (\sum_i n_i)$ 来增强表示，使得 b_i 是 [0, ..., $b-1$] 中的一个索引，用于标识 b 输入图中的一个 - 见图F.12.7。对于图分类，我们可以利用 b 对不同图对应的节点组分别进行池化。假设 H $\sim (n, c')$ 是 GCN 主干的输出，那么：

$$\text{scatter\_sum}(\mathbf{H}, \mathbf{b}) = \mathbf{Y} \sim (b, c') \tag{E.12.10}$$

称为散点求和操作，其定义如下：Y_i是H中所有行的和，其中$b_j = i$，如图F.12.8所示。可以定义其他类型的池化操作的类似操作，包括平均值和最大值。

作为一个单独的问题，有时我们可能有一个不适合内存的单个图：在这种情况下，应通过*sampling*个子图从原始图[HYL17]形成小批量。这是一个相对复杂的问题，超出了本章的范围。

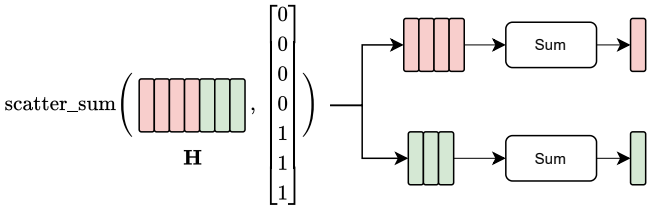


图 F.12.8：*Example of scattered sum on the graph of Figure F.12.7. In this example nodes (1,2,3,4) belong to graph 1, and nodes (5,6,7) to graph 2. After pooling, we obtain a pooled representation for each of the two graphs.*

12.3 超越图卷积层

以GC层为模板，我们现在概述一些扩展，无论是关于适应性还是可以处理图特征。最后，我们讨论图变换器，这是一种不同的层家族，其中图嵌入到一个结构嵌入中，该嵌入被加到节点特征上。

12.3.1 图注意力层

GC层的一个问题是，用于汇总邻域贡献的权重是固定的，并由邻接矩阵（或适当的归一化变体）给出。这相当于假设，除了连接的相对数量外，所有邻居都同样重要。节点主要与相似节点相连的图称为同质图：经验上，同质性是图卷积层[LLLG22]性能的良好预测指标。

并非所有图都是同质性的：例如，在交友网络中，大多数人将与异性的人建立联系。因此，在这些情况下，我们需要能够调整节点间权重的技术。

对于足够小的图，我们可以让权重矩阵A的非零元素通过梯度下降从起始值进行适应。然而，在这种情况下，可训练参数的数量与节点数量的平方成正比，并且此解决方案不适用于包含多个图的场景。如果我们假设一条边仅依赖于连接的两个节点的特征，我们可以通过类似于注意力的操作来推广GC层：

$$\mathbf{h}_i = \phi\left(\sum_{j\in\mathcal{N}(i)} \text{softmax}(\alpha(\mathbf{x}_i, \mathbf{x}_j))\mathbf{W}^\top \mathbf{x}_j\right)$$

在 α 是具有两个输入和标量输出的某些通用 MLP 块，并且对每个节点，应用 softmax 到 α 关于 $\mathcal{N}(i)$ 的输出集，以独立于邻域大小来归一化权重。由于与注意力层的相似性，这些被称为图注意力（GAT）层 [VCC+18]。从整个图的角度来看，这非常类似于 MHA 层，其中注意力操作仅限于具有连接它们的边的节点。

α的选择相对自由。而不是点积，原始GAT公式考虑了对特征拼接应用MLP：

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \text{LeakyReLU}(\mathbf{a}^\top \left[\mathbf{V}\mathbf{x}_i \parallel \mathbf{V}\mathbf{x}_j\right])$$

在V和a可训练的情况下。后来发现这过于限制性，因为在元素之间的顺序不依赖于中心节点[BAY22]。一个更宽松的变体，称为GATv2 [BAY22]，如下获得：

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{V}\left[\mathbf{x}_i \parallel \mathbf{x}_j\right])$$

GAT 和 GATv2 现在都是非常受欢迎的基线。

12.3.2 消息传递神经网络

假设我们有可用的额外边特征 e_{ij}，例如，在一个分子数据集中，我们可能知道每个分子键的类型的一个 one-hot 编码表示。我们可以通过适当修改注意力函数来泛化 GAT 层以包括这些特征：

$$\alpha(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{a}^\top \text{LeakyReLU}(\mathbf{V}\left[\mathbf{x}_i \parallel \mathbf{x}_j \parallel \mathbf{e}_{ij}\right])$$

我们可以通过抽象出它们的基本组件来进一步泛化到目前为止所看到的全部层（GC，GAT，GATv2，具有边缘特征的GAT）。考虑一个非常一般的层公式：

$$\mathbf{h}_i = \psi\left(\mathbf{x}_i, \text{Aggr}\left(\left\{M(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{ij})\right\}_{\mathcal{N}(i)}\right)\right) \qquad \text{(E.12.11)}$$

其中：

1. M 在节点 i 和节点 j 之间的边相对于构建一个特征向量（我们称之为消息）。与 GC 和 GAT 层相反，我们不是

限制消息为标量值。

2. Aggr 是一个通用的排列不变函数（例如，求和）用于聚合与节点 i 相连的所有节点的消息。3. ψ 是一个最终块，它将聚合的消息与节点特征 x_i 结合。这样，具有相同邻居的两个节点仍然可以区分。

作为一个例子，在GC层中，消息构建为$M(_, \mathrm{x}_j, _) = A_{ij}\mathrm{W}^\top \mathrm{x}_j$，聚合是一个简单的求和，以及$\psi(_, \mathrm{x}) = \phi(\mathrm{x})$。通用层（E.12.11）在[GSR+17]中引入，命名为消息传递层，并且已经成为对图上操作（并泛化）层进行分类（和泛化）的一种非常流行的方法[Vel22]。

让我们考虑一些使用此消息传递框架的例子。首先，我们可能希望在消息传递阶段更重视中心节点。我们可以通过修改 ψ 函数来实现这一点：

$$\psi(\mathbf{x}, \mathbf{m}) = \phi(\mathbf{V}\mathbf{x} + \mathbf{m})$$

V是一个通用的可训练矩阵（这在[MRF+19]中引入，并在PyTorch Geometric中作为GraphConv[4]层流行）。其次，假设节点具有更复杂的特征，例如每个节点的时间序列（例如，一组分布式传感器）。请注意，在消息传递框架中，节点级别的

[4] https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.GraphConv.html

操作与消息的聚合和处理方式解耦。用 x_i 表示节点 i 的时间序列，我们可以通过简单地修改消息函数并添加一个处理时间序列的层（例如 Conv1d 层）来泛化 GC 层：

$$h_i = \sum_{j \in \mathcal{N}(i)} A_{ij} \text{Conv1d}(x_i)$$

这是一个空间时间GC层[YYZ17]的示例。此外，到目前为止，我们假设只有节点特征应该更新。然而，通过额外的边更新层也可以轻松更新边特征：

$$\mathbf{e}_{ij} \leftarrow \text{MLP}(\mathbf{e}_{ij}, \mathbf{h}_i, \mathbf{h}_j)$$

这也可以被视为一种消息传递迭代，其中边从其邻居（两个连接的节点）聚合消息。这种推理方法允许进一步将这些层推广到考虑更广泛的邻域和图特征 [BHB+18]。

这是一个非常简短的概述，概述了许多可能的消息传递变体。由于篇幅限制，我们无法详细涵盖许多主题：在这些主题中，我们特别指出为高阶图（其中边连接多个节点）构建MP层（[CPPM22]）和MP层用于点云数据，其中我们感兴趣的是满足额外的对称性（旋转和平移对称性）[SHW21，EHB23]。

12.3.3 图变换器

我们已经看到两种利用图结构的技术：第一种是在网络输出中添加一个正则化项，迫使网络输出相对于图是平滑的；第二种是将图的运算约束为遵循图的连通性。特别是，在GAT层中，我们通过适当掩码成对比较使用了标准的注意力操作。然而，我们也在前一章中看到，由于它们提供了一个完全与数据类型无关的架构，因此变压器变得流行。我们能否设计出图变压器 [MGMR24] 的等效物?

回忆一下，构建transformer的两个基本步骤是输入数据的分词和位置嵌入的定义。对于图来说，分词很简单：例如，我们可以将每个节点视为一个标记，或者（如果给出了边特征）在将它们嵌入共享空间后，每个节点和每条边作为单独的标记。现在让我们忽略边特征。考虑以下通用架构，它以节点特征作为输入：

$$\mathrm{H} = \text{变换器}(\mathrm{X})$$

这是排列等变但完全与连通性无关。我们可以通过增强节点特征以一些基于图的特征来部分解决这个问题，例如节点的度或到某些预选节点（锚点）[RGD+22，MGMR24] 的最短路径距离。然而，更一般地，我们可以考虑将图连通性嵌入到我们所说的结构嵌入中：

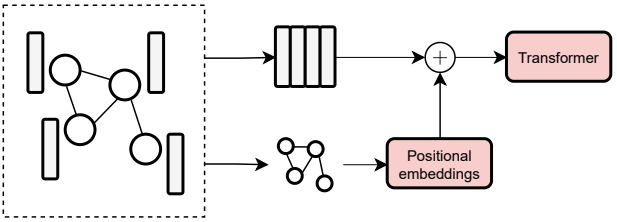


图 F.12.9：*General idea of a graph transformer: the connectivity is embedded into a set of positional embeddings, which are added to the collected features. The result is then processed by a standard transformer network.*

$$\mathrm{H} = \text{变换器}(\mathrm{X}\ \text{嵌入}+\mathrm{A}))$$

每行Embedding(A)提供相对于单个节点的图连接的向量嵌入，忽略所有特征（见图F.12.9）。幸运的是，将图结构嵌入到向量空间是一个广泛的领域。例如，我们在此描述了一种基于随机游走[DLL+22]的常见嵌入过程。回想以下矩阵：

$$\mathbf{R} = \mathbf{A}\mathbf{D}^{-1}$$

可以解释为“随机游走”，其中 R_{ij} 是从节点 i 移动到节点 j 的概率。我们可以多次迭代随机游走，用户事先固定一个 k 集合：

$$\mathbf{R}, \mathbf{R}^2, \ldots, \mathbf{R}^k$$

随机游走嵌入是通过收集节点返回自身的所有游走概率，并将它们投影到一个固定维度的嵌入中构建的：

$$\text{Embedding}(\mathbf{A}) = \begin{bmatrix} \text{diag}(\mathbf{R}) \\ \text{diag}(\mathbf{R}^2) \\ \vdots \\ \text{diag}(\mathbf{R}^k) \end{bmatrix} \mathbf{W}$$

在图结构的具体条件下，这可以证明为每个节点 [DLL+22] 提供唯一的表示。通过考虑拉普拉斯矩阵 [LRZ+23] 的特征分解，可以获得其他类型的嵌入。关于图变换器的更全面阐述，我们参考 [MGMR24]。构建图变换器为图域中的 GPT 类基础模型的可能性打开了大门，并且可以将基于图的数据作为额外模式添加到现有的语言模型 [MCT+]。

从理论到实践

高效处理图数据需要扩展基本框架，因为本章中描述的问题（例如，稀疏性）。常见的库包括用于 {v*} 的 PyTorch Geometric。

PyTorch和Jraph for JAX。两者都有丰富的教程集，例如关于小引用网络中的节点分类的教程集.[5]

如果您在第11章实现了视觉Transformer，我建议一个有趣的练习，它具有（主要是）教学价值，如图F.12.10所示。假设我们将图像分块化，但不是添加位置嵌入，

[5]Recommended example in PyTorch Geometric: https://pytorch-geometric.readthedocs.io/en/latest/get_started/introduction.html.

