

深度学习

DL-lec1

课程介绍

深度学习初探

序章

一、课程介绍

1、学习内容

深度学习的基本原理和重要模型

PyTorch 和其他深度学习框架的使用

典型的应用

动手实践、编程

2、课堂上会重点涵盖什么

深度学习的“入门”方法

一些在深度学习中通用的准则与思维方式

各种模型背后的逻辑与动机

从统计学的视角去理解深度学习模型

二、深度学习初探

1、无处不在

深度学习其实已经遍布了如今日常生活的许多角落：

刷脸进校

手机相机自动识别物品

电脑游戏中的 DLSS 加速和补帧技术

翻译软件

ChatGPT

AI 作画

2、例子

Yolo

<https://github.com/mxy493/YOLOv5-Qt>

Toon-Me

<https://github.com/vijishmadhavan/Toon-Me>

GuwenBERT

<https://github.com/Ethan-yt/guwenbert>

<https://huggingface.co/ethanyt/guwenbert-base>

Huggingface Transformers & Diffusers

<https://huggingface.co/docs/transformers>

<https://huggingface.co/docs/diffusers>

动画补帧、超分辨率

<https://www.bilibili.com/video/BV1V7411A7Th>

机器翻译

<https://www.deepl.com/translator>

DL-lec2

序章 (续)：深度学习简史

机器学习问题

线性模型

一、深度学习简史

1、人工智能的两个流派

人工智能在发展过程中衍生出了多种流派

核心焦点在于“如何实现智能”

主流的两种观点：符号主义、连接主义

1) 符号主义

基于符号和逻辑、符号表示信息、逻辑表示规则

自动定理证明、知识图谱

2) 连接主义

仿生学、认为人类的认知过程是大量神经元的信息处理过程、将大量简单的信息处理单元组成网络

神经网络

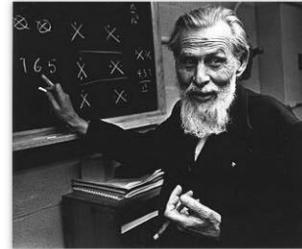
2、“气宗”VS“剑宗”



1) 起源

McCulloch & Pitts (1943)

A logical calculus of the ideas immanent in nervous activity



Warren McCulloch (1898-1969)
神经科学家



Walter Pitts (1923-1969)
数学家

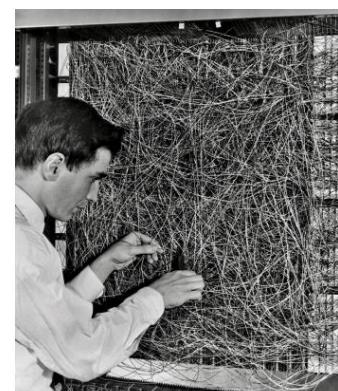
2) 感知机

Frank Rosenblatt (1958)

The perceptron: a probabilistic model for information storage and organization in the brain



Frank Rosenblatt (1928-1971)



3) 感知器

Rosenblatt 的感知器与当今的神经网络非常相像

相当于一个线性分类器 $y = \text{sgn}(w'x)$

实现了对参数的学习

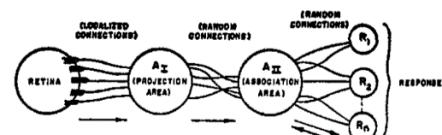


FIG. 1. Organization of a perceptron.

4) 质疑

Rosenblatt“树大招风”

另一位人工智能的奠基人 Minsky 曾对其进行过激烈的批评

指出感知器无法学习 XOR 函数 (1969)

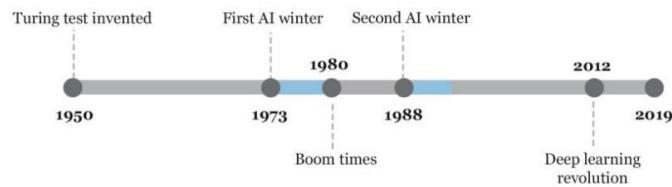


Marvin Minsky (1927-2016)

5) 第一次寒冬

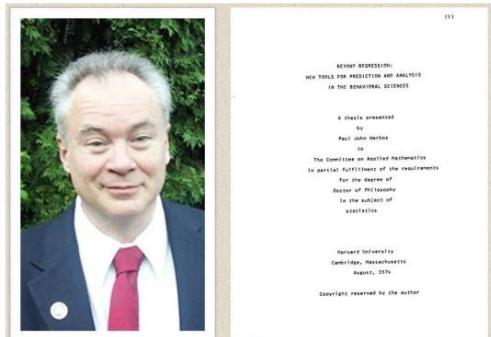
Minsky 的 XOR 问题指出 (单层) 感知器有巨大的局限而当时的研究者无法训练多层感知器

加上当时许多其他的因素，人工智能迎来了第一次寒冬



6) 复兴

1974 年 Paul Werbos 在其毕业论文中提出反向传播 (BP) 算法以解决多层神经网络的学习问题但在“第一次寒冬”中未受重视



Paul Werbos 及其毕业论文

1986 年，BP 被重新发现

1989 年，BP 被引入卷积神经网络，取得很大成功

7) 第二次低潮

多层神经网络计算复杂，解释性差

20 世纪 90 年代，SVM 等模型的流行度超过神经网络
神经网络的研究陷入低潮

8) 深度学习崛起

2006 年以后，各种因素促使神经网络迅速以“深度学习”之名崛起

海量数据的积累、GPU 等硬件设备的普及、Tensorflow 等软件极大降低入门门槛、神经网络研究者的坚持

9) 荣誉

2018 年，计算机领域的最高荣誉“图灵奖”颁给了深度学习领域的三位领军人物



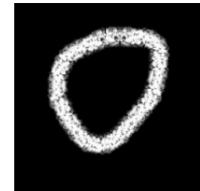
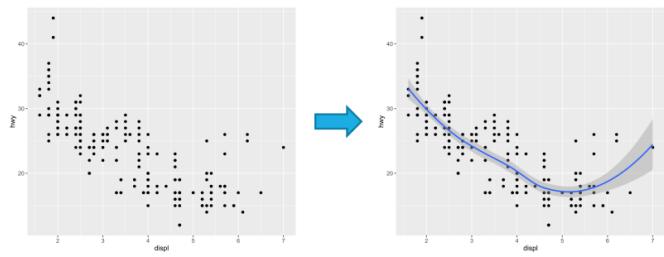
二、机器学习

1、机器学习

1) 深度学习 ∈ 机器学习

2) 机器学习的定义？

构建变量之间的函数关系？



0

X

Y



Cat

X

Y

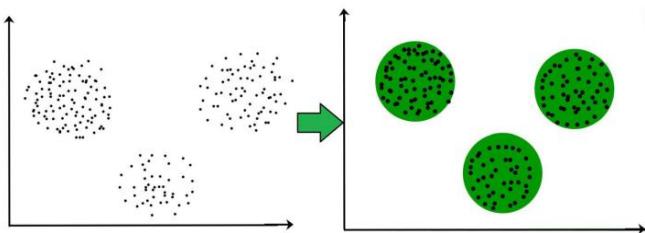


X

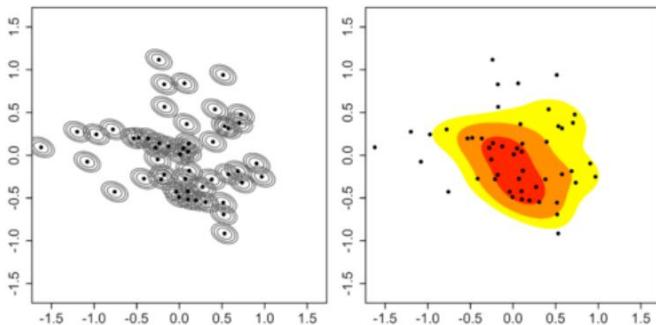
Y

3) 似乎很合理?

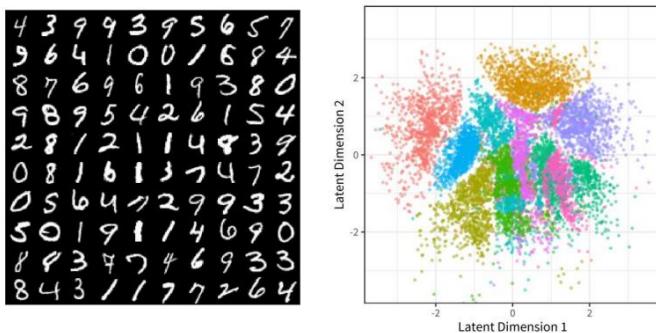
按照其定义, 聚类算不算机器学习?



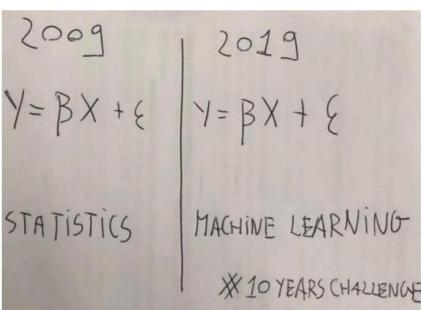
密度估计算不算机器学习?



降维算不算机器学习?



4) 我们似乎只能用一个很模糊的描述: 从数据中学习规律, **这是否就是统计学?**



很大一类机器学习模型可以称为**统计机器学习**

核心任务是找到数据的**生成规律**

或者说是数据的**统计分布**

5) 回归

研究 $p(y|x)$, Y 是连续型随机变量

如假定 $Y \sim N(\beta'x, \sigma^2)$

6) 二分类

研究 $p(y|x)$, Y 服从 Bernoulli 分布

如假定 $Y \sim Bernoulli(\text{sigmoid}(\beta'x))$

7) 多分类

研究 $p(y|x)$, Y 服从多项分布

如假定 $Y \sim Multinom(\text{softmax}(Wx))$

8) 密度估计

研究 $p(x)$

9) 聚类

数据 X 服从混合分布

每个 X 对应一个离散潜变量 $C \in \{1, \dots, K\}$

研究 $p(c|x)$

10) 降维

研究 $p(z|x)$, Z 是比 X 低维的随机变量

2、基础设施

1) 构成要素

要完成一项机器学习任务, 通常都要有以下几个部分
数据、模型、损失函数、优化算法

a. 数据

数据的大小和质量往往决定了模型效果的上限

巧妇难为无米之炊

b. 模型

描述数据是如何生成的

允许有一部分信息是未知的 (参数)

例: 密度估计, $Y \sim N(\mu, \sigma^2)$

例: 回归, $Y = \beta'x + \varepsilon$

c. 损失函数

因为有未知参数, 所以可能有无限多可能的模型

损失函数用来评价模型的好坏

例: 最小二乘准则

例: 极大似然准则

后面还将更深入讨论

d. 优化算法

根据损失函数制定的目标, 寻找“最优”的参数值

例: 梯度上升/下降

路漫漫其修远兮, 吾将上下而求索

2) 小结

数据是你的**粮草供应**

模型是你的**知识储备**

损失函数是你的**目标蓝图**

优化算法是你的**行动指南**

3) 机器学习的创新和进步, 很大程度上也是由这四项基本元素推动的

更丰富的数据类型, 更海量的数据

更具表达力的模型

更稳健、更具泛化能力的损失函数

更高效、收敛更快的优化算法

4) 从简单入手

假设数据充分、模型为**线性**、暂不考虑优化问题、重点
讨论损失函数如何选取

三、线性模型

1、线性回归

给定模型 $Y = \beta'x + \varepsilon$

给定数据 $(x_i, Y_i), i = 1, \dots, n$

如何估计 β ?