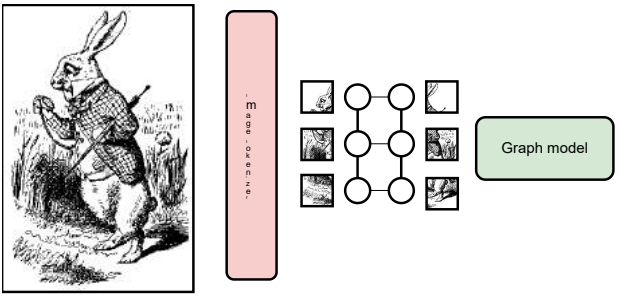


图 F.12.10：*A GNN for computer vision: the image is tokenized into patches, an adjacency matrix is built over the patches, and the two are propagated through a graph model.*

我们构造一个邻接矩阵 A ∼ (p， p) (，其中 p 是补丁的数量)，如下所示：

$$A_{ij} = \begin{cases} 1 \text{ 如果两个补丁在图像中共享一个边界 } 0 \text{ 否则} \\ \end{cases} \tag{E.12.12}$$

我们现在有一个图分类数据集，其中节点特征由补丁嵌入给出，邻接矩阵由（E.12.12）给出。因此，我们可以通过调整之前提到的教程中的GNN来执行图像分类。

13 | 循环模型

关于本章

Transformer模型在处理序列方面非常有效，但它们在序列长度上的二次复杂度限制了它们。一种可能性是用循环层来替换它们，处理序列中的每个元素只需要常数时间，无论其长度如何。在本章的最后，我们概述了几种循环模型及其特征。该领域在过去两年中发展非常迅速，我们以牺牲精度为代价提供了广泛的概述——参见[TCB+24]以获取最近的调查。

13.1 线性化注意力模型

13.1.1 替换点积

为了提供一些关于为什么循环神经网络（RNNs）可能有用的直观理解，我们从一个关注层（称为线性化关注层[KVPF20]）的推广开始，它可以写成递归形式。我们首先将SA层重写为

抽象形式具有通用的标量值注意力函数 {v*}
•,•) 而不是点积：

$$\mathbf{h}_i = \frac{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right) \mathbf{v}_j}{\sum_{j=1}^{n} \alpha\left(\mathbf{q}_i, \mathbf{k}_j\right)} \tag{E.13.1}$$

在标准SA中，对于α(x，y) = exp(x$^\top$y)。如果序列的元素必须按顺序处理（如在自回归生成中），(E.13.1)是不方便的，因为其成本随着序列长度的平方增长。即使使用KV缓存，内存仍然线性增长。相比之下，卷积层对每个要处理的元素具有固定的时间和内存成本，但如果一个标记在感受野之外，信息就会丢失。因此，我们希望有一个机制将序列的所有信息压缩成一个固定大小的输入（我们将其称为记忆或状态张量），这样在当前输入标记加上记忆上运行模型的成本就是常数。我们称这种形式的模型为循环模型。

首先，请注意任何非负的 α 都是一个有效的相似度函数。在机器学习中，这个要求等同于 α 被称为核函数 [HSS08]。许多这样的核函数可以写成广义的点积形式：

$$\alpha(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^\top \phi(\mathbf{y}) \tag{E.13.2}$$

对于某些函数 $\phi : \mathbb{R}^c \to \mathbb{R}^e$ 执行特征扩展（这是支持向量机等方法的基石，但详细讨论将超出本书的范围——此外，在本节之后也不再需要）。

内核函数

作为一个核函数的例子，多项式核函数 $\alpha(\mathrm{x}, \mathrm{y}) = (1 + \mathrm{x}^\top \mathrm{y})^d$ 可以在 $\phi($ 的条件下重写为 (E.13.2)。

•) 显式计算其输入的所有多项式，最高到阶数 d [HSS08]。一些核函数对应于无限维展开（例如高斯核），在这种情况下，(E.13.2) 仍然可以通过近似核展开来恢复，例如使用随机傅里叶特征 [SW17]。

基于（E.13.2），我们可以将（E.13.1）重写为：

$$\mathbf{h}_i = \frac{\sum_{j=1}^n \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\sum_{j=1}^n \phi(\mathbf{q}_i)^\top \phi(\mathbf{k}_j)}$$

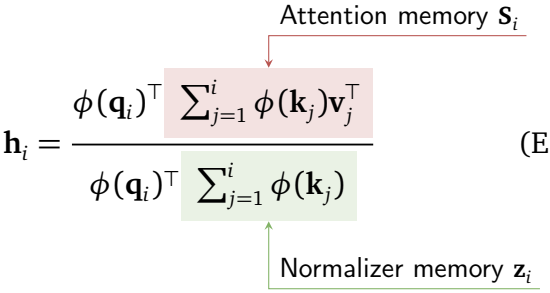
在v_j上添加了转置操作以保持维度一致性。因为$\phi(\mathrm{q}_i)$不依赖于j，我们可以将其移出求和符号，得到：

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \sum_{j=1}^n \phi(\mathbf{k}_j)\mathbf{v}_j^\top}{\phi(\mathbf{q}_i)^\top \sum_{j=1}^n \phi(\mathbf{k}_j)} \tag{E.13.3}$$

这被称为线性化注意力模型 [KVPF20]。对所有标记计算（E.13.3）的复杂度为 $\mathcal{O}(n(e^2 + ev))$，它在序列长度上是线性的，并且当 $n < e^2$ 时具有优势。ϕ 可以自由选择，例如，在 [KVPF20] 中，他们考虑了二次特征扩展，甚至更简单的 $\phi(\mathrm{x}) = \mathrm{ELU}(\mathrm{x}) + 1$ 用于短序列。

13.1.2 一个递归公式

我们现在将线性化注意力模型重写为循环形式，通过考虑层因果变体发生的情况。首先，我们通过仅对过去输入元素求和来修改（E.13.3），使其具有因果性：



(E.13.4)

这是我们的第一个循环层的例子。为了理解这个名字，我们注意到注意力和归一化器记忆可以递归地表示为：

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top \tag{E.13.5}$$

$$\mathbf{z}_i = \mathbf{z}_{i-1} + \phi(\mathbf{k}_i) \tag{E.13.6}$$

其中递归的基本情况由它们的初始化给出：

$$\mathbf{S}_0 = \mathbf{0} \tag{E.13.7}$$

$$\mathbf{z}_0 = \mathbf{0} \tag{E.13.8}$$

输出由以下给出：{v*}

$$\mathbf{h}_i = \frac{\phi(\mathbf{q}_i)^\top \mathbf{S}_i}{\phi(\mathbf{q}_i)^\top \mathbf{z}_i} \tag{E.13.9}$$

方程式（E.13.5）至（E.13.9）在自回归场景中特别有趣：对于要生成的任何新标记，我们更新两个记忆状态（方程式

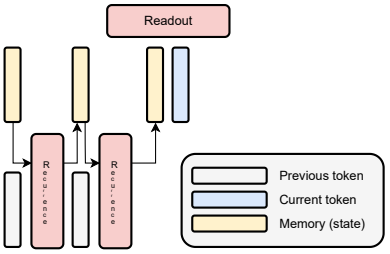


图 F.13.1：
Overview of a recurrent layer: past tokens are shown in gray, current input token in blue, the memory state in yellow.

(E.13.5) 和 (E.13.6))，我们使用这些更新后的状态来计算 i -th 元素的输出。重要的是，生成新标记的总计算量是恒定的，内存成本也是固定的，因为之前的内存 S_{i-1} 和 z_{i-1} 可以被丢弃。我们可以在层的两种公式之间交替：我们可以使用向量化变体进行训练（在 GPU 上的高效实现）和循环公式进行推理。

13.2 经典循环层

13.2.1 通用公式

让我们现在抽象出循环层的核心组件，以上一节为参考。首先，我们需要一个固定大小的状态，用于压缩直到序列的第 i 个元素的所有有用信息。我们将其通用地表示为 s_i ，并且不失一般性，我们假设从现在起它是一个单个向量。其次，我们需要一个转换函数（递归），它根据前一个值和当前标记的值更新状态向量，我们将其表示为 $f(\mathrm{s}_{i-1}, \mathrm{x}_i)$。第三，我们需要我们称之为读取函数，它为序列的第 i 个元素提供输出。我们将其表示为 $g(\mathrm{s}_i, \mathrm{x}_i)$。另见图 F.13.1 以获得可视化。

定义 D.13.1（循环层）

Given a sequence of tokens $x1, x_2, \ldots$ *, a generic recurrent layer can be written as:*

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{x}_i) \tag{E.13.10}$$

$$\mathbf{h}_i = g(\mathbf{s}_i, \mathbf{x}_i) \tag{E.13.11}$$

where the ***state vector*** $s_i \sim (e)$ *is initialized as zero by convention,* $s_0 = 0$*. The size of the state vector, e, and the size of the output vector* $h_i \sim (o)$ *are hyper-parameters. We call* f *the* ***state transition*** *function and* g *the* ***readout*** *function.*

在这个格式中，一个循环层代表一个离散时间、输入驱动的动力系统，并且它按定义是一个因果层。在控制工程中，这也被称为状态空间模型。对于因果性不必要的任务，也可以定义双向层 [SP97]。在一个双向层中，我们初始化两个循环层（具有不同的参数），其中一个从左到右处理序列，另一个从右到左。然后，将它们的输出状态连接起来以提供最终输出。

循环神经网络（RNNs）可以通过在更新序列h_1，h_2，...，h_n [PGCB14]上堆叠多个循环层来构建。有趣的是，循环层对序列长度没有要求，理论上可以是无限的。因此，具有无限精度或增长架构的RNN可以证明是图灵完备的[CS21]。

隐式层

如果我们对一个 *single* 令牌 x 应用循环层会发生什么?

$$\mathbf{s}_i = f(\mathbf{s}_{i-1}, \mathbf{x}) \tag{E.13.12}$$

如果我们从已知的初始化s_0开始运行状态转移多次，这类似于具有多个层（每个转移一个）且共享相同参数的模型。假设我们运行(E.13.12) *infinite*次。如果动态系统具有稳定的吸引子，输出将由固定点方程定义：

$$\mathbf{s} = f(\mathbf{s}, \mathbf{x}) \tag{E.13.13}$$

如果我们把（E.13.13）作为层的定义，我们得到所谓的隐层 [BKK19]。通过使用固定点方程的快速求解器和利用隐函数定理进行反向传播计算，可以实现隐层的实现 [BKK19]。隐式图层也可以通过将每个扩散操作运行到稳定状态来定义 [GMS05，SGT+08]。

13.2.2 “香草”循环层

历史上，通过将两个全连接层视为转换和读出函数来实例化循环层：

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \phi(\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i) \tag{E.13.14}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.15}$$

在始终忽略偏差以简化的情况下，我们拥有四个可训练矩阵 A ∼ (e、e)、B ∼ (e、c)、C ∼ (o、e) 和 D ∼ (o、c)，其中 c 是输入维度

(每个标记的大小)。这种形式的层有时被统称为“*recurrent layer*”，“*vanilla recurrent layer*”，或Elman循环层。当两个矩阵A和B未被训练，并且只有单层时，这些模型被称为回声状态网络（ESNs）或水库计算机[LJ09]。ESNs可以作为时间序列预测的有力基线，尤其是在未训练的矩阵（水库）以适当方式初始化时[GBGB21]。

尽管它们具有历史意义，但这种形式的层在训练上极其低效（且困难）。为了了解这一点，请注意，由于其设计，无法有效地并行化对序列元素的计算，如Box C.13.1所示。因此，我们需要求助于迭代（for循环）实现，即使是高度定制的CUDA实现[1]也比大多数替代序列层要慢。

另一个问题源于层计算中涉及的梯度。考虑一个只有过渡函数的简化情况。我们可以展开完整计算如下：

$$
\begin{aligned}
\mathbf{s}_1 &= f(\mathbf{s}_0, \mathbf{x}_1) \\
\mathbf{s}_2 &= f(\mathbf{s}_1, \mathbf{x}_2) \\
&\vdots \\
\mathbf{s}_n &= f(\mathbf{s}_{n-1}, \mathbf{x}_n)
\end{aligned}
$$

[1] https://docs.nvidia.com/deeplearning/performance/dl-performance-recurrent/index.html

```
# Input tensor
x = torch.randn(batch_size,
                sequence_length,
                features)

# State tensor
s = torch.zeros(batch_size,
                state_size)

# State update
state_update = nn.RNNCell(features,
                          state_size)
for i in range(x.shape[1]):
  s = state_update(x[:, i, :], s)
```

盒子 C.13.1: *Vanilla recurrence in PyTorch. It is impossible to parallelize the for-loop because of the dependencies in the recurrence. In PyTorch, the state update is called a* ***recurrent cell****, while the recurrent layers, such as* torch.nn.RNN*, wrap a cell and perform the complete for-loop.*

这与具有 n 层的模型类似，但参数在层之间是共享的（相同的）。以下我们关注 $\partial_{\mathbf{A}}\mathbf{s}_n$ (相对于 A) 的权重雅可比，但类似的考虑也适用于所有梯度。让我们定义以下累积乘积：

$$\widetilde{\mathbf{s}}_i = \prod_{j=i+1}^{n} \partial_{\mathbf{s}_{j-1}} f(\mathbf{s}_{j-1}, \mathbf{x}_j) \tag{E.13.16}$$

这表示从序列末尾向元素 i 的过渡函数的梯度，如图 F.13.2 所示。由于权重共享，我们寻找的梯度对于序列中的每个涉及这些累积乘积的元素都有一个单独的项：

$$\partial_{\mathbf{A}}\mathbf{s}_n = \underbrace{\partial_{\mathbf{A}} f(\mathbf{s}_{n-1}, \mathbf{x}_n)}_{\text{Gradient from element } n} + \sum_{i=1}^{n-1} \widetilde{\mathbf{s}}_i \left[\partial_{\mathbf{A}} f(\mathbf{s}_{i-1}, \mathbf{x}_i)\right] \tag{E.13.17}$$

Gradient from element i

第一项对应于一个“标准”权重雅可比矩阵，描述了A对序列最后一个元素的影响。求和项是额外的贡献，每个序列元素一个，这些贡献由在序列本身上计算的链式输入雅可比矩阵加权。

以这种形式编写，反向模式自动微分也称为时间反向传播（BPTT），它可能在梯度下降过程中成为不稳定或梯度问题的强大来源。为了看到这一点，请注意，(E.13.17)中的内积中的每个输入雅可比矩阵都涉及与导数的乘法。

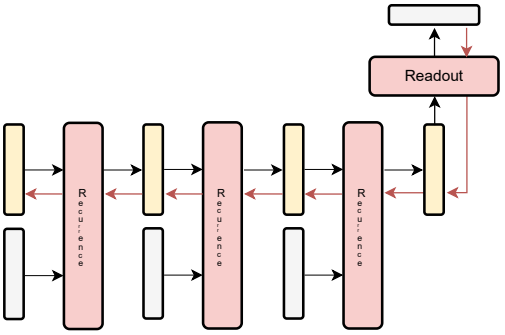


图 F.13.2：*Backward pass for a recurrent layer: the adjoint values have to be propagated through all the transition steps. Each state then contributes a single term to the full gradient of the parameters.*

激活函数 ϕ。一些关于梯度消失和爆炸的最早分析是在这个背景下进行的 [Hoc98]。对于长序列，只有当转换矩阵的特征值得到适当约束时，层的稳定性才能得到保证 [GM17]。层归一化最初也是为了稳定RNN的训练而开发的，通过计算状态序列的统计信息 [BKH16]。

一些技术已被开发出来，在循环层的环境中部分解决这些不稳定性。例如，(E.13.17)中的和可以被截断到给定的区间（截断BPTT），或者如果梯度超过预定义的上限，则可以对其进行阈值处理（剪裁梯度）。

13.2.3 门控循环网络

多年来，提出了多种香草层的变体以改进其性能。在本节中，我们关注一类流行的此类模型，称为门控RNN。

RNNs的一个问题是每次转换时整个状态都会被覆盖，这在(E.13.17)中的部分乘积中得到了反映。然而，我们可以假设对于许多序列，这些转换中只有少数元素是重要的：以音频信号为例，空白区域或无信息区域是典型的。在这些情况下，我们可能对*sparsifying*转换感兴趣（类似于大多数注意力权重倾向于接近零），因此将$\tilde{\mathrm{s}}_i$中的大多数元素设置为1。这可以通过添加专门的门控层来实现。

我们考虑最简单的门控RNN形式，称为轻量级门控循环单元（Li-GRU，[RBOB18]），具有单个门。对于我们的目的，门控函数只是一个输出范围在$[0, 1]$的层，可以用来“屏蔽”输入。例如，可以通过具有sigmoid激活函数的全连接层获得状态上的门控：

$$\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) = \sigma\left(\mathbf{V}\mathbf{s}_{i-1} + \mathbf{U}\mathbf{x}_i\right)$$

在 V 和 U 与 A 和 B 形状相似的情况下，我们可以这样解释：如果 $\gamma_i \approx 0$，则状态的第 i 个特征应保持不变，而如果 $\gamma_1 \approx 1$，则应将其更新的值作为输出传播。因此，我们可以通过适当掩码新旧值来重写转换函数：

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \overbrace{\gamma(\mathbf{s}_{i-1}, \mathbf{x}_i) \odot \phi\left(\mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i\right)}^{\text{New values}} + \underbrace{(1 - \gamma(\mathbf{s}_{i-1}, \mathbf{x}_i)) \odot \mathbf{s}_{i-1}}_{\text{Old values}}$$

这可以被视为一个对只有二进制值的“真实”门的软（可微）逼近，或者视为原始层和跳跃连接的凸组合。我们可以通过向损失中添加一个额外的正则化器来理论上控制这种逼近的好坏，该正则化器约束门的输出尽可能接近0或1。

其他门控循环层可以通过向此设计添加额外的门来获得：原始的门控循环单元（GRU）向层 [CVMG+14] 添加了一个所谓的“重置门”，而长短期记忆单元（LSTMs）有一个第三个“忘记门” [HS97]。LSTMs 是文献中首次引入的门控变体，并且长期以来一直是处理序列最成功的深度架构 [Sch15]。因此，对 LSTM 模型的研究仍然非常活跃 [BPS+24]。

13.3 结构化状态空间模型

13.3.1 线性循环层

我们现在考虑一类简化的循环层，其中我们在转换函数中移除了中间的非线性：

$$f(\mathbf{s}_{i-1}, \mathbf{x}_i) = \mathbf{A}\mathbf{s}_{i-1} + \mathbf{B}\mathbf{x}_i \tag{E.13.18}$$

$$g(\mathbf{s}_i, \mathbf{x}_i) = \mathbf{C}\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.19}$$

以这种形式编写，（E.13.18）-（E.13.19）被称为状态空间模型（SSM）。直观上，SSM层是“较少的”[2]

[2]Confusingly, any recurrent layer in the form (E.13.10)-(E.13.11) is an SSM, but in the neural network's literature the term SSM has come

比标准循环层“表达性”更强（因为缺乏非线性）。然而，这可以通过在输出后添加激活函数或通过将这些层与按令牌的MLP [ODG+23]交织来恢复。

对此类模型的兴趣在2020年重新启动，当时[GDE+20]分析了一个理论构造，该构造可以有效地压缩一维输入序列中的矩阵A（E.13.18）。该结果被称为HiPPO（高阶多项式投影算子）矩阵。基于HiPPO理论构建的神经网络家族紧随其后，导致2021年[GGR22]的序列建模结构化状态空间（S4）层和2022年[SWL23]的简化S4模型（S5）。

由于其根植于HiPPO理论，所提出的SSM层直到S4被视为一个堆叠的1D模型，每个输入通道一个，过渡矩阵初始化为HiPPO矩阵。相比之下，S5引入了(E.13.18)-(E.13.19)形式的标准多输入、多输出模型，这是我们在此描述的模型。特别是，我们关注一个称为线性循环单元（LRU）的简化变体，记作[OSG+23]。

这个公式具有许多有趣的性质，主要源于线性转换函数的结合性。为了看到这一点，我们首先注意到该递归有一个封闭形式的解：

$$\mathbf{s}_i = \sum_{j=1}^{i} \mathbf{A}^{i-j} \mathbf{B} \mathbf{x}_j \tag{E.13.20}$$

to be associated only with the linear variant. Sometimes we refer to them as **structured** SSMs because, as we will see, we need to properly constrain the transition matrix to make them effective.

我们可以从两个不同的角度看待这个求和。首先，我们可以将所有关于输入序列的系数聚合到一个秩为3的张量中：

$$K = \text{stack}\left(\mathbf{A}^{n-1}\mathbf{B}, \mathbf{A}^{n-2}\mathbf{B}, \ldots, \mathbf{AB}, \mathbf{B}\right)$$

我们可以通过一个与序列长度相等的滤波器大小的单维卷积（一个 *long* 卷积）来计算所有输出，该卷积在输入序列堆叠成一个矩阵 $\mathrm{X} \sim (n, c)$ 和预计算的核 K 之间进行：

$$\mathbf{S} = \text{Conv1D}(\mathbf{X}, K)$$

因此，SSM层可以解释为卷积 [GJG+21]。如果将转换矩阵应用于单个通道，可以通过在频域中操作来利用这一点以加快计算速度，例如在FlashConv实现中。[3]然而，通过利用一种称为关联（并行）扫描（或所有前缀和）的算法族，可以找到更有效的解决方案。

13.3.2 间奏：关联扫描

我们在看到它们在线性SSMs中的应用之前，先介绍它们的并行扫描的一般公式。考虑一个元素序列 $(x_1, x_2, \ldots, x_n)$，以及一个假设为二元的操作 $\star$（它作用于序列中的任意两个元素）和结合律。我们希望计算这个操作对序列的所有部分应用（使用不同的颜色以提高可读性）：$(x_1$
$x_2\, x_n)(x_1\, x_2\, x_n)(x_1\, x_2\, x_n)(x_1\, x_2\, x_n)(x_1\, x_2\, x_n)(x_1\, x_2\, x_n)

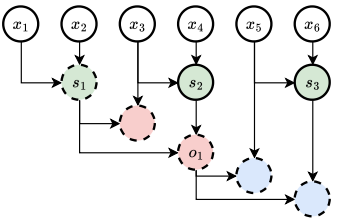


图 F.13.3：*Parallel scan on a sequence of six elements: circles of the same color can be computed in parallel; dashed circles are the outputs of the parallel scan.*

$$x_1,\ x_1 \star x_2,\ x_1 \star x_2 \star x_3,\ \ldots,\ x_1 \star x_2 \star \cdots \star x_n$$

这可以通过一个迭代算法轻易完成，该算法逐个计算元素，每次迭代添加一个元素（这对应于标准循环层如何计算）。然而，我们可以通过利用操作符 $\star$ [Ble90] 的结合性来设计一个高效的 *parallel* 算法。关键直觉是，可以并行计算多个元素对，然后递归聚合。

作为一个简单的例子，考虑一个包含6个元素的序列 x_1，x_2，x_3，x_4，x_5，x_6 (。在 [SWL23]) 中可以找到一个应用于 SSMs 的深入示例。我们将用 $\hat{x}_i$ 表示我们想要计算的 i-th 前缀。整个过程在图 F.13.3 中以示意图的形式展示。我们首先将相邻的值配对汇总，如下所示：

$$\begin{aligned} s_1 &= x_1 \star x_2 \rightarrow \hat{x}_2 \\ s_2 &= x_3 \star x_4 \\ s_3 &= x_5 \star x_6 \end{aligned}$$

我们在其中使用箭头表示算法的输出。我们现在进行第二层聚合：

$$s_1 \star x_3 \rightarrow \hat{x}_3$$
$$o_1 = s_1 \star s_2 \rightarrow \hat{x}_4$$

并且最后：

$$o_1 \star x_5 \rightarrow \hat{x}_5$$
$$o_1 \star s_3 \rightarrow \hat{x}_6$$

虽然这看起来很奇怪（我们做了7步而不是5步），但如果我们有3个独立的线程，这三个计算块可以简单地并行化。一般来说，通过平衡地组织计算集，我们能够在 $\mathcal{O}(T \log n)$ 的时间内计算并行扫描，其中 T 是二进制运算符 $\star$ 的成本。JAX 中的关联扫描函数是一个实现示例.[4]

它很容易证明线性SSM中的转移函数是一个所有前缀和问题的例子。我们定义我们的序列元素为对 $x_i = (\mathrm{A}, \mathrm{Bx}_i)$，二元运算符为：

$$(\mathbf{Z}, \mathbf{z}) \star (\mathbf{V}, \mathbf{v}) = (\mathbf{VZ}, \mathbf{Vz} + \mathbf{v})$$

$\star$ 的前缀由 [SWL23] 给出：

$$x_1 \star x_2 \star \ldots \star x_i = (\mathbf{A}^i, \mathbf{s}_i)$$

因此，运行并行扫描使我们得到A的幂作为输出的第一个元素，并将层的所有状态作为输出的第二个元素。复杂度

[4] https://jax.readthedocs.io/en/latest/_autosummary/jax.lax.associative_scan.html

此操作的复杂度上界由A^{i-1}A的复杂度决定，其缩放比例为$\mathcal{O}(n^3)$。为了使整个过程可行，我们可以约束A，使其幂可以更有效地计算。这是下一节的主题。

13.3.3 对角线SSMs

一种使先前想法可行的常见策略是与对角转移矩阵（或对角矩阵加上低秩项 [GGR22]）一起工作。在这种情况下，可以通过对角元素求幂在线性时间内轻松计算 A 的幂。此外，正如我们将看到的，使用对角矩阵可以使我们控制转移函数的动力学，以避免数值不稳定性。

特别地，如果一个方阵 A 可以找到另一个方阵（可逆）P 和一个对角矩阵 {v*}，使得：

$$\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{-1} \tag{E.13.21}$$

对角化矩阵在某种意义上比一般矩阵“更简单”，例如，如果存在这样的分解，则很容易证明幂也可以高效地计算，如下所示：

$$\mathbf{A}^i = \mathbf{P}\mathbf{\Lambda}^i\mathbf{P}^{-1}$$

假设转移矩阵是可对角化的。那么，我们可以将SSM重新写成具有对角转移矩阵的等价形式。我们首先通过替换

(E.13.21)纳入SSM的定义中，并在两边乘以$\mathbf{P}^{-1}$：

$$\underbrace{\mathbf{P}^{-1}\mathbf{s}_i}_{\text{新状态向量 } \bar{\mathbf{s}}_i} = \sum_{j=1}^{i} \mathbf{\Lambda}^{i-j} \underbrace{\mathbf{PB}}_{\text{New input-state matrix } \bar{\mathbf{B}}} \mathbf{x}_j$$

我们现在用新变量 $\bar{\mathbf{s}}$ 重新编写读出函数：

$$\mathbf{y}_i = \underbrace{\mathbf{CP}}_{\text{New readout matrix } \bar{\mathbf{C}}} \bar{\mathbf{s}}_i + \mathbf{D}\mathbf{x}_i$$

将一切汇总：

$$\bar{\mathbf{s}}_i = \mathbf{\Lambda}\bar{\mathbf{s}}_{i-1} + \bar{\mathbf{B}}\mathbf{x}_i \tag{E.13.22}$$

$$\mathbf{y}_i = \bar{\mathbf{C}}\bar{\mathbf{s}}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.23}$$

因此，每当存在A的对角化时，我们总能将SSM重写为具有对角转换矩阵的等价形式。在这种情况下，我们可以直接训练四个矩阵 $\mathbf{\Lambda} = \text{diag}(\lambda)$，$\lambda \sim (e)$，$\bar{\mathrm{B}} \sim (e, c)$，$\bar{\mathrm{C}} \sim (o, e)$ 和 $\mathrm{D} \sim (o, c)$，其中对角矩阵由一个维度为 e 的单个向量参数化。

并非所有矩阵都可以对角化。然而，如果允许矩阵 P 和 $\mathbf{\Lambda}$ 具有复数值的项 [OSG+23]，则总能找到近似对角化。必须小心地参数化对角线上的值，以确保转移矩阵的特征值在绝对值上保持 < 1，以避免发散动力学。我们参考 [OSG+23]。