2、OLS

1) 最小二乘几乎成为了做回归的“铁律”

$$L(\beta) = n^{-1} \sum_{i=1}^n (Y_i - \beta' x_i)^2$$

但多想一步:

什么时候 OLS 具有好的性质?

如何向更复杂的问题扩展?

2) 极大似然

统计中一个经典的估计方法是极大似然

当我们有了数据 X_1, \dots, X_n 以及其分布的模型 $p_\theta(x)$

定义似然函数 $l(\theta; x) = \log p_\theta(x)$

极大似然准则 $\max_{\theta} n^{-1} \sum_{i=1}^n l(\theta; X_i)$

3) 在线性回归中, 如果假定误差是正态

$Y|x \sim N(\beta' x, \sigma^2)$

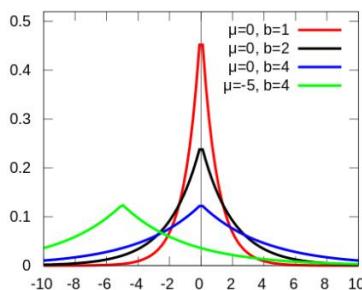
$$\log p(y|x) = -\frac{1}{2\sigma^2} (y - \beta' x)^2 + C$$

极大似然准则可以导出 OLS

4) 练习

如果误差服从 Laplace 分布

$$p(x; b) = \frac{1}{2b} e^{-|x|/b}$$



极大似然准则可以导出什么损失函数?

5) 损失函数

似然函数可能比你想象中的要重要得多

一个被经常忽视的事实是, 机器学习模型中很大一部分的损失函数都可以由似然函数导出

当你不知道选择什么损失函数的时候, 先试试看极大似然准则

3、线性二分类

1) 给定模型 $Y|x \sim Bernoulli(\rho(\beta' x))$

$\rho(\cdot)$ 是把 $\beta' x$ 映射到 $(0,1)$ 上的一个给定的函数

$\rho(\beta' x)$ 代表 Y 取 1 的概率

数据 $(x_i, Y_i), i = 1, \dots, n$

2) Logistic 回归

当 $\rho(x) = e^x/(e^x + 1)$

模型即为 Logistic 回归

利用极大似然准则, Bernoulli 分布的似然函数为

$$l(p; y) = y \log p + (1 - y) \log(1 - p)$$

其中 $p = \rho(\beta' x)$

3) 练习

根据前述信息, 写出 Logistic 回归关于 β 的损失函数

注意尽可能化简表达式

4) 思考题

除了 Logistic 回归, 你还能想出哪些 ρ 的构造方式?

它们对模型的最终结果影响大吗?

4、我们有办法超越极大似然准则吗?

1) KL 散度

我们先引入 KL 散度的概念

给定两个密度函数 p 和 q

$$\text{定义 } \text{KL}(q||p) = \int q(z) \log \frac{q(z)}{p(z)} dz$$

还可以写成 $\text{KL}(q||p) = E_q \log q(Z) - E_q \log p(Z)$

KL 散度的英文全称是 Kullback-Leibler divergence



Solomon Kullback



Richard Leibler

2) KL 性质

KL divergence 非负, $\text{KL}(q||p) \geq 0$

当 $q = p$ 时, $\text{KL}(q||p) = 0$

KL 不是关于 q 和 p 对称的

KL 可以用来衡量两个分布之间的区别

3) KL → MLE

现在让 $q = p^*$, 表示数据的真实分布

$p = p_\theta(x)$, 表示模型的分布

$$\text{KL}(p^*||p_\theta) = E_{p^*} \log p^*(Z) - E_{p^*} \log p_\theta(Z)$$

第一项与参数无关

$$\text{第二项 } E_{p^*} \log p_\theta(Z) \approx n^{-1} \sum_{i=1}^n \log p_\theta(X_i)$$

最小化 KL 等价于最大化似然函数

4) 启示

由于许多机器学习模型刻画的是数据的统计分布

所以对于这类问题, 损失函数可以看作是两个分布之间的差异

除了 KL 散度之外, 还有许多准则可以用来衡量分布之间的差异

启发了包括 GAN、WGAN 在内的众多流行的深度学习模型

5) 小结

定义损失函数是许多机器学习模型的关键

极大似然准则是非常重要的方法

可以作为默认选项

还存在很多其他损失函数的定义方式

如 hinge loss, 对应 SVM; check loss, 对应分位数回归

DL-lec3

一、线性模型（续）

5、线性多分类

给定模型 $Y \sim Multinom(\rho(Wx))$

$x \sim \mathbf{R}^P, Y \in \{0,1\}^k, Wx \in \mathbf{R}^k$

每一个 Y 只有一个元素是 1，其余为 0

$\rho(\cdot)$ 是把 Wx 映射到 $(0,1)^k$ 上的一个 **给定** 的函数，且得到的向量元素和为 1

$\rho(Wx)$ 代表 Y 取各类别的概率

数据 $(x_i, Y_i), i = 1, \dots, n$

估计 W

6、Softmax 回归

一种常见的 ρ 的选择是 Softmax 函数

$$\text{softmax}(w_1, \dots, w_k) = \frac{1}{Z}(e^{w_1}, \dots, e^{w_k})$$

其中 $Z = \sum_{j=1}^k e^{w_j}$

此时的模型通常称为 Softmax 回归

7、损失函数

多项分布 $Multinom(p)$ 的对数密度函数为

$$l(p; y) = \sum_{j=1}^k y_i \log p_i$$

其中 $p = \rho(Wx) = (p_1, \dots, p_k)$

似然函数 $L(\beta) = n^{-1} \sum_{i=1}^n l(\rho(Wx_i); Y_i)$

lec3-pytorch

1、Pytorch 基础

如遇到不清楚的函数或主题，可以查阅官方文档 (<https://pytorch.org/docs/stable/index.html>) 或利用搜索引擎寻求帮助

1.1 必备代码

PyTorch 通常会与 Numpy 包共同使用，因此导入 PyTorch 的时候按惯例也把 Numpy 加载进来。更重要的是，在每个 PyTorch 程序的最开头都应设置好随机数种子，从而让结果可重复。作业中也应在每题的代码开头设置随机数种子。由于 PyTorch 和 Numpy 使用了两套随机数生成机制，因此还需分别设置。

```
import numpy as np
import torch

np.random.seed(123456)
torch.manual_seed(123456)
<torch._C.Generator at 0x20d6b84abd0>
```

1.2 Tensor

PyTorch 中最为重要的数据结构是张量 (Tensor)，可以简单理解为多维数组，我们一般使用的向量和矩阵都是张量的特例。本次作业中我们只用到一维和二维 Tensor，分别对应向量和矩阵。

Tensor 可以通过 `torch.tensor()` 加上给定的数据创建，注意矩阵是按行创建的。

```
vec = torch.tensor([1.0, 2.0, 5.0])
```

```
print(vec)
```

```
tensor([1., 2., 5.])
```

```
mat = torch.tensor([[1.0, 2.0, 2.0], [3.0, 5.0, 4.5]])
```

```
print(mat)
```

```
tensor([[1.0000, 2.0000, 2.0000],
```

```
[3.0000, 5.0000, 4.5000]])
```

一些常用的 Tensor 有专用的创建方法：

```
torch.ones(3, 2)
```

```
tensor([[1., 1.],
```

```
[1., 1.],
```

```
[1., 1.]])
```

```
torch.zeros(5)
```

```
tensor([0., 0., 0., 0., 0.])
```

```
torch.linspace(3, 10, steps=5)
```

```
tensor([3.0000, 4.7500, 6.5000, 8.2500, 10.0000])
```

也可以给定形状生成随机数：

```
torch.randn(2, 3)
```

```
tensor([[1.8645, 0.4071, -1.1971],
```

```
[0.3489, -1.1437, -0.6611]])
```

Tensor 的形状可以通过 `shape` 属性来获取：

```
print(vec.shape)
```

```
print(mat.shape)
```

```
n = mat.shape[0]
```

```
p = mat.shape[1]
```

```
print(n)
```

```
print(p)
```

```
torch.Size([3])
```

```
torch.Size([2, 3])
```

```
2
```

```
3
```

将 Tensor 变形：

```
print(vec.view(3, 1))
```

```
print(mat.view(3, 2))
```

```
tensor([[1.,
```

```
[2.],
```

```
[5.]])
```

```
tensor([[1.0000, 2.0000],
```

```
[2.0000, 3.0000],
```

```
[5.0000, 4.5000]])
```

1.3 矩阵向量运算

Tensor 可以直观地与标量进行运算：

```
print(vec + 2.0)
```

```
print(mat * 1.2)
```

```
tensor([3., 4., 7.])
```

```
tensor([[1.2000, 2.4000, 2.4000],  
       [3.6000, 6.0000, 5.4000]])
```

逐元素运算的数学函数：

```
torch.exp(vec) # 不建议使用 torch.exp_  
tensor([ 2.7183, 7.3891, 148.4132])
```

```
torch.sin(mat)  
tensor([[ 0.8415, 0.9093, 0.9093],  
       [ 0.1411, -0.9589, -0.9775]])
```

汇总运算：

```
torch.mean(vec)  
tensor(2.6667)
```

```
torch.sum(mat)  
tensor(17.5000)
```