描述这两个点，并对产生的LRU层进行完整分析。

13.4 其他变体

平衡卷积、递归和注意力的不同优势是一个活跃的研究课题。为了结束本书，我们列出了一些最近在文献中引入的递归层（或可以解释为递归的层）。

13.4.1 无注意力变换器

线性化变压器模型（第13.1.1节）的一个问题是特征维度 {v*} 中的二次复杂度。无注意力变压器（ATF）被引入作为一种基本注意力层的变体，它在序列长度和特征数量 [ZTS+21] 上都是线性的。

核心思想是用更简单的 *multiplicative interaction* (逐元素) 替换键、查询和值之间的点积交互：

$$\mathbf{h}_i = \sigma(\mathbf{q}_i) \odot \frac{\sum_j \exp\left(\mathbf{k}_j\right) \odot \mathbf{v}_j}{\sum_j \exp\left(\mathbf{k}_j\right)} \tag{E.13.24}$$

这与自注意力层类似，但我们用逐元素（Hadamard）乘法替换了所有点积。它也受到了线性化注意力层的启发，其中查询仅用作全局调制因子，在这种情况下，在sigmoid操作后对其进行归一化。实际上，我们可以通过利用我们只执行的事实来重写（E.13.24）以恢复标准注意力公式，即对单维z（

元素级操作）：

$$h_{iz} = \frac{\sigma(q_{iz}) \sum_j \exp(k_{jz})}{\sum_j \exp(k_{jz})} v_{jz}$$

因此，ATF层可以被重新解释为通道维度的注意力，从每个通道的角度来看，我们可以将其重写为对序列元素的注意力操作。为了增加灵活性，[ZTS+21]还考虑了添加相对嵌入$W \sim (m, m)$ (其中m是序列允许的最大长度)：

$$\mathbf{h}_i = \sigma(\mathbf{q}_i) \odot \frac{\sum_j \exp\left(\mathbf{k}_j + W_{ij}\right) \odot \mathbf{v}_j}{\sum_j \exp\left(\mathbf{k}_j + W_{ij}\right)} \tag{E.13.25}$$

相对嵌入也可以通过低秩分解来训练，以减少参数数量。参见 [ZTS+21]，以及基本ATF层的其他变体（例如，与卷积操作混合）。我们还可以通过适当限制求和来将（E.13.24）转换为因果（递归）变体。

13.4.2 接收加权关键值（RWKV）模型

RWKV模型[PAA+23]通过引入一些额外的架构修改扩展了ATF层。在撰写本文时，这是唯一与最大规模变换器相匹配的预训练RNN之一，因此我们对其进行了更详细的描述。首先，通过考虑单个向量$w \sim (e)$简化了相对嵌入。

为每个偏移量进行缩放：

$$w_{ij} = -(i-j)\mathbf{w}$$

此外，实验表明，对于当前元素，使用单独的偏移量u（而不是w{v*}）是有益的。用因果形式表示，这给出：

$$\mathbf{h}_i = \mathbf{W}_o\left(\sigma(\mathbf{q}_i)\odot\frac{\sum_{j=1}^{i-1}\exp\left(\mathbf{k}_j + w_{ij}\right)\odot\mathbf{v}_j + \exp\left(\mathbf{k}_i+\mathbf{u}\right)\odot\mathbf{v}_i}{\sum_{j=1}^{i-1}\exp\left(\mathbf{k}_j + w_{ij}\right) + \exp\left(\mathbf{k}_i+\mathbf{u}\right)}\right)$$

在红色中突出显示与基本ATF层之间的差异。查询称为[PAA+23]中的接收度，并在末尾添加了一个额外的输出投影W_o。其次，RWKV模型通过一个不同的*gated*分词块修改了变压器块中的标准MLP。对于给定的输入标记x，这可以写成：

$$\mathbf{y} = \sigma(\mathbf{W}_1\mathbf{x})\odot\mathbf{W}_2\max(0,\mathbf{W}_3\mathbf{x})^2 \tag{E.13.26}$$

在 W_1、W_2 和 W_3 是可训练参数的地方。这是一个标准的MLP，除了最左边的门和使用了平方 ReLU。作为最后的修改，第一个块中的所有三个投影（以及 (13.4.2) 中 x 的两次出现）都被替换为 x_i 和 x_{i-1} 的凸组合，以提高性能，这被称为 *token shift*。

13.4.3 选择性状态空间模型

我们已经看到三种循环模型：标准循环层（及其门控版本）、线性化注意力层和结构化状态空间模型。

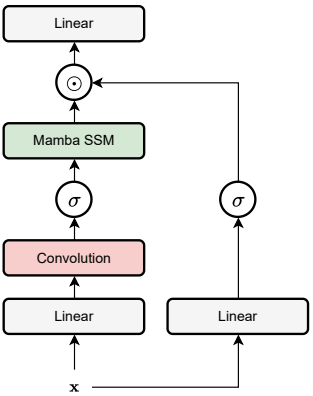


图 F.13.4：*Mamba block (residual connections around the block and normalization are not shown). σ is the sigmoid function. Adapted from [GD23].*

尽管它们看起来不同，但从一类模型移动到另一类模型相对容易。为了看到这一点，让我们考虑一个线性化的注意力层，其中我们忽略分母：

$$\mathbf{S}_i = \mathbf{S}_{i-1} + \phi(\mathbf{k}_i)\mathbf{v}_i^\top \tag{E.13.27}$$

$$\mathbf{h}_i = \phi(\mathbf{q}_i)^\top \mathbf{S}_i \tag{E.13.28}$$

除了矩阵值状态之外，我们发现这具有SSM层的结构，除了某些矩阵（例如，$\mathrm{C} = \phi(\mathrm{q}_i)^\top$）不是固定的，而是依赖于特定的输入标记。从动态系统的角度来看，我们说标准SSM描述 *time-invariant* 系统，而（E.13.27）-（E.13.28）描述一个 *time-varying* 系统。这激发了另一类SSM层，其矩阵不受时间不变性的约束，被称为选择性SSM。这些模型中的大多数都利用了在层计算之前将输入投影多次的注意力层思想。

作为一个例子，我们在此关注所谓的Mamba层 [GD23]，在撰写本文时，它是少数几个扩展到与SSM层性能相匹配的层之一。

Transformer模型在非常大的上下文和参数数量。首先，为了使SSM层具有时变性，其矩阵的一部分被设置为输入相关：[5]

$$\mathbf{s}_i = A(\mathbf{x}_i)\mathbf{s}_{i-1} + B(\mathbf{x}_i)\mathbf{x}_i \tag{E.13.29}$$

$$\mathbf{h}_i = C(\mathbf{x}_i)\mathbf{s}_i + \mathbf{D}\mathbf{x}_i \tag{E.13.30}$$

在 $A(\text{处 } \bullet)$，$B(\bullet)$，和 $C(\bullet)$ 它们是输入标记的线性投影。为了实现这一点，该层独立应用于输入的每个通道，并将转换矩阵选为对角矩阵，这样SSM的所有矩阵都可以用一个值向量表示。该层失去了简单的并行扫描实现，并需要一个定制的硬件感知实现[GD23]。可以证明，Mamba SSM变体和几个其他SSM层是门控循环层[GJG+21、GD23]的退化情况。

为了使整体架构更简单，Mamba避免交替使用MLP和SSM，转而采用门控架构（类似于第11.3节中的门控注意力单元），其中使用MLP对SSM的输出进行加权。为了提高灵活性，还添加了一个深度卷积 - 见图F.13.4。

[5] The matrix $\mathbf{D}$ can be seen as a simple residual connection and it is left untouched. The original layer has a slightly different parameterization where $\mathbf{A} = \exp(\Delta\bar{\mathbf{A}})$, for some trainable $\bar{\mathbf{A}}$ and input-dependent scalar value Δ. This does not change our discussion.

再见（现在）

Alice在这可微分的奇妙世界中第一次旅行（目前）已经结束。我们只进行了一次非常广泛的游览，重点关注了层可以设计成和组合成现代可微分模型（即神经网络）的许多方式。

有许多我们只是简要讨论的话题，包括我们如何在实践中 *use* 这些模型：从微调到生成建模、持续学习、多模态、可解释性等等。

我们也避免了烦人的工程方面：
训练和部署大型模型是一项巨大的工程壮举，这需要分布式训练策略、快速编译器和DevOps技术等。[6] LLMs的出现为它们的应用开辟了新的途径，甚至不需要了解其内部工作原理，从提示

[6] As an example, see https://jax-ml.github.io/scaling-book/.

工程到模型链和代理行为。

这本书有一个配套网站，[7]，我希望在那里发布一些涉及这些主题的额外章节。如果时间允许，其中一些可能会被合并成一个新的卷。

我希望您喜欢这次旅程！对于对本书的评论、建议和反馈，请随时联系我。

[7] https://sscardapane.it/alice-book

概率论 {v*}

关于本章

机器学习涉及广泛的不确定性（如数据收集阶段），使概率的使用变得基本。在此，我们非正式地回顾与概率分布和概率密度相关的基本概念，这些概念有助于正文。本附录介绍了许多概念，但其中许多应该是熟悉的。有关机器学习和神经网络中概率的更深入阐述，请参阅[Bis06，BB23]。

A.1 概率的基本定律

考虑一个简单的彩票，你可以购买具有3种可能结果的彩票：“无中奖”、“小奖”和“大奖”。对于任何10张彩票，其中1张将获得大奖，3张将获得小奖，6张将无中奖。我们可以用描述三种事件相对频率的概率分布来表示这一点（我们假设无限

票务供应）：

$$p(w = \text{‘no win’}) = 6/10$$
$$p(w = \text{‘small win’}) = 3/10$$
$$p(w = \text{‘large win’}) = 1/10$$

等效地，我们可以将一个整数值 $w = \{1, 2, 3\}$ 与三个事件相关联，并写成 $p(w = 1) = 6/10, p(w = 2) = 3/10,$ 和 $p(w = 3) = 1/10$。我们称 w 为随机变量。在以下内容中，当可能时，我们总是用 $p(w)$ 代替 $p(w = i)$ 以提高可读性。概率分布的元素必须是正数，并且它们的总和必须为 1：

$$p(w) \geq 0, \sum_{w} p(w) = 1$$

所有此类向量的空间被称为概率单纯形。

> 记住我们使用 $\mathrm{p} \sim \Delta(n)$ 来表示属于概率单纯形的、大小为 n 的向量。

假设我们引入第二个随机变量 r，一个二元变量，描述票是否真实（1）或伪造（2）。伪造的票更有利可图，但总体上概率较低，如表T.A.1所示。

我们可以使用表中的数字来描述一个联合概率分布，描述两个随机变量共同取某个值的概率：

$$p(r = 2, w = 3) = 8/100$$

另一种方法是定义一个条件概率分布，例如回答问题“*what is the probability of a certain event given that another event has*

表 T.A.1: *Relative frequency of winning at an hypothetical lottery, in which tickets can be either real or fake, shown for a set of* 100 *tickets.*

	$r = 1$ (real ticket)	$r = 2$ (fake ticket)
$w = 1$ (no win)	58	2
$w = 2$ (small win)	27	3
$w = 3$ (large win)	2	8
Sum	87	13

occurred? “: {v*}

$$p(r = 1 \mid w = 3) = \frac{p(r = 1, w = 3)}{p(w = 3)} = 0.2$$

这是称为概率乘法法则。与之前一样，我们可以通过使用随机变量代替其值来使符号更加简洁：

$$p(r, w) = p(r \mid w)p(w) \tag{E.A.1}$$

如果 $p(r \mid w) = p(r)$ 我们有 $p(r, w) = p(r)p(w)$，我们说这两个变量是独立的。我们可以使用条件概率来对其中一个随机变量进行边缘化：

$$p(w) = \sum_r p(w, r) = \sum_r p(w \mid r)p(r) \tag{E.A.2}$$

这是概率的加法法则。乘法法则和加法法则是定义概率代数的基本公理。通过将它们结合起来，我们得到基本的贝叶斯法则：

$$p(r \mid w) = \frac{p(w \mid r)p(r)}{p(w)} = \frac{p(w \mid r)p(r)}{\sum_{r'} p(w \mid r')p(r')} \tag{E.A.3}$$

贝叶斯定理使我们能够“反转”条件分布，例如，通过知道两类中奖彩票的相对比例来计算中奖彩票是真实还是伪造的概率（试试看）。

A.2 实值分布

在实值情况下，定义 $p(x)$ 更为棘手，因为 x 可以取无限多种可能的值，根据定义，每种值的概率都是 0。然而，我们可以通过定义一个概率累积密度函数（CDF）来解决这个问题：

$$P(x) = \int_0^x p(t)dt$$

并且将概率密度函数 $p(x)$ 定义为其导数。我们忽略了与处理概率密度相关的大多数细微差别，这些差别最好在测度理论 [BR07] 的背景下解决。我们仅指出，在这种情况下，通过适当地将和替换为积分，乘积和求和规则仍然有效：

$$p(x, y) = p(x \mid y)p(y) \tag{E.A.4}$$

$$p(x) = \int_y p(x \mid y)p(y)dy \tag{E.A.5}$$

注意概率密度不限制小于一。

A.3 常见分布

前述随机变量是分类概率分布的例子，描述了变量可以取 k 个可能值之一的情况。我们可以通过定义 $\mathrm{p} \sim \Delta(k)$ 为概率向量，以及 $\mathrm{x} \sim \mathrm{Binary}(k)$ 为观察到的类别的单热编码来简洁地表示：Translated Text: 前述随机变量是分类概率分布的例子，描述了变量可以取 k 个可能值之一的情况。我们可以通过定义 $\mathrm{p} \sim \Delta(k)$ 为概率向量，以及 $\mathrm{x} \sim \mathrm{Binary}(k)$ 为观察到的类别的单热编码来简洁地表示：

$$p(\mathbf{x}) = \mathrm{Cat}(\mathbf{x}; \mathbf{p}) = \prod_i p_i^{x_i}$$

我们使用分号来区分分布的输入与其参数。如果 $k = 2$，我们可以用单个标量值 p 等价地重写分布。得到的分布称为伯努利分布：

$$p(x) = \mathrm{Bern}(x; p) = p^x (1-p)^{(1-x)}$$

在连续情况下，我们将反复处理高斯分布，表示为 $\mathcal{N}(x; \mu, \sigma^2)$，描述一个以 μ（均值）为中心的钟形概率分布，具有 σ^2（方差）的范围：

$$p(x) = \mathcal{N}(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

在均值零和单位方差的最简单情况下，$\mu = 0$，$\sigma^2 = 1$，这被称为正态分布。对于向量 x $\sim (k)$，通过考虑均值向量 $\mu \sim (k)$ 和协方差矩阵 $\Sigma \sim (k,\ k)$，得到高斯分布的多变量变体：

$$p(\mathbf{x}) = \mathcal{N}(\mathbf{x}; \mu, \Sigma) = (2\pi)^{-k/2} \det(\Sigma)^{-1/2} \exp\left((\mathbf{x}-\mu)^\top \Sigma^{-1} (\mathbf{x}-\mu)\right)$$

两个有趣的案例是具有对角协方差矩阵的高斯分布，以及更简单的各向同性高斯分布，其协方差对角线上的所有项都相同：

$$\Sigma = \sigma^2 \mathbf{I}$$

第一个可以可视化为轴对齐的椭球体，各向同性的一个为轴对齐的球体。

A.4 瞬时和期望值

在许多情况下，我们需要用一个或多个值来总结一个概率分布。有时有限数量的值就足够了：例如，对于一个分类分布，如果我们有对p的访问权限，或者对于一个高斯分布，如果我们有对μ和σ^2的访问权限，就可以完全描述分布本身。这些被称为充分统计量。

更一般地，对于任何给定的函数$f(x)$，我们可以定义其期望值为：

$$\mathbb{E}_{p(x)}[f(x)] = \sum_x f(x) p(x) \qquad \text{(E.A.6)}$$

在实值情况下，我们通过将和替换为积分得到相同的定义。特别地，当 $f(x) = x^p$ 我们有分布的（p 阶）矩，其中 $p = 1$ 被称为分布的均值：

$$\mathbb{E}_{p(x)}[x] = \sum_x x p(x)$$

尽管无法访问底层概率分布，我们可能仍想估计一些期望值。如果我们有从 $p(x)$ 中采样元素的方法，我们可以应用所谓的蒙特卡洛估计器：

$$\mathbb{E}_{p(x)}[f(x)] \approx \frac{1}{n} \sum_{x_i \sim p(x)} f(x_i) \tag{E.A.7}$$

在 n 控制估计质量的情况下，我们使用 $x_i \sim p(x)$ 表示从概率分布 $p(x)$ 中采样的操作。对于一阶矩，这回到了从多个测量值计算一个量平均值的非常熟悉的符号表示：

$$\mathbb{E}_{p(x)}[x] = \frac{1}{n} \sum_{x_i \sim p(x)} x_i$$

A.5 分布之间的距离

有时我们可能还需要某种形式的概率分布之间的距离，以便评估两个分布的接近程度。$p(x)$ 和 $q(x)$ 之间的 Kullback-Leibler (KL) 散度是一个常见的选择：

$$\mathrm{KL}(p \parallel q) = \int p(x) \log \frac{p(x)}{q(x)} dx$$

KL散度不是一个合适的度量（它是不对称的，并且不满足三角不等式）。它下界为0，但没有上界。只有当对于任何满足 $q(x) = > 0$ 的 x，都有 $p(x) = > 0$（即 p 的支持集是 q 的支持集的子集）时，才能定义这种散度。当两个分布相同的时候，0是能够达到的最小值。KL散度可以写成期望值的形式，因此可以通过蒙特卡洛抽样来估计，如（E.A.7）所示。

A.6 最大似然估计

蒙特卡洛采样表明，如果我们能访问其样本，我们可以估计关于概率分布的感兴趣的数量。然而，我们可能对估计概率分布本身感兴趣。假设我们对它的函数形式有一个猜测 $f(x;\ s)$，其中 s 是充分统计量（例如高斯分布的均值和方差），以及一组 n 样本 $x_i \sim p(x)$。我们称这些样本为相同的（因为它们来自同一个概率分布）并且独立分布，简称为 i.i.d. 由于独立性，它们的联合分布对于任何选择的 s 都会分解：

$$p(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i; s)$$

大型产品在计算上不方便，但我们可以通过对数等效地将其重写为和

转换：

$$L(s) = \sum_{i=1}^{n} \log(f(x_i; s))$$

寻找最大化前述数量的参数 s 被称为最大似然（ML）方法。由于其重要性，我们简要重述如下。

定义 D.A.1（最大似然）

Given a parametric family of probability distributions $f(x; s)$, and a set of n values $\{x_i\}_{i=1}^{n}$ which are i.i.d. samples from an unknown distribution $p(x)$, the best approximation to $p(x)$ according to the ***maximum likelihood*** *(ML) principle is:*

$$s^* = \arg\max_s \sum_{i=1}^{n} \log(f(x_i; s))$$

如果 f 可微，我们可以通过梯度下降来最大化目标函数。这是我们训练可微模型的核心方法。目前，我们通过描述在标准概率分布下的机器学习估计的简单例子来结束附录。我们不提供详细的计算，具体可参考 [Bis06, BB23]。

最大似然估计对于伯努利分布

首先考虑参数 p 未知的高斯分布的情况。在这种情况下，最大似然估计量为：

$$p^* = \frac{\sum_i x_i}{n}$$

which是整个da中正样本比 taset.

最大似然估计高斯分布

对于高斯分布，我们可以将其对数似然重写为：

$$L(\mu, \sigma^2) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i - \mu)^2$$

最大化 μ 和 σ^2 分别返回计算高斯分布经验均值和方差的已知规则：

$$\mu^* = \frac{1}{n}\sum_i x_i \tag{E.A.8}$$

$$\sigma^{2*} = \frac{1}{n}\sum_i (x_i - \mu^*)^2 \tag{E.A.9}$$

两个可以依次计算。因为我们在方差的公式中使用了对均值的估计，所以得到的估计显示略微有偏差。这可以通过修改归一化项为$\frac{1}{n-1}$来纠正；这被称为贝塞尔校正.[1] 对于大的 n，两种变体之间的差异最小。

[1] https://en.wikipedia.org/wiki/Bessel%27s_correction

B | 1D 通用逼近

关于本章

虽然形式上证明通用逼近定理超出了本书的范围，但了解这类证明如何构建是有益的。在本附录中，我们遵循并扩展了M. Nielsen在2019年在线书籍章节中的视觉直觉[a]，我们将其用于扩展讨论（以及一些交互式可视化），特别是对于多维输入的情况。

[a] http://neuralnetworksanddeeplearning.com/chap4.html

我们关注Cybenko [Cyb89]提出的原始近似定理，该定理考虑具有一个具有sigmoid激活函数的隐藏层的模型。我们还限制分析到具有单个输入和单个输出的函数，这些函数可以轻松可视化。推理可以扩展到其他激活函数和更高维度。

这个视觉证明的轮廓相对简单：

1. 作为第一步，我们展示了如何手动设置隐藏层中单个神经元的权重，以逼近阶跃函数。

2. 然后，我们展示如何通过在隐藏层中添加另一个单元，使得可以逼近任何在小区间内恒定而在其他地方为零的函数（我们称这些区间函数为“bin”函数）。

3. 最后，我们描述了一种简单的方法来近似一个通用函数，首先将其分箱到所需的精度，然后添加所需的神经元以依次近似所有分箱。对于 m 个分箱，我们得到一个包含 $2m$ 个神经元的网络。对于具有多个输入的通用函数，这个数字会随着维度的增加而呈指数增长，使得在实际情况下证明不具有构造性。

B.1 步进函数的近似

首先，让我们考虑隐藏层中的一个单个神经元，在这种情况下，我们可以将网络的方程写成（忽略输出偏置项，因为它在我们的推导中并不有用）：

$$f(x) = a\sigma(wx + s)$$

为了可视化目的，我们在偏差上添加负号，并将乘法项因式分解在整个输入 σ (上，两个变体显然是等价的)：

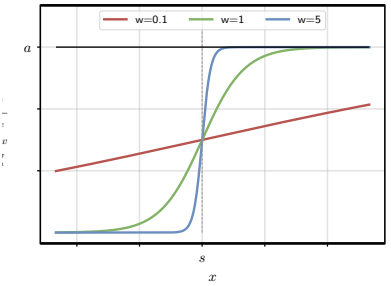


图 F.B.1: *A network with a single neuron in the hidden layer can be visualized as a sigmoid with controllable slope, center, and amplitude. We show here an example where we fix the amplitude and the center, but we vary the slope.*

$$f(x) = a\ \sigma(\ w\ (x - s\ )) \qquad \text{(E.B.1)}$$

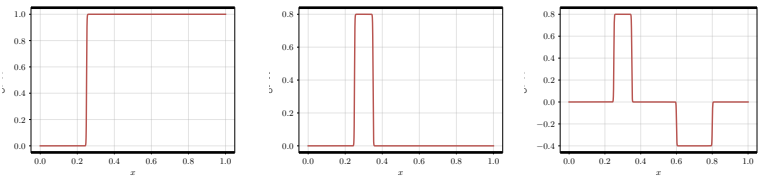
Amplitude Slope Shift

这与我们在第5.4节中介绍的“可调”sigmoid变体相似。特别是，在这个公式中，a控制sigmoid的振幅，w控制斜率，s通过一个固定量移动函数。

我们在图F.B.1中展示了(E.B.1)的几个图，其中我们固定 a 和 s，同时改变 w。如图所示，通过增加 w，斜率变得更为陡峭。将其固定为一个非常大的常数（例如，$w = 10^4$），我们得到了一个非常好的阶梯函数近似，我们可以控制阶梯的位置（s参数）和振幅（a参数），如图F.B.2a所示。

B.2 近似常数函数

如果我们添加一个振幅相反（并且略微偏移位置）的第二个神经元，我们可以近似一个在小区间内恒定的函数（我们称之为“bin”）



(a) *1 neuron* **(b)** *2 neurons* **(c)** *4 neurons*

图 F.B.2: *(a) A neural network with one input, one hidden neuron, and one output can approximate any step function (here shown with $a = 1$ and $s = 0.3$). (b) With two hidden neurons and one output we can approximate any function which is constant over a small interval. (c) With four neurons, we can approximate any function which is piecewise constant over two non-zero intervals. Note that bins can be negative by defining a negative amplitude.*

函数）。定义一个宽度 Δ，我们可以写出：

$$f(x) = a\sigma\left(w\left(x - \left(s - \frac{\Delta}{2}\right)\right)\right) - a\sigma\left(w\left(x - \left(s + \frac{\Delta}{2}\right)\right)\right) \quad \text{(E.B.2)}$$

Go up [down] at $s - \frac{\Delta}{2}$ Go down [up] at $s + \frac{\Delta}{2}$

我们在其中回忆，w 现在是一个大常数，例如 10^4。(E.B.2) 描述了一个函数（相当于一个具有两个神经元的单隐藏层模型），它在 $s - \frac{\Delta}{2}$ 时增加 a，在区间 $\left[s - \frac{\Delta}{2},\ s + \frac{\Delta}{2}\right]$ 上保持值 $f(x) = a$ 不变，然后之后减少到 0。一个例子在图 F.B.2b 中显示。

对于以下内容，我们可以将之前的函数重写为 $f(x;\ a、s、\Delta)$，以突出对三个参数 a、s 和 Δ 的依赖性。

B.3 逼近通用函数

因为 $f_{a,s,\Delta}(x)$ 在对应区间外有效值为0，定义在非相交区间上的两个函数不会相互影响，即我们刚才定义的“bin”函数具有高度局部化。因此，通过在隐藏层中添加两个额外的神经元，我们可以定义一个在两个单独的区间上恒定的函数（如图F.B.2c所示的一个例子）：

$$f(x) = f(x; a_1, s_2, \Delta_1) + f(x; a_2, s_2, \Delta_2)$$

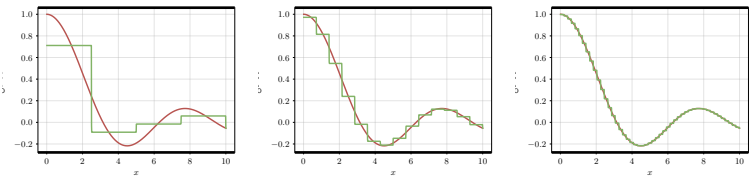
证明的其余部分现在很简单，通过将我们要逼近的函数分成许多小区间来进行。给定任何区间（我们为了简单起见假设为 $[0, 1]$）上的（连续）函数 $g(x)$，我们首先将输入域分成 m 个等间距的区间，其中 m 控制逼近的精度（m 越高，逼近越好）。因此，i-th 个区间跨越的区间为：

$$B_i = \left[\frac{i}{m} - \frac{\Delta}{2}, \frac{i}{m} + \frac{\Delta}{2}\right]$$