按坐标轴汇总：

```
print(torch.sum(mat, dim=0))  
print(torch.sum(mat, dim=1))  
tensor([4.0000, 7.0000, 6.5000])  
tensor([ 5.0000, 12.5000])
```

矩阵乘法：

```
torch.matmul(mat, vec)  
tensor([15.0000, 35.5000])
```

搭配转置：

```
torch.matmul(torch.t(mat), mat)  
tensor([[10.0000, 17.0000, 15.5000],  
       [17.0000, 29.0000, 26.5000],  
       [15.5000, 26.5000, 24.2500]])
```

1.4 统计分布

PyTorch 包含了许多常见的统计分布，参见说明文档 (<https://pytorch.org/docs/stable/distributions.html>)。大部分分布都可以计算 p.d.f. 和 c.d.f. 等函数，以及进行随机模拟抽样。

建立一个正态分布 $N(1,3)$ 对象：

```
import torch.distributions as D  
  
import math  
  
norm = D.Normal(loc=torch.tensor([1.0]),  
                 scale=torch.tensor([math.sqrt(3.0)]))
```

计算 $x=1,2,3$ 处的对数密度函数值：

```
norm.log_prob(torch.tensor([1.0, 2.0, 3.0]))  
tensor([-1.4682, -1.6349, -2.1349])
```

生成 5 个随机数（注意这里形状的写法）：

```
norm.sample(sample_shape=(5,))  
tensor([-1.9078],  
      [ 2.6222],  
      [ 1.6566],
```

```
[ 1.5657],  
[ 2.8333]])
```

分布的参数可以是一个向量，例如三个分别以 0.1, 0.5 和 0.9 为参数的 Bernoulli 分布：

```
bern = D.Bernoulli(probs=torch.tensor([0.1, 0.5, 0.9]))
```

各自生成一个随机数：

```
bern.sample()  
tensor([0., 1., 1.])
```

其他常见的操作可以参考 [官方教程] (https://pytorch.org/tutorials/beginner/basics/tensors_tutorial.html)，完整的函数列表可以查看 [官方 API 文档] (<https://pytorch.org/docs/stable/torch.html>)。

2、练习题

第 1 题

(a) 生成一个大小为 $[n \times p] = [200 \times 10]$ 的数据矩阵 x ，用正态分布 $N(0,2)$ 填充。随机数种子设为 123456。

```
n = 200  
p = 10
```

(b) 生成一个长度为 p 的向量 β ，每个元素服从均匀分布 $Uniform(-1,1)$ 。

(c) 生成一个长度为 n 的向量 ϵ ，每个元素服从独立正态 $N(0,0.1)$ 。

(d) 创建向量 y ，令其在数学上等于 $y = X\beta + \epsilon$

(e) 回归问题：给定数据 x 和 y ，估计 β 的取值。以 MSE 为损失函数，编写 Python 函数 `loss_fn_reg(bhat, x, y)`，用来返回任意 $\hat{\beta}$ 下的目标函数值。请用基础的矩阵和向量运算实现。

(f) Pytorch 中也提供了 MSE 损失函数，参见其文档 (<https://pytorch.org/docs/stable/generated/torch.nn.MSELoss.html>)。其用法是先建立一个损失函数对象，然后将 \hat{y} 和 y 作为参数传入。请利用这种方法计算如下给定 $\hat{\beta}$ 后的损失函数值，并与你自己的函数结果进行对比。

第 2 题

(a) 与第 1 题类似创建变量 x 和 β ，但使用不同的 n 和 p 。

(b) 定义函数 $sigmoid(x)$ ，其中 x 是一个 Tensor， $sigmoid(x) = e^x / (1 + e^x)$ 。

(c) 根据分布 $Y|X \sim Bernoulli(sigmoid(X\beta))$ ，生成 Y 的随机数。提示：参照 1.4 节的方法，先计算 Bernoulli 分布

的参数向量，然后生成随机数。

(d) 已知 $Bernoulli(\rho)$ 分布的对数密度函数为 $\log p(y; \rho) = y \log \rho + (1 - y) \cdot \log(1 - \rho)$ 。根据此信息，推导出给定 $\hat{\beta}$ 时的对数似然函数，并编写损失函数 `loss_fn_logistic(bhat, x, y)`，返回负对数似然值。

【说明文字】

(e) Pytorch 中也提供了 BCELoss 损失函数，参见其文档 (<https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html>)。其用法是先建立一个损失函数对象，然后将 $\hat{\rho}$ 和 y 作为参数传入。请利用这种方法计算如下给定 $\hat{\beta}$ 后的损失函数值，并与你自己的函数结果进行对比。

【说明文字】

第 3 题

(a) 多分类问题的数据通常包括数据阵 X 和标签向量 l ，其中标签为整数。在计算损失函数时，我们需要先将 l 转换成多项分布的 0-1 数据，即所谓 One-hot 编码。运行并观察下面的代码。

(b) 创建矩阵 W ，大小为 $k \times p$ ，用 $N(0,1)$ 填充其取值。

(c) 接下来计算对 Y 的概率预测值，其中每个 Y_i 观测对应一个等长的概率向量 p_i ，而 $p_i = \text{Softmax}(Wx_i)$ 。首先计算 Wx_i ，其中 x_i 是第 i 个观测。由于 X 是把 x_i 按行组合，因此矩阵形式表达为 $U = XW'$ ，其中 U 的第 i 行即为 Wx_i 。

我们先测试一下 $\text{Softmax}(Wx_{100})$ 的结果，观察其元素和是否为 1。代码中的 `dim=0` 意思是对第一个下标方向计算 Softmax，由于 $u[99]$ 是一个向量，因此第一个下标方向就是该向量本身。

而为了对 U 的每一行都计算 Softmax，我们可以直接对整个 u 矩阵用 `torch.softmax`，其中 `dim` 需指定为 1，意思是对第二个下标方向求 Softmax，即对 U 的每一行。原理类似于 1.3 节的按坐标轴汇总。请完成该计算，得到矩阵 P ，其中 P 的第 i 行即为 p_i 。

(d) 根据 y 和 p 两个矩阵，即可根据公式得到对数似然函数值。总结上述步骤，编写损失函数 `loss_fn_softmax(w, x, y)`，返回负对数似然值。

(e) Pytorch 中也提供了 CrossEntropyLoss 损失函数，参见其文档 (<https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>)。其用法是先建立一个损失函数对

象，然后将 U 和 l 作为参数传入（注意 U 是经过 Softmax 之前的矩阵， l 是原始的标签）。请利用这种方法计算损失函数值，并与你自己的函数结果进行对比。

lec3-autograd

1、自动微分

1) 自动微分是 PyTorch 的一个非常重要的功能，也是驱动深度学习快速发展的重要原因。

首先看一个简单的例子。考虑函数 $f(x, y) = x \cdot \log(x) + \sin(xy)$ ，我们想求 f 在 $(x, y) = (1, 2)$ 处对 x 和 y 的偏导。

首先创建 Tensor 对象：

```
import torch
x = torch.tensor([1.0])
y = torch.tensor([2.0])
x
tensor([1.])
```

为了告诉 PyTorch 要对 x 和 y 求导，我们需要设置 `requires_grad` 属性：

```
x.requires_grad = True
y.requires_grad = True
x
tensor([1.], requires_grad=True)
```

```
x = torch.tensor([1.0], requires_grad = True)
y = torch.tensor([2.0], requires_grad = True)
x
tensor([1.], requires_grad=True)
```

接下来利用 PyTorch 定义的运算计算函数值：

```
f = x * torch.log(x) + torch.sin(x * y)
f
tensor([0.9093], grad_fn=<AddBackward0>)
```

然后调用 `backward()` 函数进行反向传播：

```
f.backward()
```

此时 x 和 y 会有一个 `grad` 属性，即为计算出的导数值：

```
print(x.grad)
print(y.grad)
tensor([0.1677])
tensor([-0.4161])
```

我们可以手动计算 $\frac{\partial f}{\partial x} = \log(x) + 1 + y \cos(xy)$, $\frac{\partial f}{\partial y} = x \cos(xy)$ ，对结果进行验证。因为此时不再需要 PyTorch 记录导数，所以可以把结果放在 `torch.no_grad()` 中：

```
with torch.no_grad():
```

```
    print(torch.log(x) + 1.0 + y * torch.cos(x * y))
    print(x * torch.cos(x * y))
    tensor([0.1677])
    tensor([-0.4161])
```

自动微分同样适用于任意形状的 Tensor, 包括向量和矩阵。例如 $f(x, y) = (x + y)'(x + y)$, 其中 x 和 y 为向量。

```
torch.manual_seed(123)
```

```
x = torch.randn(5)
y = torch.rand(5)
x.requires_grad = True
y.requires_grad = True

f = (x + y).dot(x + y)
f.backward()

print(x.grad)
print(y.grad)
print(2.0 * (x + y))

tensor([-0.0717,  0.6340, -0.1064,  0.3226, -2.1567])
tensor([-0.0717,  0.6340, -0.1064,  0.3226, -2.1567])
tensor([-0.0717,  0.6340, -0.1064,  0.3226, -2.1567], grad_fn=<MulBackward0>)
```

思考题：给定一个行列式为正的矩阵 X , 定义 $f(X) = \log \det(X)$, 其中 $\det(X)$ 为 X 的行列式。那么 $\frac{\partial f}{\partial X}$ 应该是什么？

```
torch.manual_seed(123)
```

```
x = torch.randn(5, 5)
if x.det().item() < 0:
    x = -x

print(x.det().log())

x.requires_grad = True
f = torch.logdet(x)
f.backward()
print(x.grad)

tensor(1.1183)
tensor([[ 0.7673, -0.2915, -0.4447, -1.1218, -0.3021],
        [ 0.5376,  0.2348, -0.5945,  1.1406,  1.0058],
        [ 0.0020, -0.6879,  0.3541, -0.4003, -0.7196],
        [-0.1782,  0.3465, -0.3755,  0.6708,  1.1536],
        [ 0.0992,  0.9788, -0.3062,  0.9806,  1.7905]])
```