Δ是每个桶的大小。对于每个桶，我们计算该区间内$g(x)$的平均值：

$$g_i = \frac{1}{\Delta}\int_{x \in B_i} g(x)dx$$

最后，我们在隐藏层中定义了一个包含$2m$个神经元的网络。



(a) *5 bins* **(b)** *15 bins* **(c)** *50 bins*

图 F.B.3: *Approximating* $g(x) = \frac{\sin(x)}{x}$ *in* $[0, 10]$ *with (a)* $m = 5$*, (b)* $m = 15$*, and (c)* $m = 50$ *bins. The original function is in red, the approximation* (E.B.3) *in green. The average squared error in the three cases decreases exponentially (approximately* 0.02*,* 0.002*, and* 0.00016*).*

层，每个bin两个。每个bin函数位于bin中心，取值 g_i:

$$f(x) = \sum_{i=1}^{m} f\left(x; g_i, \frac{i}{m}, \Delta\right) \tag{E.B.3}$$

(Approximated) constant value

The i-th bin is centered in $\frac{i}{m}$

我们在图F.B.3中展示了$g(x) = \frac{\sin(x)}{x}$在增加的箱数（$m = 5$，$m = 15$，$m = 50$）情况下的此类近似示例。应该很明显，均方误差与m成反比，我们可以通过简单地增加近似的分辨率来尽可能减小误差。

类似的理由可以应用于多维输入和不同的激活函数.[1]

[1] http://neuralnetworksanddeeplearning.com/chap4.html

参考文献

[AB21] A. N. Angelopoulos 和 S. Bates. 对一致预测和无分布不确定性量化的温和介绍。*arXiv preprint arXiv:2107.07511*，2021。98 [ADIP21] A. Apicella，F. Donnarumma，F. Isgrò 和 R. Prevete。现代可训练激活函数综述。*Neural Networks*，138:14–32，2021。118 [AHS23] S. K. Ainsworth，J. Hayase 和 S. Srinivasa。Git re-basin：模态置换对称性下的模型合并。在 *ICLR*，2023。3 [AHSB14] F. Agostinelli，M. Hoffman，P. Sadowski 和 P. Baldi。学习激活函数以改进深度神经网络。*arXiv preprint arXiv:1412.6830*，2014。118 [AJB+17] D. Arpit，S. Jastrzębski，N. Ballas，D. Krueger，E. Bengio，M. S. Kanwal，T. Maharaj，A. Fischer，A. Courville，Y. Bengio 等。深度网络中记忆的近距离观察。在 *ICML*，2017。108 [AR00] J. A. Anderson 和 E. Rosenfeld。
Talking nets: An oral history of neural networks。麻省理工学院出版社，2000。9 [ARA+16] R. Al-Rfou，G. Alain，A. Almahairi，C. Angermueller，D. Bahdanau，N. Ballas，F. Bastien，J. Bayer，A. Belikov，A. Belopolsky 等。Theano：用于快速计算数学表达式的 Python 框架。*arXiv preprint arXiv:1605.02688*，第 1-19 页，2016。i，13 [ASA+23] E. Akyürek，D. Schuurmans，J. Andreas，T. Ma 和 D. Zhou。上下文学习中的学习算法是什么？使用线性模型的调查。在 *ICLR*，2023。2，3，57 [AST+24] A. F. Ansari，L. Stella，C. Turkmen，X. Zhang，P. Mercado，H. Shen，O. Shchur，S. S. Rangapuram，S. P. Arango，S. Kapoor 等。Chronos：学习时间序列的语言。
arXiv preprint arXiv:2403.07815，2024。188，278 [AZLL19] Z. Allen-Zhu，Y. Li 和 Y. Liang。在超参数化神经网络中学习和泛化，超越两层。在 *NeurIPS*，2019。110 [BAY22] S. Brody，U. Alon 和 E. Yahav。图注意力网络有多关注？在 *ICLR*，2022。307 [BB23] C. M. Bishop 和 H. Bishop。*Deep learning: Foundations and concepts*。施普林格自然出版社，2023。12，63，341，349

[BBCJ20] B. Biesialska, K. Biesialska 和 M. R. Costa-Jussa. 自然语言处理中的终身持续学习：一项调查。在 *COLING*，2020。57 [BBL+17] M. M. Bronstein，J. Bruna，Y. LeCun，A. Szlam 和 P. Vandergheynst。几何深度学习：超越欧几里得数据。*IEEE Signal Processing Magazine*，34(4):18–42，2017。167，296 [BCB15] D. Bahdanau，K. Cho 和 Y. Bengio。通过联合学习对齐和翻译进行神经机器翻译。在 *ICLR*，2015。240 [BCZ+16] T. Bolukbasi，K.-W. Chang，J. Y. Zou，V. Saligrama 和 A. T. Kalai。人是计算机程序员，女人是家庭主妇？消除词嵌入偏差。在 *NeurIPS*，2016。56 [Ben09] Y. Bengio。为人工智能学习深度架构。*Foundations and Trends® in Machine Learning*，2(1):1–127，2009。i，8，109 [BGHN24] J. Blasiok，P. Gopalan，L. Hu 和 P. Nakkiran。何时优化适当的损失会产生校准？在 *NeurIPS*，2024。95 [BGLA21] F. M. Bianchi，D. Grattarola，L. Livi 和 C. Alippi。具有卷积ARMA滤波器的图神经网络。*IEEE Transactions on Pattern Analysis and Machine Intelligence*，44(7):3496–3507，2021。298 [BGMMS21] E. M. Bender，T. Gebru，A. McMillan-Major 和 S. Shmitchell。关于随机鹦鹉的危险：语言模型可以太大吗？在 *ACM FAccT*。ACM，2021。56 [BGSW18] N. Bjorck，C. P. Gomes，B. Selman 和 K. Q. Weinberger。理解批量归一化。在 *NeurIPS*，2018。224 [BHB+18] P. W. Battaglia，J. B. Hamrick，V. Bapst，A. Sanchez-Gonzalez，V. Zambaldi，M. Malinowski，A. Tacchetti，D. Raposo，A. Santoro，R. Faulkner 等人。关系归纳偏差，深度学习和图网络。*arXiv preprint arXiv:1806.01261*，2018。309 [Bis95] C. M. Bishop。在噪声中训练相当于Tikhonov正则化。*Neural Computation*，7(1):108–116，1995。211 [Bis06] C. Bishop。*Pattern recognition and machine learning*。Springer，2006。68，69，81，97，341，349 [BJMO12] F. Bach，R. Jenatton，J. Mairal 和 G. Obozinski。具有稀疏诱导惩罚的优化。*Foundations and Trends® in Machine Learning*，4(1):1–106，2012。208 [BKH16] J. L. Ba，J. R. Kiros 和 G. E. Hinton。层归一化。*arXiv preprint arXiv:1607.06450*，2016。226，325 [BKK19] S. Bai，J. Z. Kolter 和 V. Koltun。深度平衡模型。在 *NeurIPS*，2019。321 [Ble90] G. E. Blelloch。*Prefix sums and their applications*。卡内基梅隆大学计算机科学学院匹兹堡，宾夕法尼亚州，美国，1990。330 [BN24] J. Bernstein 和 L. Newhouse。旧优化器，新规范：一部选集。*arXiv preprint arXiv:2409.20325*，2024。47 [BNS06] M. Belkin，P. Niyogi 和 V. Sindhwani。流形正则化：从标记和无标记示例中学习的几何框架。*Journal of Machine Learning Research*，7(11)，2006。57，292，293 [BP20] J. Bolte 和 E. Pauwels。机器学习中自动微分的一个数学模型。在 *NeurIPS*，2020。145 [BPA+24] F. Bordes，R. Y. Pang，A. Ajay，A. C. Li，A. Bardes，S. Petryk，O. Mañas，Z. Lin，A. Mahmoud，B. Jayaraman 等人。视觉语言建模简介。*arXiv preprint arXiv:2405.17247*，2024。240，278

[BPRS18] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, 和 J. M. Siskind. 机器学习中的自动微分：综述。*Journal of Marchine Learning Research*，18:1–43，2018。123 [BPS+24] M. Beck, K. Pöppel, M. Spanring, A. Auer, O. Prudnikova, M. Kopp, G. Klambauer, J. Brandstetter, 和 S. Hochreiter. xLSTM：扩展长短期记忆。*arXiv preprint arXiv:2405.04517*，2024。327 [BR07] V. I. Bogachev 和 M. A. S. Ruas. *Measure theory*。Springer，2007。344 [BR24] M. Blondel 和 V. Roulet. 可微编程的元素。*arXiv preprint arXiv:2403.14606*，2024。8，39，124，135 [BZMA20] A. Baevski，Y. Zhou，A. Mohamed，和 M. Auli。wav2vec 2.0：语音表示的自监督学习框架。在 *NeurIPS*，2020。277 [CCGC24] Y. Cheng，G. G. Chrysos，M. Georgopoulos，和 V. Cevher。多线性算子网络。在 *ICLR*，2024。281 [CMMB22] F. Cinus，M. Minici，C. Monti，和 F. Bonchi。人们推荐者对回音室和极化的影响。在 *AAAI ICWSM*，2022。53 [CPPM22] E. Chien，C. Pan，J. Peng，和 O. Milenkovic。You are AllSet：超图神经网络的多集函数框架。在 *ICLR*，2022。309 [CRBD18] R. T. Chen，Y. Rubanova，J. Bettencourt，和 D. K. Duvenaud。神经常微分方程。在 *NeurIPS*，2018。233 [CS21] S. Chung 和 H. Siegelmann。有界精度循环神经网络的可计算完备性。在 *NeurIPS*，2021。320 [CVMG+14] K. Cho，B. Van Merriënboer，C. Gulcehre，D. Bahdanau，F. Bougares，H. Schwenk，和 Y. Bengio。使用 RNN 编码器-解码器学习短语表示以进行统计机器翻译。在 *EMNLP*。ACL，2014。327 [CW82] D. Coppersmith 和 S. Winograd。矩阵乘法的渐近复杂性。*SIAM Journal on Computing*，11(3):472–492，1982。27 [Cyb89] G. Cybenko。Sigmoid 函数叠加的逼近。*Mathematics of Control, Signals and Systems*，2(4):303–314，1989。109，351 [CZJ+22] H. Chang，H. Zhang，L. Jiang，C. Liu，和 W. T. Freeman。MaskGIT：掩码生成图像转换器。在 *IEEE/CVF CVPR*，2022。277 [CZSL20] E. D. Cubuk，B. Zoph，J. Shlens，和 Q. V. Le。RandAugment：具有减少搜索空间的实用自动化数据增强。在 *IEEE/CVF CVPR Workshops*，2020。212 [DBK+21] A. Dosovitskiy，L. Beyer，A. Kolesnikov，D. Weissenborn，X. Zhai，T. Unterthiner，M. Dehghani，M. Minderer，G. Heigold，S. Gelly，等。一张图像等于16x16个单词：用于大规模图像识别的转换器。在 *ICLR*，2021。261，276，281 [DCLT18] J. Devlin，M.-W. Chang，K. Lee，和 K. Toutanova。BERT：用于语言理解的深度双向转换器预训练。在 *NAACL*。ACL，2018。275 [DCSA23] A. Défossez，J. Copet，G. Synnaeve，和 Y. Adi。高保真神经网络音频压缩。*Transactions on Machine Learning Research*，2023。278 [DDM+23] M. Dehghani，J. Djolonga，B. Mustafa，P. Padlewski，J. Heek，J. Gilmer，A. P. Steiner，M. Caron，R. Geirhos，I. Alabdulmohsin，等。将视觉转换器扩展到220亿个参数。在 *ICML*，2023。279

[DFAG17] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier. 基于门控卷积网络的语言模型。在 *ICML*，2017。119 [DFE+22] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention：具有IO感知的快速和内存高效的精确注意力。在 *NeurIPS*，2022。272 [DLL+22] V. P. Dwivedi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson. 具有可学习结构和位置表示的图神经网络。在 *ICLR*，2022。311，312 [DOMB24] T. Darcet, M. Oquab, J. Mairal, and P. Bojanowski. 视觉Transformer需要寄存器。在 *ICLR*，2024。262 [DS20] S. De and S. Smith. 批标准化使深度网络中的残差块偏向恒等函数。在 *NeurIPS*，2020。229 [DT17] T. DeVries and G. W. Taylor. 使用cutout改进卷积神经网络的正则化。*arXiv preprint arXiv:1708.04552*，2017。221 [DZPS19] S. S. Du, X. Zhai, B. Poczos, and A. Singh. 梯度下降可以证明地优化过参数化的神经网络。在 *ICLR*，2019。110 [EHB23] F. Eijkelboom, R. Hesselink, and E. J. Bekkers. E (n)等变消息传递单纯复形网络。在 *ICML*，2023。309 [FAL17] C. Finn, P. Abbeel, and S. Levine. 用于快速适应深度网络的模型无关元学习。在 *ICML*，2017。57 [Fle23] F. Fleuret。*The Little Book of Deep Learning*。Lulu Press, Inc.，2023。12 [GBGB21] D. J. Gauthier, E. Bollt, A. Griffith, and W. A. Barbosa. 新一代水库计算。*Nature Communications*，12(1)：5564，2021。322 [GD23] A. Gu and T. Dao. Mamba：具有选择性状态空间的线性时间序列建模。*arXiv preprint arXiv:2312.00752*，2023。337，338 [GDE+20] A. Gu, T. Dao, S. Ermon, A. Rudra, and C. Ré. Hippo：具有最优多项式投影的循环记忆。在 *NeurIPS*，2020。328 [GFGS06] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. 连接主义时间分类：使用循环神经网络对未分割序列数据进行标记。在 *ICML*，2006。278 [GG16] Y. Gal and Z. Ghahramani. Dropout作为贝叶斯近似：在深度学习中表示模型不确定性。在 *ICML*，2016。219 [GGR22] A. Gu, K. Goel, and C. Ré. 使用结构化状态空间有效地建模长序列。在 *ICLR*，2022。328，332 [GJG+21] A. Gu, I. Johnson, K. Goel, K. Saab, T. Dao, A. Rudra, and C. Ré. 结合循环、卷积和连续时间模型与线性状态空间层。在 *NeurIPS*，2021。329，338 [GM17] C. Gallicchio and A. Micheli. 深度水库计算网络的回声状态属性。*Cognitive Computation*，9:337–350，2017。325 [GMS05] M. Gori, G. Monfardini, and F. Scarselli. 图域学习的新模型。在 *IEEE IJCNN*。IEEE，2005。321 [GOV22] L. Grinsztajn, E. Oyallon, and G. Varoquaux。为什么基于树的模型在典型表格数据上仍然优于深度学习？在 *NeurIPS*，2022。151 [GPE+23] S. Golkar, M. Pettee, M. Eickenberg, A. Bietti, M. Cranmer, G. Krawezik, F. Lanusse, M. McCabe, R. Ohana, L. Parker, 等等。xVal：大型语言模型的连续数字编码。*arXiv preprint arXiv:2310.02989*，2023。183

[GPSW17] C. Guo, G. Pleiss, Y. Sun, 和 K. Q. Weinberger. 关于现代神经网络的校准。在 *ICML*，2017。98 [Gri12] A. Griewank. 谁发明了微分反向模式?
Documenta Mathematica, Extra Volume ISMP，389400，2012。124 [GSBL20] M. Geva, R. Schuster, J. Berant, 和 O. Levy. Transformer前馈层是键值存储。在 *EMNLP*。ACL，2020。269 [GSR+17] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, 和 G. E. Dahl. 用于量子化学的神经消息传递。在 *ICML*，2017。308 [GW08] A. Griewank 和 A. Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*。SIAM，2008。124，139 [GWFM+13] I. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, 和 Y. Bengio. Maxout网络。在 *ICML*，2013。119 [GZBA22] D. Grattarola, D. Zambon, F. M. Bianchi, 和 C. Alippi. 理解图神经网络中的池化。
IEEE Transactions on Neural Networks and Learning Systems，2022。298 [HABN+21] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, 和 A. Peste. 深度学习中的稀疏性：神经网络中高效推理和训练的剪枝和增长。*Journal of Machine Learning Research*，22(241):1–124，2021。208 [HBE+24] A. Ho, T. Besiroglu, E. Erdil, D. Owen, R. Rahman, Z. C. Guo, D. Atkinson, N. Thompson, 和 J. Sevilla. 语言模型中的算法进步。
arXiv preprint arXiv:1710.05941，2024。1，2 [HDLL22] W. Hua, Z. Dai, H. Liu, 和 Q. Le. 线性时间内的Transformer质量。在 *ICML*，2022。280 [HG16] D. Hendrycks 和 K. Gimpel. 高斯误差线性单元（GELUs）。*arXiv preprint arXiv:1606.08415*，2016。117 [HHWW14] P. Huang, Y. Huang, W. Wang, 和 L. Wang. 用于聚类的深度嵌入网络。在 *ICPR*。IEEE，2014。55 [Hoc98] S. Hochreiter. 递归神经网络学习和梯度消失。
International Journal Of Uncertainity, Fuzziness and Knowledge-Based Systems，6(2):107–116，1998。325 [Hor91] K. Hornik. 多层前馈网络的逼近能力。*Neural Networks*，4(2):251–257，1991。109 [HR22] M. Hardt 和 B. Recht。
Patterns, predictions, and actions: Foundations of machine learning。普林斯顿大学出版社，2022。12 [HS97] S. Hochreiter 和 J. Schmidhuber. 长短期记忆。*Neural Computation*，9(8):1735–1780，1997。327 [HSS08] T. Hofmann, B. Schölkopf, 和 A. J. Smola。机器学习中的核方法。*The Annals of Statistics*，36(3):1171–1220，2008。108，316，317 [HTF09] T. Hastie, R. Tibshirani, 和 J. H. Friedman。*The elements of statistical learning: data mining, inference, and prediction*。Springer，2009。55，95 [HWG25] S. Hwang, B. Wang, 和 A. Gu。端到端分层序列建模的动态分块。
arXiv preprint arXiv:2507.07955，2025。183 [HYL17] W. Hamilton, Z. Ying, 和 J. Leskovec。在大图上的归纳表示学习。在 *NeurIPS*，2017。304 [HZC+17] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, 和 H. Adam。MobileNets：高效的卷积神经网络

网络用于移动视觉应用。*arXiv preprint arXiv:1704.04861*，2017. 170 [HZRS15] K. He, X. Zhang, S. Ren, 和 J. Sun. 深入研究整流器：在ImageNet分类中超越人类水平的表现。在 *IEEE ICCV*，2015. 116 [HZRS16] K. He, X. Zhang, S. Ren, 和 J. Sun. 深度残差学习用于图像识别。在 *IEEE/CVF CVPR*，2016. 229，230，231，234 [ICS22] K. Irie, R. Csordás, 和 J. Schmidhuber. 重新审视神经网络的双向形式：通过注意力的焦点将测试时间预测与训练模式连接起来。在 *ICML*，2022. 79 [IS15] S. Ioffe 和 C. Szegedy. 批标准化：通过减少内部协变量偏移来加速深度网络训练。在 *ICML*，2015. 221 [JGB+21] A. Jaegle, F. Gimeno, A. Brock, O. Vinyals, A. Zisserman, 和 J. Carreira. Perceiver：通过迭代注意力实现通用感知。在 *ICML*，2021. 271 [JK+17] P. Jain, P. Kar, 等。非凸优化用于机器学习。*Foundations and Trends® in Machine Learning*，10(3-4):142–363，2017. 44 [JLB+22] L. V. Jospin, H. Laga, F. Boussaid, W. Buntine, 和 M. Bennamoun。动手实践贝叶斯神经网络——深度学习用户的教程。*IEEE Computational Intelligence Magazine*，17(2):29–48，2022。66，68 [KB15] D. P. Kingma 和 J. Ba。Adam：一种随机优化方法。在 *ICLR*，2015。46 [KG21] P. Kidger 和 C. Garcia。Equinox：通过可调用的PyTrees和过滤转换在JAX中的神经网络。*Differentiable Programming Workshop, NeurIPS*，2021。121 [KL18] S. M. Kakade 和 J. D. Lee。对于合格程序的可证明正确的自动子微分。在 *NeurIPS*，2018。145 [KMH+20] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, 和 D. Amodei。神经网络的语言模型的比例法则。*arXiv preprint arXiv:2001.08361*，2020。1，240 [KPR+17] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, 等。克服神经网络中的灾难性遗忘。*Proceedings of the National Academy of Sciences*，114(13):3521–3526，2017。69 [KSH12] A. Krizhevsky, I. Sutskever, 和 G. E. Hinton。使用深度卷积神经网络进行ImageNet分类。在 *NeurIPS*，2012。115，204 [KVPF20] A. Katharopoulos, A. Vyas, N. Pappas, 和 F. Fleuret。Transformers是RNNs：具有线性注意力的快速自回归变换器。在 *ICML*，2020。315，317 [KW17] T. N. Kipf 和 M. Welling。使用图卷积网络的半监督分类。在 *ICLR*，2017。295 [Lau19] S. Laue。自动微分与符号微分等价。*arXiv preprint arXiv:1904.02990*，2019。129 [LBBH98] Y. LeCun, L. Bottou, Y. Bengio, 和 P. Haffner。基于梯度的学习应用于文档识别。*Proceedings of the IEEE*，86(11):2278– 2324，1998。9 [LBH15] Y. LeCun, Y. Bengio, 和 G. Hinton。深度学习。*Nature*，521(7553):436–444，2015。8

[LCX{v*}23{v*} 李杰，程宇，夏志，莫宇，黄国。通过多元投影的广义激活。{v*}，2023。118 {v*}LDR23{v*} 莉亚琳·李，德什潘德，鲁姆什基。缩小以扩大规模：参数高效微调指南。{v*}，2023。57 {v*}LDSL21{v*} 刘浩，戴志，苏，V. Le。注意MLP。在{v*}，2021。280 {v*}LH19{v*}洛什奇洛夫，胡特。解耦权重衰减正则化。在{v*}，2019。47，208 {v*}Lim21{v*} 李-浩。计算中的张量。{v*}，30:555–764，2021。9 {v*}LJ09{v*} 卢科舍夫斯基，雅格。循环神经网络训练的蓄水池计算方法。{v*}，3(3):127– 149，2009。322 {v*}LKM23{v*} 莱维坦，卡尔曼，马蒂亚斯。通过投机解码快速从Transformer中进行推理。在{v*}，2023。271 {v*}LLLG22{v*} 李，林，罗，归。超越节点和同质性的图表示学习。{v*}，35(5):4880–4893，2022。305 {v*}LLS21{v*} 李，李，宋。小型数据集的视觉Transformer。{v*}，2021。281 {v*}LMW{v*}22{v*} 刘，毛，吴，费希滕霍费尔，达尔，谢。2020年代的ConvNet。在{v*}，2022。231，234 {v*}LPW{v*}17{v*} 卢，浦，王，胡，王。神经网络的表示能力：从宽度看。在{v*}，2017。110 {v*}LRZ{v*}23{v*} 李，罗宾逊，赵，斯密特，斯，马龙，杰格尔卡。用于光谱图表示学习的符号和基不变网络。在{v*}，2023。312 {v*}LTM{v*}22{v*} 刘，谭，穆克，莫塔，黄，班萨尔，拉费尔。少量参数高效的微调比情境学习更好且更便宜。在{v*}，2022。3 {v*}LWV{v*}24{v*} 刘，王，维迪亚，鲁赫尔，哈尔弗森，索尔贾奇，侯，特格马克。KAN：科尔莫哥洛夫-阿诺德网络。{v*}，2024。119 {v*}LZA23{v*} 刘，扎哈里亚，阿贝尔。具有块状Transformer的环状注意力，用于近乎无限的上下文。在{v*}，2023。273 {v*}MCT{v*} 毛毛，陈，唐，赵，马，赵，沙，加尔金，唐。位置：图基础模型已经到来。在{v*}。312 {v*}Met22{v*} 梅茨。{v*}。企鹅，2022。8 {v*}MGMR24{v*} 范德，加尔金，莫里斯，兰帕谢克。关注图Transformer。{v*}，2024。310，312 {v*}MKS{v*}20{v*} 莫霍蒂，库拉哈里亚，萨扬，戈洛德茨，托尔，多卡尼亚。使用焦点损失校准深度神经网络。在{v*}，2020。98 {v*}MRF{v*}19{v*} 莫里斯，里茨特，费伊，汉密尔顿，伦森，拉坦，格罗赫。Weisfeiler和Leman走向神经网络：高阶图神经网络。在{v*}，卷33，第4602–4609页，2019。308

[MRT18] M. Mohri, A. Rostamizadeh, 和 A. Talwalkar. *Foundations of machine learning*. MIT Press, 2018. 60 [MSC+13] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, 和 J. Dean. 词和短语的分布式表示及其组合性。在 *NeurIPS*，2013。56 [MZBG18] G. Marra, D. Zanca, A. Betti, 和 M. Gori. 使用基于核的深度神经网络学习神经元的非线性。*arXiv preprint arXiv:1807.06302*，2018。118 [NCN+23] V. Niculae, C. F. Corro, N. Nangia, T. Mihaylova, 和 A. F. Martins. 神经网络中的离散潜在结构。*arXiv preprint arXiv:2301.07473*，2023。125 [ODG+23] A. Orvieto, S. De, C. Gulcehre, R. Pascanu, 和 S. L. Smith。关于线性递归后跟非线性投影的普遍性。在 *HLD 2023 Workshop, ICML*，2023。328 [ODZ+16] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, 和 K. Kavukcuoglu。WaveNet：原始音频的生成模型。在 *ISCA SSW Workshop*，2016。186，191，194 [OSG+23] A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, 和 S. De。复活循环神经网络以处理长序列。在 *ICML*，2023。328，333 [PAA+23] B. Peng, E. Alcaide, Q. Anthony, A. Albalak, S. Arcadinho, H. Cao, X. Cheng, M. Chung, M. Grella, K. K. GV, 等。RWKV：在transformer时代重新发明RNN。在 *EMNLP*。ACL，2023。335，336 [PABH+21] O. Puny, M. Atzmon, H. Ben-Hamu, I. Misra, A. Grover, E. J. Smith, 和 Y. Lipman。用于不变和等变网络设计的框架平均。*arXiv preprint arXiv:2110.03336*，2021。167 [PBE+22] A. Power, Y. Burda, H. Edwards, I. Babuschkin, 和 V. Misra。在小型算法数据集上超越过拟合的泛化。在 *1st Mathematical Reasoning in General Artificial Intelligence Workshop, ICLR*，2022。3，210 [PBL20] T. Poggio, A. Banburski, 和 Q. Liao。深度网络中的理论问题。*Proceedings of the National Academy of Sciences*，117(48):30039–30045，2020。60 [PGCB14] R. Pascanu, C. Gulcehre, K. Cho, 和 Y. Bengio。如何构建深度循环神经网络。在 *ICLR*，2014。320 [PKP+19] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, 和 S. Wermter。使用神经网络的持续终身学习：综述。*Neural Networks*，113:54–71，2019。57 [PNR+21] G. Papamakarios, E. Nalisnick, D. J. Rezende, S. Mohamed, 和 B. Lakshminarayanan。用于概率建模和推理的正态化流。*Journal of Machine Learning Research*，22(57):1–64，2021。233 [PP08] K. B. Petersen 和 M. S. Pedersen。*The matrix cookbook*。丹麦技术大学，2008。36 [PPR+24] A. Pagnoni, R. Pasunuru, P. Rodriguez, J. Nguyen, B. Muller, M. Li, C. Zhou, L. Yu, J. Weston, L. Zettlemoyer, 等。Byte潜在转换器：补丁比标记扩展得更好。*arXiv preprint arXiv:2412.09871*，2024。183 [PPVF21] S. Pesme, L. Pillaud-Vivien, 和 N. Flammarion。对于对角线性网络的SGD的隐式偏差：随机性的可证明好处。在 *NeurIPS*，2021。108