DL-lec4

今天的主题

前馈神经网络

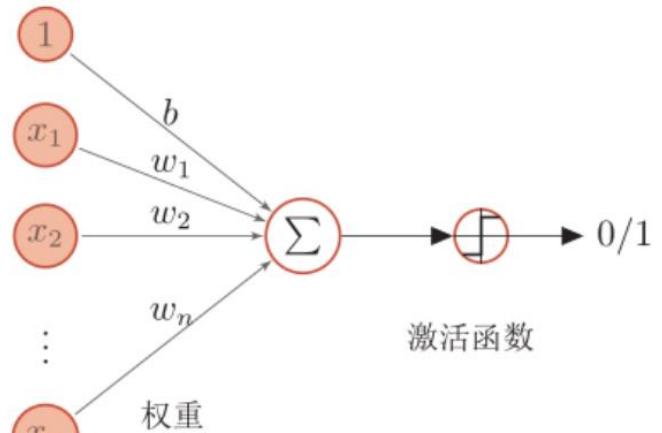
反向传播算法

一、前馈神经网络

1、前馈神经网络 ∈ 人工神经网络

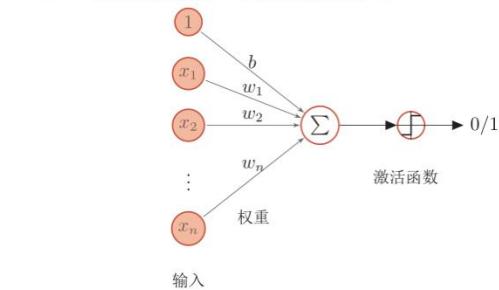
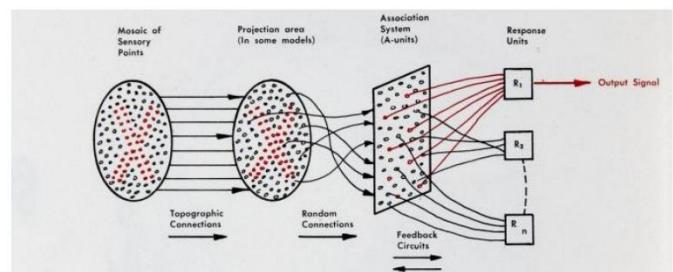
1) 人工神经元

- 神经网络是连接主义模型的典型代表
- 基本构成单元为人工神经元
- “知识”存储在神经元之间的连接上
- 神经元负责信号/数据的输入和输出



输入

- e. 现代的人工神经元结构基本源自 Rosenblatt



f. 输入信号 → 加权求和 → 激活输出

$$\begin{aligned} z &= \sum_{i=1}^d w_i x_i + b \\ &= \mathbf{w}^T \mathbf{x} + b, \end{aligned}$$

$a = f(z)$

权重

激活函数

基本相当于一个线性模型

- 2) 激活函数

a. Rosenblatt 当时的激活函数是一个阶梯函数,

例如 $a = f(z) = \begin{cases} 1 & z \geq 0.5 \\ 0 & z < 0.5 \end{cases}$

b. 当今广泛使用的激活函数一般满足如下性质：

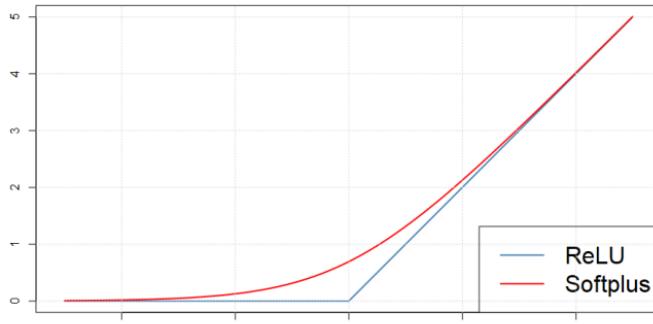
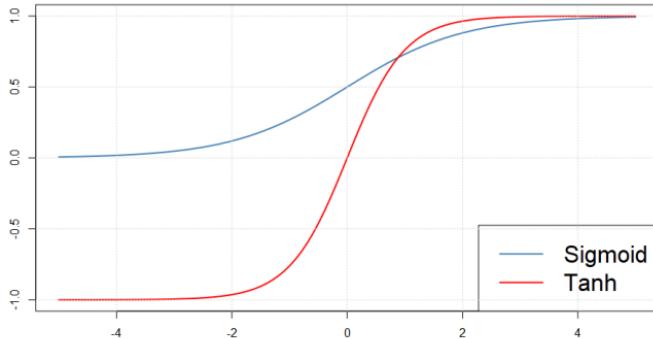
- a) 非线性
- b) 连续、除了少数点外处处可导
- c) 计算简单，数值稳定
- c. 常见激活函数

$$\text{Sigmoid } \sigma(x) = \frac{1}{1+exp(-x)}$$

$$\text{Tanh } tanh(x) = \frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$$

$$\text{ReLU } \text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases} = \max(0, x)$$