[PRCB24] A. Patel, C. Raffel, 和 C. Callison-Burch. Datadreamer：用于合成数据生成和可重复的LLM工作流程的工具。在 *ACL*。ACL，2024。211 [Pri23] S. J. Prince。*Understanding Deep Learning*。麻省理工学院出版社，2023。12 [PS+03] T. Poggio，S. Smale，等人。学习的数学：处理数据。*Notices of the AMS*，50(5):537–544，2003。60 [PSL22] O. Press，N. A. Smith，和 M. Lewis。训练短，测试长：具有线性偏差的注意力使输入长度外推成为可能。在 *ICLR*，2022。258 [QPF+24] S. Qiu，A. Potapczynski，M. Finzi，M. Goldblum，和 A. G. Wilson。计算更好的花费：用结构化矩阵替换密集层。*arXiv preprint arXiv:2406.06248*，2024。161 [RBOB18] M. Ravanelli，P. Brakel，M. Omologo，和 Y. Bengio。用于语音识别的光门控循环单元。*IEEE Transactions on Emerging Topics in Computational Intelligence*，2(2):92–102，2018。326 [RGD+22] L. Rampá ek，M. Galkin，V. P. Dwivedi，A. T. Luu，G. Wolf，和 D. Beaini。通用、强大、可扩展的图变换器的配方。在 *NeurIPS*，2022。310 [RHM86] D. E. Rumelhart，G. E. Hinton，和 J. L. McClelland。并行分布式处理的通用框架。在 *Parallel Distributed Processing Volume 1*。麻省理工学院出版社，1986。8 [RKG+22] D. W. Romero，D. M. Knigge，A. Gu，E. J. Bekkers，E. Gavves，J. M. Tomczak，和 M. Hoogendoorn。朝着用于 *n*D. *arXiv preprint arXiv:2206.03398*长距离依赖的通用CNN迈进。2022。242 [RKX+23] A. Radford，J. W. Kim，T. Xu，G. Brockman，C. McLeavey，和 I. Sutskever。通过大规模弱监督实现鲁棒的语音识别。在 *ICML*，2023。277，278 [RM22] J. W. Rocks 和 P. Mehta。无过拟合地记忆：在过参数化模型中的偏差、方差和插值。*Physical Review Research*，4(1):013201，2022。210 [RS21] M. N. Rabe 和 C. Staats。自注意力不需要 $\mathcal{O}(n^2)$内存。*arXiv preprint arXiv:2112.05682*，2021。272 [RSR+20] C. Raffel，N. Shazeer，A. Roberts，K. Lee，S. Narang，M. Matena，Y. Zhou，W. Li，和 P. J. Liu。通过统一的文本到文本变换器探索迁移学习的极限。*The Journal of Machine Learning Research*，21(1):5485–5551，2020。275 [RWC+19] A. Radford，J. Wu，R. Child，D. Luan，D. Amodei，I. Sutskever，等人。语言模型是无监督的多任务学习者。*OpenAI blog*，2019。55，270，275 [RZL17] P. Ramachandran，B. Zoph，和 Q. V. Le。寻找激活函数。*arXiv preprint arXiv:1710.05941*，2017。117 [SAL+24] J. Su，M. Ahmed，Y. Lu，S. Pan，W. Bo，和 Y. Liu。Roformer：具有旋转位置嵌入的增强变换器。*Neurocomputing*，568:127063，2024。258 [Sch15] J. Schmidhuber。神经网络中的深度学习：概述。*Neural Networks*，61:85–117，2015。327 [SCHU17] S. Scardapane，D. Comminiello，A. Hussain，和 A. Uncini。深度神经网络中的组稀疏正则化。*Neurocomputing*，241:81–89，2017。208

[SGT+08] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 图神经网络模型. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008. 321 [Sha19] N. Shazeer. 快速Transformer解码：你需要一个写头。 *arXiv preprint arXiv:1911.02150*, 2019. 279 [Sha20] N. Shazeer. GLU变体改进Transformer. *arXiv preprint arXiv:2002.05202*, 2020. 119 [SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout：防止神经网络过拟合的简单方法. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014. 215 [SHW21] V. G. Satorras, E. Hoogeboom, and M. Welling. E(n)等变图神经网络. 在 *ICML*，2021. 309 [SKF+99] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. 字节对编码：一种加速模式匹配的文本压缩方案. 1999. 181 [SKZ+21] A. Steiner, A. Kolesnikov, X. Zhai, R. Wightman, J. Uszkoreit, and L. Beyer. 如何训练你的ViT？数据增强和正则化在视觉Transformer中. *Transactions on Machine Learning Researc*, 2021. 281 [SLJ+15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. 深度卷积. 在 *IEEE CVPR*，2015. 166 [SMDH13] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. 深度学习中初始化和动量的重要性. 在 *ICML*，2013. 46 [SP97] M. Schuster and K. K. Paliwal. 双向循环神经网络. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997. 320 [SSBD14] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. 剑桥大学出版社，2014. 60 [Sti81] S. M. Stigler. 高斯与最小二乘法的发明. *The Annals of Statistics*, 页码 465–474, 1981. 8 [SVL14] I. Sutskever, O. Vinyals, and Q. V. Le. 使用神经网络的序列到序列学习. 在 *NeurIPS*，2014. 266 [SVVTU19] S. Scardapane, S. Van Vaerenbergh, S. Totaro, and A. Uncini. Kafnets：基于核的非参数激活函数用于神经网络. *Neural Networks*, 110:19–32, 2019. 118 [SW17] S. Scardapane and D. Wang. 神经网络中的随机性：概述. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(2):e1200, 2017. 317 [SWF+15] S. Sukhbaatar, J. Weston, R. Fergus, et al. 端到端记忆网络. 在 *NeurIPS*，2015. 268 [SWL23] J. T. Smith, A. Warrington, and S. W. Linderman. 简化的状态空间层用于序列建模. 在 *ICLR*，2023. 328, 330, 331 [TCB+24] M. Tiezzi, M. Casoni, A. Betti, M. Gori, and S. Melacci. 长序列处理中的状态空间建模：Transformer时代中的循环综述. *arXiv preprint arXiv:2406.09062*, 2024. 315 [TEM23] M. Tschannen, C. Eastwood, and F. Mentzer. GIVT：生成无限词汇Transformer. *arXiv preprint arXiv:2312.02116*, 2023. 277 [TGJ+15] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler. 使用卷积网络进行高效目标定位. 在 *IEEE/CVF CVPR*，2015. 221

[THK+21] I. O. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, et al. MLP-Mixer: 一种全MLP架构的视觉。在 *NeurIPS*，2021。279 [TLI+23] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: 开放且高效的基座语言模型。*arXiv preprint arXiv:2302.13971*，2023。3，56，275，280 [TNHA24] D. Teney, A. M. Nicolicioiu, V. Hartmann, and E. Abbasnejad. 神经红移：随机网络不是随机函数。在 *IEEE/CVF CVPR*，2024。108 [Unc15] A. Uncini. *Fundamentals of adaptive signal processing*。Springer，2015。162 [Vap13] V. Vapnik。*The nature of statistical learning theory*。Springer Science & Business Media，2013。60 [VCC+18] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. 图注意力网络。在 *ICLR*，2018。306 [Vel22] P. Veličković。一路向上传递消息。*arXiv preprint arXiv:2202.11097*，2022。308 [VSP+17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, . Kaiser, and I. Polosukhin。注意力就是一切。在 *NeurIPS*，2017。239，240，255，256，257，265，269 [VWB16] A. Veit, M. J. Wilber, and S. Belongie。残差网络的行为类似于相对较浅的网络集合。在 *NeurIPS*，2016。232 [WBF+24] S. Welleck, A. Bertsch, M. Finlayson, H. Schoelkopf, A. Xie, G. Neubig, I. Kulikov, and Z. Harchaoui。从解码到元生成：大型语言模型推理时间算法。*arXiv preprint arXiv:2406.16838*，2024。198 [WCW+23] C. Wang, S. Chen, Y. Wu, Z. Zhang, L. Zhou, S. Liu, Z. Chen, Y. Liu, H. Wang, J. Li, et al. 神经编解码器语言模型是零样本文本到语音合成器。*arXiv preprint arXiv:2301.02111*，2023。278 [WFD+23] H. Wang, T. Fu, Y. Du, W. Gao, K. Huang, Z. Liu, P. Chandak, S. Liu, P. Van Katwyk, A. Deac, et al. 人工智能时代的科学发现。*Nature*，620(7972):47–60，2023。1，187 [WGGH18] X. Wang, R. Girshick, A. Gupta, and K. He。非局部神经网络。在 *IEEE/CVF CVPR*，2018。243 [WJ21] Y. Wu and J. Johnson。重新思考“批量”在BatchNorm中的作用。*arXiv preprint arXiv:2105.07576*，2021。225 [WZZ+13] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus。使用DropConnect对神经网络进行正则化。在 *ICML*，2013。221 [XYH+20] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T. Liu。关于transformer架构中的层归一化。在 *ICML*，2020。259 [YCC+24] L. Yuan, Y. Chen, G. Cui, H. Gao, F. Zou, X. Cheng, H. Ji, Z. Liu, and M. Sun。重新审视NLP中的分布外鲁棒性：基准、分析和llms评估。在 *NeurIPS*，2024。53 [YHO+19] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo。CutMix：具有可定位特征的强分类器的正则化策略。在 *IEEE/CVF ICCV*，2019。213

[YLC{v*}22{v*} T. Yu, X. Li, Y. Cai, M. Sun, 和 P. Li. S2-MLP：用于视觉的空间移位MLP架构。在 {v*}，2022。280 {v*}YLZ{v*}22{v*} W. Yu, M. Luo, P. Zhou, C. Si, Y. Zhou, X. Wang, J. Feng, 和 S. Yan。Metaformer实际上是您需要的视觉解决方案。在 {v*}，2022。280 {v*}YTL{v*}25{v*} L. Yang, Y. Tian, B. Li, X. Zhang, K. Shen, Y. Tong, 和 M. Wang。Mmada：多模态大扩散语言模型。{v*} {v*}，2025。275 {v*}YYZ17{v*} B. Yu, H. Yin, 和 Z. Zhu。时空图卷积网络：交通预测的深度学习框架。在 {v*}，2017。309 {v*}ZBH{v*}21{v*} C. Zhang, S. Bengio, M. Hardt, B. Recht, 和 O. Vinyals。理解深度学习（仍然）需要重新思考泛化。{v*} {v*}，64(3):107–115，2021。2 {v*}ZCDLP17{v*} H. Zhang, M. Cisse, Y. N. Dauphin, 和 D. Lopez-Paz。mixup：超越经验风险最小化。在 {v*}，2017。213 {v*}ZER{v*}23{v*} A. Zador, S. Escola, B. Richards, B. Ölveczky, Y. Bengio, K. Boahen, M. Botvinick, D. Chklovskii, A. Churchland, C. Clopath, 等等。通过神经AI催化下一代人工智能。{v*} {v*}，14(1):1597，2023。8 {v*}ZJM{v*}21{v*} J. Zbontar, L. Jing, I. Misra, Y. LeCun, 和 S. Deny。Barlow twins：通过冗余减少进行自监督学习。在 {v*}，2021。55 {v*}ZKR{v*}17{v*} M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, 和 A. J. Smola。深度集。在 {v*}，2017。264 {v*}ZLLS23{v*} A. Zhang, Z. C. Lipton, M. Li, 和 A. J. Smola。{v*}。剑桥大学出版社，2023。12，45，46 {v*}ZS19{v*} B. Zhang 和 R. Sennrich。均方根层归一化。在 {v*}，2019。228 {v*}ZTS{v*}21{v*} S. Zhai, W. Talbott, N. Srivastava, C. Huang, H. Goh, R. Zhang, 和 J. Susskind。无注意力变换器。{v*} {v*}，2021。334，335 {v*}ZW23{v*} L. Ziyin 和 Z. Wang。spred：使用SGD解决 {v*}惩罚。在 {v*}，2023。208