$$\text{Softplus } \text{softmax}(x) = \log(1 + exp(x))$$



d. 不知道怎么选择时，就以 ReLU 作为默认选项

a) 编程实现：

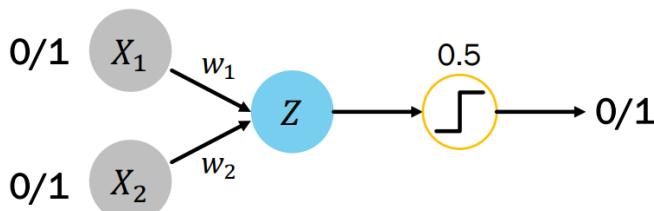
在编程实现中需要特别注意数值稳定性
特别是牵涉到指数函数 $exp(x)$

b) 试着用 Rosenblatt 的感知器模拟逻辑运算

两个输入： $X_1 \in \{0,1\}, X_2 \in \{0,1\}$

线性函数： $z = w_1X_1 + w_2X_2$

一个输出：若 $z > 0.5$ ，则 $Y = 1$ ，否则 $Y = 0$



c) 思考题：尝试找出适当的参数 w_1 和 w_2 ，使得这个人工神经元可以实现：

1. AND 运算
2. OR 运算

3. XOR 运算

INPUT	OUTPUT	
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

INPUT	OUTPUT	
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

INPUT	OUTPUT	
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

d) 启示

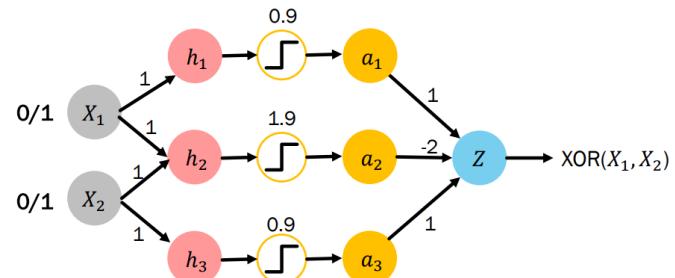
单层的感知器具有很大的局限

可以将神经元串联起来

构建多层的神经网络

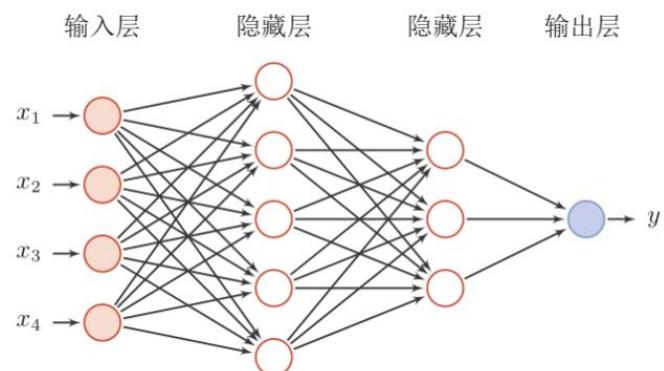
e) XOR 问题

只需增加一层神经元，即可模拟 XOR 运算



3) 前馈神经网络

- a. 将神经元按照一定的规则组合，可以得到复杂的神经网络
- b. 前馈神经网络是一种结构相对简单的神经网络
- c. 也称为全连接神经网络、多层次感知器



d. 各神经元分别属于不同的层，**层内无连接**

e. **相邻两层之间的神经元全部两两连接**

f. 信号从输入层向输出层**单向传播**

g. 正式定义

记号

含义

L 神经网络的层数

M_l 第 l 层神经元的个数

$f_l(\cdot)$ 第 l 层神经元的激活函数

$W^{(l)} \in R^{M_l \times M_{l-1}}$ 第 $l-1$ 层到第 l 层的权重矩阵

$b^{(l)} \in R^{M_l}$ 第 $l-1$ 层到第 l 层的偏置

$z^{(l)} \in R^{M_l}$ 第 l 层神经元的净输入（净活性值）

$a^{(l)} \in R^{M_l}$ 第 l 层神经元的输出（活性值）

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

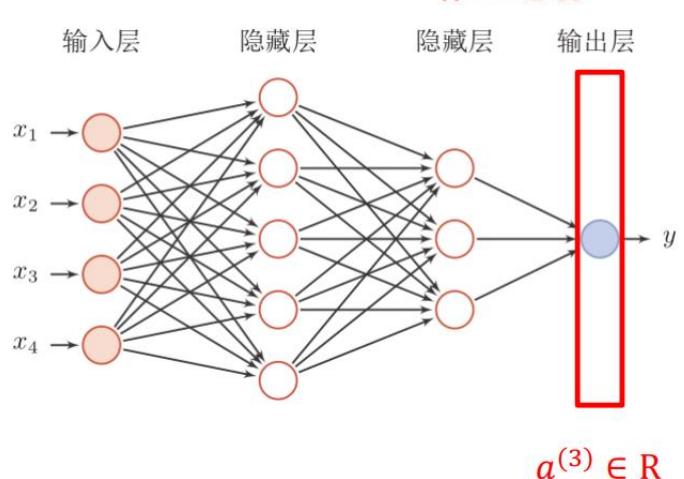
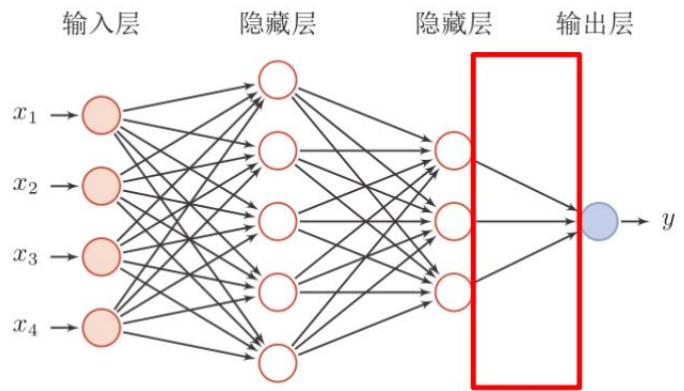
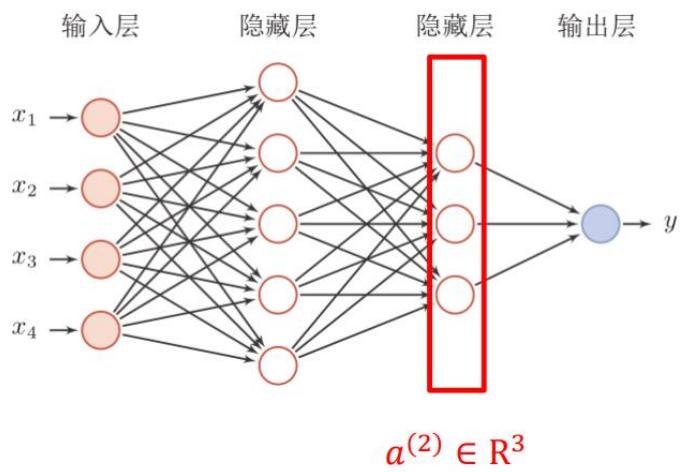
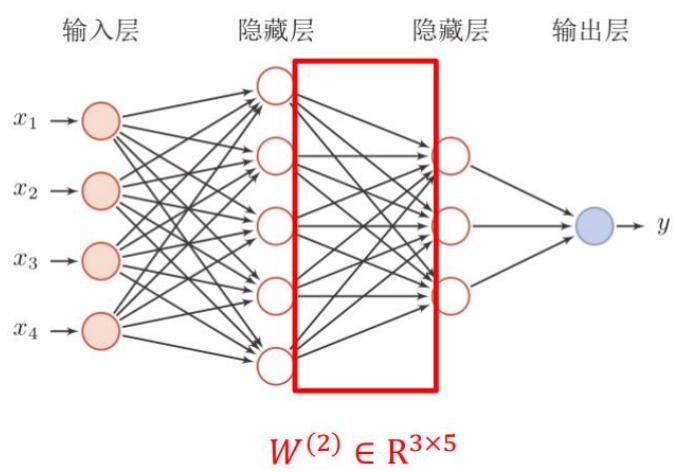
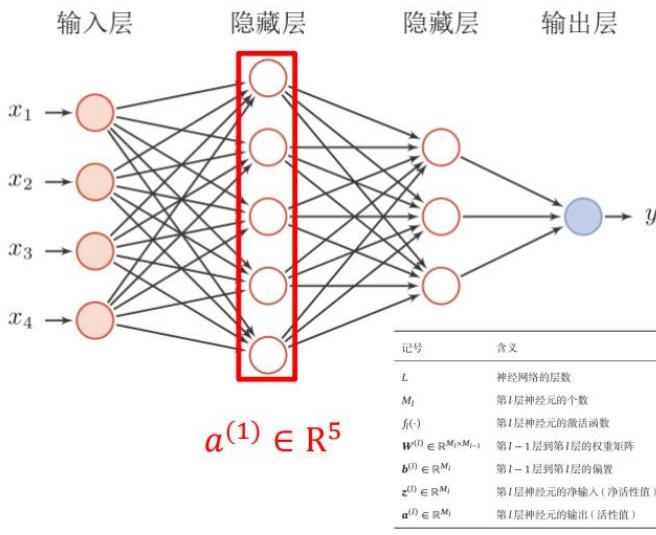
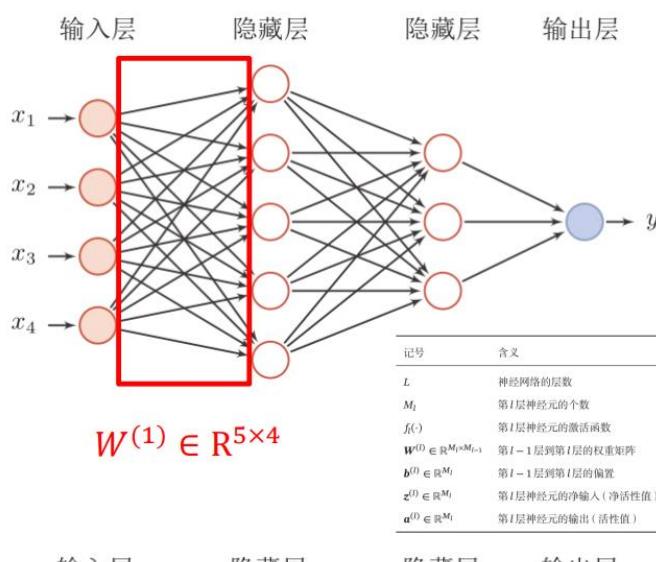
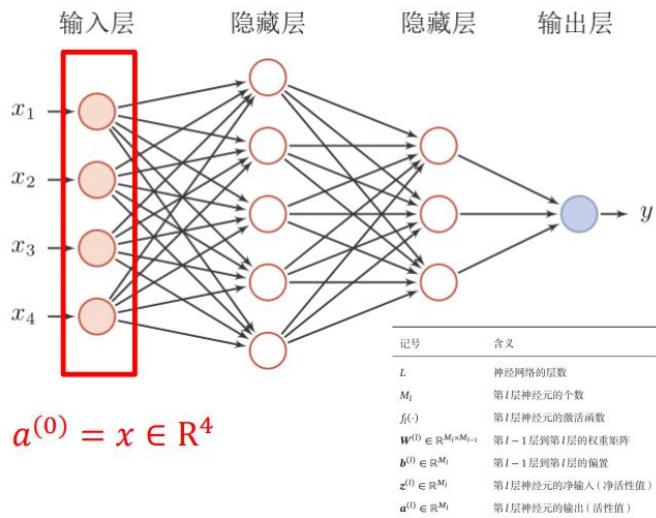
$$a^{(l)} = f_l(z^{(l)})$$

h. 信息前馈传播方式

$$x = a^{(0)} \rightarrow z^{(1)} \rightarrow a^{(1)} \rightarrow z^{(2)} \rightarrow \dots \rightarrow a^{(L-1)} \rightarrow z^{(L)}$$

$$\rightarrow a^{(L)} = \phi(x; W, b)$$

g.



h. 本质上，前馈神经网络是一个复合函数

$$y = f^5 \left(f^4 \left(f^3 \left(f^2 \left(f^1(x) \right) \right) \right) \right)$$

- a) 每一个 f^i 可以是固定的（如激活函数），或者是带参数的（如带权重的线性变换）
- b) 每个 f^i 都相对简单
- c) 但复合之后可以变得非常复杂

4) 通用近似定理

a. 之前我们展示了，一个两层的前馈神经网络可以拟合 XOR 函数

b. 事实上，两层的网络几乎可以拟合任意函数！

定理 4-1 通用近似定理：

令 $\varphi(\cdot)$ 是一个非常数、有界、单调递增的连续函数， L_d 是一个 d 维的单位超立方体 $[0,1]^d$ ， $C(L_d)$ 是定义在 L_d 上的连续函数集合。对于任何一个函数 $f \in C(L_d)$ ，存在一个整数 m ，和一组实数 $v_i, b_i \in \mathbb{R}$ 以及实数向量 $w_i \in \mathbb{R}^d, i = 1, \dots, m$ ，以至于我们可以定义函数

$$F(x) = \sum_{i=1}^m v_i \varphi(W_i^T x + b_i)$$

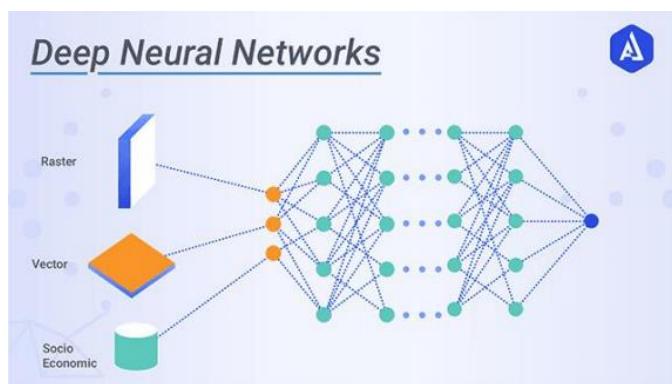
作为函数 f 的近似实现，即

$$|F(x) - f(x)| < \epsilon, \forall x \in L_d$$

其中 $\epsilon > 0$ 是一个很小的正数。

5) 深度神经网络

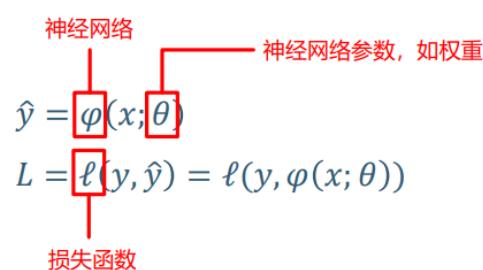
- a. 根据通用近似定理，两层的神经网路已经可以近似绝大部分函数
- b. 但一些新的研究表明，网络的 **深度** 可能发挥着重要的作用



c. 对于网络深度的研究是当前的热点课题

6) 模型训练

- a. 神经网络作为一个函数类具有良好的性质
- b. “万能函数”
- c. 接下来的问题在于如何估计其中的参数



d. 梯度下降法

$$\theta_{k+1} = \theta_k - \frac{\partial l(y; \varphi(x; \theta))}{\partial \theta} |_{\theta=\theta_k}$$

核心在于求损失函数对参数的导数

二、反向传播算法

1. 反向传播算法(Backpropagation, BP)

a. 本质是一种聪明、高效地求导数的办法

b. 一些吐槽

a) BP 的地位比较尴尬

b) 首先它确实很重要，甚至一度推动了历史的进程

c) 但如今几乎已经被更通用的自动微分取代

d) 现实中你以 99.9% 的可能不需要自己实现

c. 求导

a) 求导就像搬砖，总是可以用基础的方法慢慢来

b) 但有些方法可以更省力、更省时

DL-lec5

一、反向传播算法 (续)

1. 回到神经网络

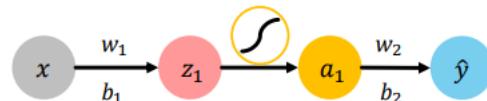
1) 利用之前介绍的方法，考虑一个简单的前馈神经网络

$$a_0 = x, z_1 = w_1 a_0 + b_1, a_1 = \sigma(z_1)$$

$$\hat{y} = z_2 = w_2 a_1 + b_2$$

$$l = (y - \hat{y})^2$$

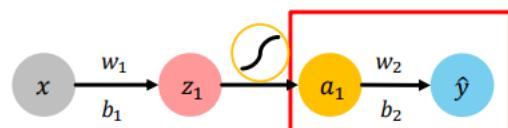
需要计算 $\frac{dl}{dw_1}, \frac{dl}{db_1}, \frac{dl}{dw_2}, \frac{dl}{db_2}$



$$\hat{y} = z_2 \text{ 的上游导数: } \frac{dl}{dz_2} = -2(y - z_2)$$

$$z_2 \text{ 对 } w_2, a_1, b_2 \text{ 的局部导数: } \frac{dz_2}{da_1} = w_2, \frac{dz_2}{dw_2} = a_1, \frac{dz_2}{db_2} = 1$$

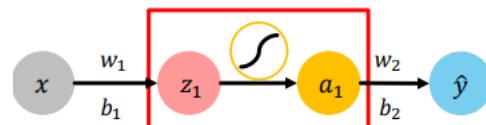
$$\text{因此 } \frac{dl}{da_1} = \frac{dl}{dz_2} \cdot w_2, \frac{dl}{dw_2} = \frac{dl}{dz_2} \cdot a_1, \frac{dl}{db_2} = \frac{dl}{dz_2}$$



$$a_1 \text{ 的上游导数 } \frac{dl}{da_1} \text{ 已经在上一层计算好}$$

$$a_1 \text{ 对 } z_1 \text{ 的局部导数: } \frac{da_1}{dz_1} = \sigma'(z_1) = \sigma(z_1)(1 - \sigma(z_1)) = a_1(1 - a_1)$$

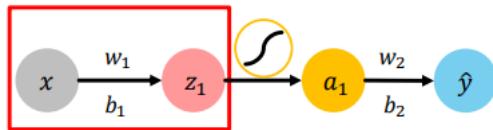
$$\text{因此 } \frac{dl}{dz_1} = \frac{dl}{da_1} \cdot a_1(1 - a_1)$$



z_1 的上游导数 $\frac{dl}{dz_1}$ 已经在上一层计算好

z_1 对 w_1, b_1 的局部函数: $\frac{dz_i}{dw_1} = x, \frac{dz_i}{db_1} = 1$

因此 $\frac{dl}{dw_1} = \frac{dl}{dz_1} \cdot x, \frac{dl}{db_1} = \frac{dl}{dz_1}$



2) 扩展

- a. 这一方法可以很容易扩展到任意层数的情形
- b. 但每层有多个神经元的情况怎么办?

此时 W_i 是矩阵, b_i 是向量

3) 规则 0:

理论: 参数导数的维度和参数的维度保持一致

实际: 参数导数的维度和参数的维度保持一致

4) 规则 1: 对于逐元素计算的激活函数, $a = \sigma(z)$

$$a = (a^1, \dots, a^m)^T, z = (z^1, \dots, z^m)^T, a^i = \sigma(z^i)$$

导数也可逐元素计算

假设 a 的上游导数 $\frac{dl}{da} = \left(\frac{dl}{da^1}, \dots, \frac{dl}{da^m} \right)^T$, 则 $\frac{dl}{dz} = \left(\frac{dl}{da^1} \cdot \sigma'(z^1), \dots, \frac{dl}{da^m} \cdot \sigma'(z^m) \right)^T$

$$\sigma'(z^1), \dots, \frac{dl}{da^m} \cdot \sigma'(z^m) = \frac{dl}{da} \circ \sigma'(z)$$

理论: 对于逐元素计算的激活函数, 导数也可逐元素计算

$$a: [m \times 1], z: [m \times 1], a_i = \sigma(z_i)$$

$$\frac{dl}{dz} = \frac{dl}{da} \circ \sigma'(z)$$

实际: 对于逐元素计算的激活函数, 导数也可逐元素计算

$$A: [n \times m], Z: [n \times m], A_{ij} = \sigma(Z_{ij})$$

$$\frac{dl}{dz} = \frac{dl}{dA} \circ \sigma'(Z)$$

5) 规则 1.5:

理论:

线性变换, $z = Wa + b, W \in R^{n \times m}$

$$a: [m \times 1], z: [p \times 1]$$

$$W: [p \times m], b: [p \times 1]$$

a 代表上一层神经元 (m 个)

z 代表上一层神经元 (p 个)

实际:

$$Z = AW + b^T, W \in R^{m \times p}$$

$$A: [n \times m], Z: [n \times p]$$

$$W: [m \times p], Z: [p \times 1]$$

n 代表样本量

A 的第 i 行代表第 i 个观测在上一层的神经元

Z 的第 i 行代表第 i 个观测在下一层的神经元

6) 规则 2: 对于线性变换, $z = Wa + b, W \in R^{n \times m}$

$$a = (a^1, \dots, a^m)^T, z = (z^1, \dots, z^m)^T$$

$$\text{假设 } z \text{ 的上游导数为 } \frac{dl}{dz} = \left(\frac{dl}{dz^1}, \dots, \frac{dl}{dz^n} \right)^T$$

$$\text{那么 } \frac{dl}{dw} = \frac{dl}{dz} a^T, \frac{dl}{db} = \frac{dl}{dz}, \frac{dl}{da} = W^T \frac{dl}{dz}$$

$$\frac{dl}{dw} = \frac{dl}{dz} a^T, \frac{dl}{db} = \frac{dl}{dz}, \frac{dl}{da} = W^T \frac{dl}{dz}$$

$$\frac{dl}{dw} = \frac{dl}{dz} a^T, \frac{dl}{db} = \frac{dl}{dz}, \frac{dl}{da} = W^T \frac{dl}{dz}$$

$$\frac{dl}{dw} = \frac{dl}{dz} a^T, \frac{dl}{db} = \frac{dl}{dz}, \frac{dl}{da} = W^T \frac{dl}{dz}$$

理论

$$\frac{dl}{dw} = \frac{dl}{dz} a^T, \frac{dl}{db} = \frac{dl}{dz}, \frac{dl}{da} = W^T \frac{dl}{dz}$$

$$\frac{dl}{db} = \frac{dl}{dz}$$

$$\frac{dl}{da} = W^T \frac{dl}{dz}$$

实际

$$\frac{dl}{dw} = A^T \frac{dl}{dz}$$

$$\frac{dl}{db} = \left(\frac{dl}{dz} \right)^T \mathbf{1}_n$$

$$\frac{dl}{da} = \frac{dl}{dz} W^T$$

6) 规则 3: 不要照搬数学表达式

实际编程中要考虑到数据的存储方式

lec5-module

自动微分与优化

有了自动微分以后, 对一大类机器学习的求解会非常方便。首先考虑一个简单的线性模型:

```
import math
import numpy as np
import torch
```

```
np.random.seed(123456)
```

```
torch.manual_seed(123456)
```

```
n = 100
```

```
p = 5
```

```
beta = torch.randn(p)
```

```
x = torch.randn(n, p)
```

```
y = torch.matmul(x, beta) + torch.randn(n) *
```

```
math.sqrt(0.1)
```

我们将利用简单的梯度下降对 β 进行优化。首先创建一个初始值并声明需要梯度：

```
bhat = torch.zeros(p)
bhat.requires_grad = True
```

编写损失函数：

```
def loss_fn(bhat, x, y):
    yhat = torch.matmul(x, bhat)
    return torch.mean(torch.square(y - yhat))
```

建立一个循环用来不断更新参数：

```
# 迭代次数
nepoch = 500
# 学习率, 即步长
learning_rate = 0.01
# 记录损失函数值
losses = []

for i in range(nepoch):
    loss = loss_fn(bhat, x, y) #给定 bhat 计算 Loss

    loss.backward() # 调用 backward 方法, 具有
    # bhat.grad 属性自动微分, 不需要手动写求导函数
    losses.append(loss.item())

    if i % 50 == 0:
        print(f"iteration {i}, loss = {loss.item()}, error = "
              f"{torch.mean(torch.square(bhat - beta))}")

    with torch.no_grad():
        bhat -= learning_rate * bhat.grad

#清空梯度项
bhat.grad = None

iteration 0, loss = 6.666600227355957, error = 1.300956130027771
iteration 50, loss = 0.8894939422607422, error = 0.17852994799613953
iteration 100, loss = 0.20490001142024994, error = 0.03156755864620209
iteration 150, loss = 0.11406504362821579, error = 0.00791635736823082
iteration 200, loss = 0.09939472377300262, error = 0.0029288791120052338
iteration 250, loss = 0.09633128345012665, error = 0.0016137382481247187
iteration 300, loss = 0.09553375095129013, error = 0.0012249473948031664
iteration 350, loss = 0.09529763460159302, error = 0.001107055344618857
iteration 400, loss = 0.09522351622581482, error = 0.0010728153865784407
iteration 450, loss = 0.09519967436790466, error = 0.0010642616543918848
```

loss.item()

该方法的功能是以标准的 Python 数字的形式来返回这个张量的值.这个方法

只能用于只包含一个元素的张量.对于其他的张量,请查看方法 tolist().

with 作用

在该模块下, 所有计算得出的 tensor 的 requires_grad 都自动设置为 False。即使一个 tensor (命名为 x) 的 requires_grad = True, 在 with torch.no_grad 计算, 由 x 得到的新 tensor (命名为 w-标量) requires_grad 也为 False, 且 grad_fn 也为 None,即不会对 w 求导。

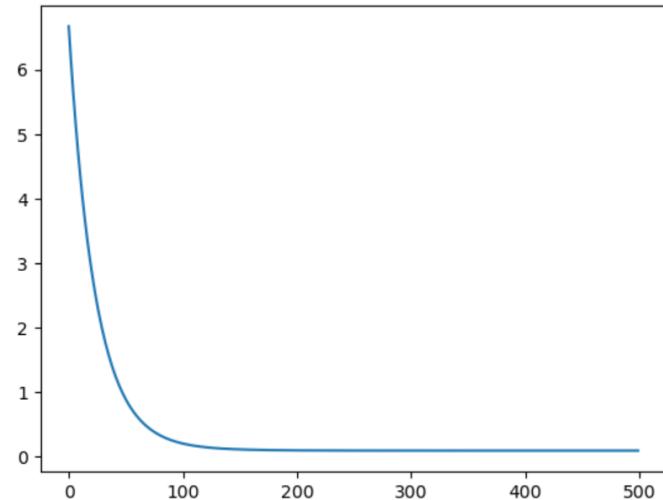
画出损失函数值的图像：

```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

```
import matplotlib.pyplot as plt
```

```
plt.plot(losses)
```

```
[<matplotlib.lines.Line2D at 0x2372c9d6640>]
```



比较真实和估计的 β

```
print(beta)
print(bhat)
tensor([1.8645, 0.4071, -1.1971, 0.3489, -1.1437])
tensor([1.8304, 0.4262, -1.1998, 0.3770, -1.0890], requires_grad=True)
```

模块化编程

PyTorch 中一种更常用的模型搭建和求解的方法是利用模块化编程, 即将所有的参数装进一个类中, 然后通过定义 forward() 函数来就散预测值

```
import torch.nn as nn
```

```
class MyModel(nn.Module): # 定义一个类, nn.Module 是
                           # pytorch 写好的, 从这里继承
```

```
def __init__(self, beta_dim):
    super(MyModel, self).__init__()
    self.bhat=nn.Parameter(torch.zeros(beta_dim))
    # 初始化成分装入 nn.Parameter, 自动具有求
    # 导功能, bhat 编程这个模块中的参数
    # self.w1 = nn.Parameter(torch.zeros(5, 3))
```

```
def forward(self, x):
```

```
yhat = torch.matmul(x, self.bhat)
```

```

        return yhat # 返回预测值

np.random.seed(123456)
torch.random.manual_seed(123456)

model = MyModel(beta_dim=p)
print(list(model.parameters())) # 使用这个方法，打印所有被定义的参数，更新迭代时不需要更新每个参数
[Parameter containing:
tensor([0., 0., 0., 0., 0.], requires_grad=True)]

```

nn.Module 函数详解

nn.Module 是所有网络模型结构的基类，无论是 pytorch 自带的模型，还是要自定义模型，都需要继承这个类。这个模块包含了很多子模块，如下所示，`_parameters` 存放的是模型的参数，`_buffers` 也存放的是模型的参数，但是是那些不需要更新的参数。带 hook 的都是钩子函数，详见钩子函数部分。

```

self._parameters = OrderedDict()
self._buffers = OrderedDict()
self._non_persistent_buffers_set = set()
self._backward_hooks = OrderedDict()
self._is_full_backward_hook = None
self._forward_hooks = OrderedDict()
self._forward_pre_hooks = OrderedDict()
self._state_dict_hooks = OrderedDict()
self._load_state_dict_pre_hooks = OrderedDict()
self._modules = OrderedDict()

```

此外，每一个模块还内置了一些常用的方法来帮助访问和操作网络。

```

load_state_dict() #加载模型权重参数
parameters() #读取所有参数
named_parameters() #读取参数名称和参数
buffers() #读取 self.named_buffers 中的参数
named_buffers() #读取 self.named_buffers 中的参数名称和参数
children() #读取模型中，所有的子模型
named_children() #读取子模型名称和子模型
requires_grad_() #设置模型是否开启梯度反向传播

```

Parameter 类

Parameter 是 Tensor 子类，所以继承了 Tensor 类的属性。例如 `data` 和 `grad` 属性，可以根据 `data` 来访问参数数值，用 `grad` 来访问参数梯度。

```

weight_0 = nn.Parameter(torch.randn(10,10))
print(weight_0.data)
print(weight_0.grad)
定义变量的时候，nn.Parameter 会被自动加入到参数列表中去

```

```

# 迭代次数
nepoch = 500

```

```

# 学习率，即步长
learning_rate = 0.01
# 记录损失函数值
losses = []

opt = torch.optim.SGD(model.parameters(), lr=learning_rate) # pytorch 自带优化器

for i in range(nepoch):
    yhat = model(x)
    loss = torch.mean(torch.square(y - yhat))

    opt.zero_grad()
    loss.backward()
    opt.step() # 更新参数进行简化

    losses.append(loss.item())

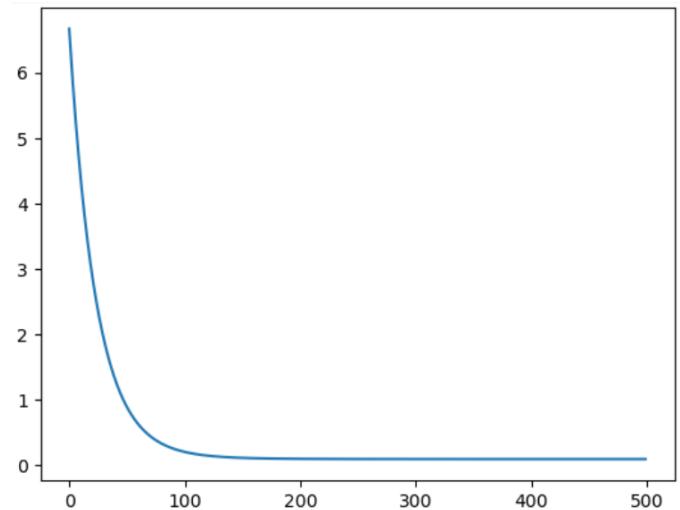
    if i % 50 == 0:
        print(f"iteration {i}, loss = {loss.item()}")
iteration 0, loss = 6.666600227355957
iteration 50, loss = 0.8894939422607422
iteration 100, loss = 0.20490001142024994
iteration 150, loss = 0.11406504362821579
iteration 200, loss = 0.09939472377300262
iteration 250, loss = 0.09633128345012665
iteration 300, loss = 0.09553375095129013
iteration 350, loss = 0.09529763460159302
iteration 400, loss = 0.09522351622581482
iteration 450, loss = 0.09519967436790466

```

```

print(list(model.parameters()))
plt.plot(losses)
[Parameter containing:
tensor([ 1.8304,  0.4262, -1.1998,  0.3770, -1.0890], requires_grad=True)]
[<matplotlib.lines.Line2D at 0x24c288fc80>]

```



模块化编程的好处是模型的结构可以随时发生调整，而后续的循环、优化等部分的代码可以保持不变。例如，假设我们此时希望加入一个截距项，可以把模型修改为如下形式：

```
class MyModel(nn.Module): # 统一流程化
    def __init__(self, beta_dim):
        super(MyModel, self).__init__()
        self.bhat = nn.Parameter(torch.randn(beta_dim))
        self.b0 = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        yhat = torch.matmul(x, self.bhat) + self.b0
        return yhat

np.random.seed(123456)
torch.random.manual_seed(123456)

model = MyModel(beta_dim=p)
print(list(model.parameters()))
[Parameter containing:
tensor([ 1.8645,  0.4071, -1.1971,  0.3489, -1.1437], requires_grad=True),
Parameter containing:
tensor([0.], requires_grad=True)]
```

接下来模型训练的代码可以不做任何改变：

```
# 迭代次数
nepoch = 500
# 学习率
learning_rate = 0.01
# 计算损失函数值
losses = []

opt = torch.optim.SGD(model.parameters(), lr=learning_rate)
for i in range(nepoch):
    yhat = model(x)
    loss = torch.mean(torch.square(y - yhat))

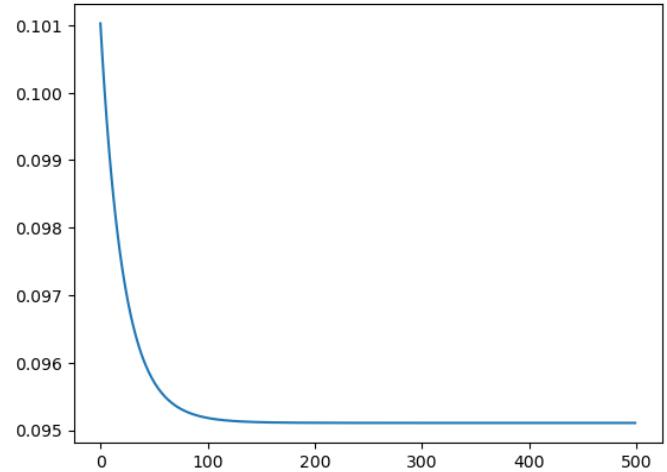
    opt.zero_grad()
    loss.backward()
    opt.step()

    losses.append(loss.item())

    if i % 50 == 0:
        print(f"iteration {i}, loss = {loss.item()}")

print(list(model.parameters()))
plt.plot(losses)
iteration 0, loss = 0.10102550685405731
```

```
iteration 50, loss = 0.09571389853954315
iteration 100, loss = 0.09518022835254669
iteration 150, loss = 0.0951167494058609
iteration 200, loss = 0.09510793536901474
iteration 250, loss = 0.09510650485754013
iteration 300, loss = 0.09510625153779984
iteration 350, loss = 0.09510622173547745
iteration 400, loss = 0.09510618448257446
iteration 450, loss = 0.09510619193315506
[Parameter containing:
tensor([ 1.8301,  0.4255, -1.2002,  0.3755, -1.0897], requires_grad=True),
Parameter containing:
tensor([0.0094], requires_grad=True)]
[<matplotlib.lines.Line2D at 0x24c28973760>]
```



DL-lec6

一、卷积神经网络

1、卷积的概念

1) 先看一道初中数学题

考虑两个多项式, $p(x) = 3 - 2x + x^2$, $q(x) = -5 + 4x$ 问 $r(x) = p(x) \cdot q(x)$ 的展开式是什么?

2) 再看一道“大学”算法题

a. 一般地, 用一个向量表示多项式的系数

如 $v = (1, 0, 2)'$ 表示 $1 + 2x^2$, $w = (0, 1, 2, 3)'$ 表示 $x + 2x^2 + 3x^3$

b. 给定两个任意的多项式系数 v 和 w , 求多项式成绩的系数向量 r

c. 先说结论

a) R

➤ $v = c(3, -2, 1)$

➤ $w = c(-5, 4)$

➤ `convolve(v, rev(w), type = "open")`

[1] -15 22 -13 4

b) Python

`import numpy as np`

`v = [3, -2, 1]`

`w = [-5, 4]`

`np.convolve(v, w)`

array([-15, 22, -13, 4])

3)

convolve

英 [kən'vəlv] 美 [kən've:lv]

v. 使卷曲, 使缠绕; 卷积

[第三人称单数 convolves 现在分词 convolving 过去式 convolved 过去分词 convolved]

同近义词 同根词

词根: **convolve**

adj.

convoluted 复杂的; 费解的; 旋绕的

n.

convolution [数] 卷积; 回旋; 盘旋; 卷绕

v.

convoluted 盘绕; 缠绕 (convolute的过去分词)

2. 一维卷积

$$p(x) = 3 - 2x + x^2, \quad q(x) = -5 + 4x$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$w \quad \begin{matrix} 1 & x \\ -5 & 4 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$w \quad \begin{matrix} 1 & x \\ -5 & 4 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$\downarrow$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix} \rightarrow$$

$$\downarrow$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$r \quad \begin{matrix} 1 \\ -15 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w)$$

$$r \quad \begin{matrix} 1 & x \\ -15 & 22 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix} \rightarrow$$

$$r \quad \begin{matrix} 1 & x \\ -15 & 22 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$r \quad \begin{matrix} 1 & x \\ -15 & 22 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

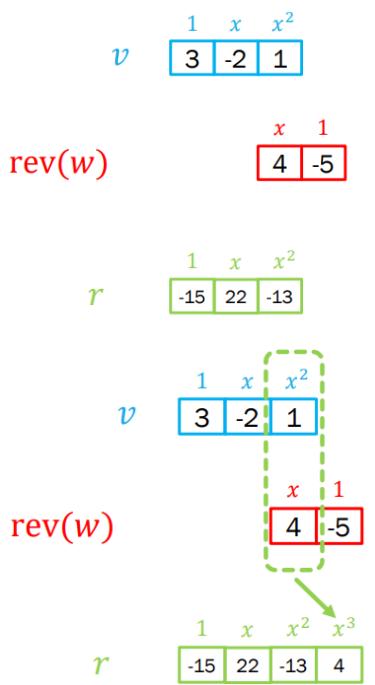
$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix}$$

$$r \quad \begin{matrix} 1 & x & x^2 \\ -15 & 22 & -13 \end{matrix}$$

$$\nu \quad \begin{matrix} 1 & x & x^2 \\ 3 & -2 & 1 \end{matrix}$$

$$rev(w) \quad \begin{matrix} x & 1 \\ 4 & -5 \end{matrix} \rightarrow$$

$$r \quad \begin{matrix} 1 & x & x^2 \\ -15 & 22 & -13 \end{matrix}$$



3、数学定义

$$v = (v_1, \dots, v_m)', w = (w_1, \dots, w_n)'$$

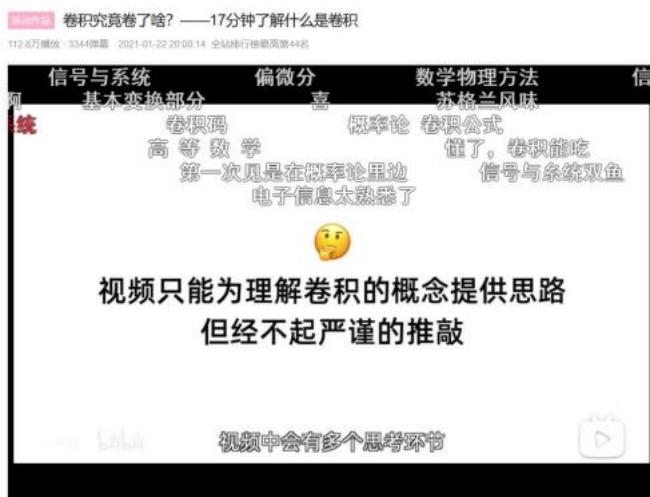
$$r = v * w$$

$$r_i = \sum_{k=-\infty}^{\infty} v_k w_{i-k+1}$$

没有定义的元素用 0 代替

4、延伸学习

<https://www.bilibili.com/video/BV1JX4y1K7Dr/>



5、二维卷积

1	2	3	4
5	6	7	8
9	-1	-2	-3
-4	-5	-6	-7

0.1	0.2	0.3
0.4	0.5	0.6
0.7	0.8	0.9

卷积核
(convolutional kernel) /
滤波器
(filter)

W

0	0	0	0	0	0
1	2	3	4	0	0
5	6	7	8	0	0
9	-1	-2	-3	0	0
-4	-5	-6	-7	0	0
0	0	0	0	0	0

V

R

0	0	0	0	0	0
0.9	0.8	0.7	0	0	0
0.6	0.5	0.4	0	0	0
0.3	0.2	0.1	0	0	0
0	9	-1	-2	-3	0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

V

R

0	0	0	0	0	0
0	0.9	0.8	0.7	0	0
0	1	2	3	4	0
0	0.6	0.5	0.4	0	0
0	5	6	7	8	0
0	0.3	0.2	0.1	0	0
0	9	-1	-2	-3	0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

V

R

0	0	0	0	0	0
0	0.9	0.8	0.7	0	0
0	1	2	3	4	0
0	0.6	0.5	0.4	0	0
0	5	6	7	8	0
0	0.3	0.2	0.1	0	0
0	9	-1	-2	-3	0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

V

R

0	0	0	0	0	0
0	0.9	0.8	0.7	0	0
0	1	2	3	4	0
0	0.6	0.5	0.4	0	0
0	5	6	7	8	0
0	0.3	0.2	0.1	0	0
0	9	-1	-2	-3	0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

V

V

0	0	0	0	0	0
0	1	2	3	4	0
0.9	0.8	0.7			0
0	5	6	7	8	0
0.6	0.5	0.4			0
0	9	-1	-2	-3	0
0.3	0.2	0.1			0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

R

V

0	0	0	0	0	0
0	1	2	3	4	0
0	0.9	0.8	0.7		0
0	5	6	7	8	0
0	0.6	0.5	0.4		0
0	9	-1	-2	-3	0
0	0.3	0.2	0.1		0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

R

V

0	0	0	0	0	0
0	1	2	3	4	0
0	0.9	0.8	0.7		0
0	5	6	7	8	0
0	0.6	0.5	0.4		0
0	9	-1	-2	-3	0
0	0.3	0.2	0.1		0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

R

V

0	0	0	0	0	0
0	1	2	3	4	0
0	0.9	0.8	0.7		0
0	5	6	7	8	0
0	0.6	0.5	0.4		0
0	9	-1	-2	-3	0
0	0.3	0.2	0.1		0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

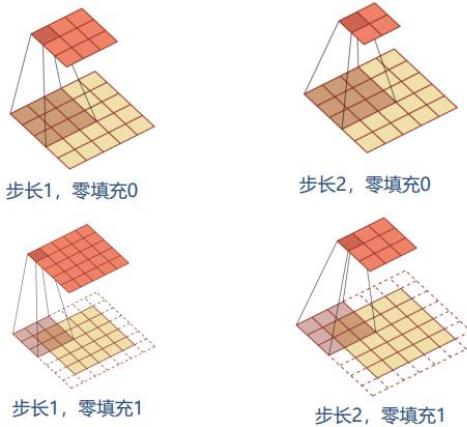
R

V

0	0	0	0	0	0
0	1	2	3	4	0
0	0.9	0.8	0.7		0
0	5	6	7	8	0
0	0.6	0.5	0.4		0
0	9	-1	-2	-3	0
0	0.3	0.2	0.1		0
0	-4	-5	-6	-7	0
0	0	0	0	0	0

R

V



二、卷积有什么用？

1、卷积的应用

1) 与深度学习直接相关

提取图片特征

2) 与概率统计直接相关

滑动平均

核密度估计

随机变量之和的分布

3) 其他

信号处理

整数乘法、多项式乘法

2、卷积与图像处理

卷积是提取图形特征的一种工具

参见 lec6-conv2d.ipynb

三、卷积神经网络

1、前馈神经网络

1) 实现简单、通用近似、计算高效

2) 为什么还不够好?

a. 数据维度高时参数很多

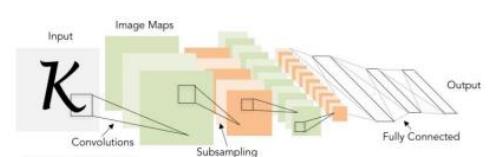
b. 对于图片的缩放、平移、旋转等操作较敏感

c. 难以提取图片的局部信息

2、卷积神经网络

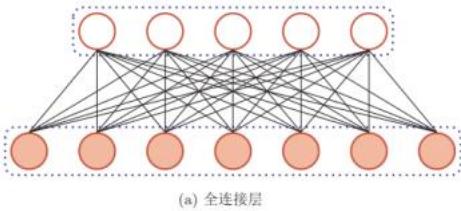
1) Convolutional neural network

CNN/ConvNet

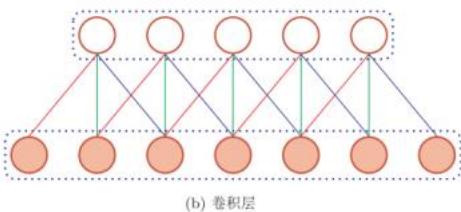


2) 简而言之，CNN 是利用了卷积操作的神经网络结构用卷积层替换全连接层

a. 二维卷积有不同的类型
取决于步长和补零的数量
输出大小相应变化



(a) 全连接层

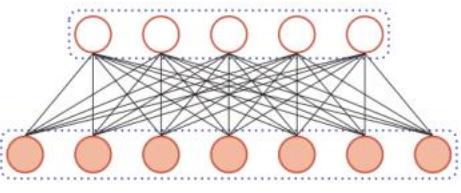


(b) 卷积层

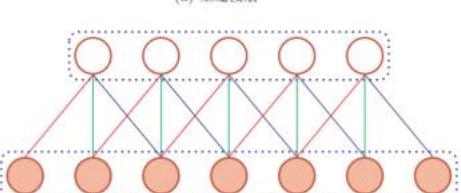
3) 卷积层

a. 局部连接

b. 权重共享



? 个参数



? 个参数

c. 卷积核作为待估参数（可学习参数）

输入：图片数据 $[N \times H_{in} \times W_{in} \times C_{in}]$

输出：特征映射 (feature map) $[N \times H_{out} \times W_{out} \times C_{out}]$

某一层输出的特征映射（经过激活函数作用后）可以作为下一层的输入

d. 注意事项

a) 对于原始图片， C_{in} 通常有明确的实际意义

b) 如灰度图片 $C_{in} = 1$ （单个通道）

c) 彩色图片有红绿蓝 (RGB) 三个通道， $C_{in} = 3$

d) 经过处理后的特征映射 C_{out} 可以任意指定，只具有抽象的意义

e) 不同的软件框架可能采用不同的数据格式

Tensorflow 采用 $[N \times H \times W \times C]$

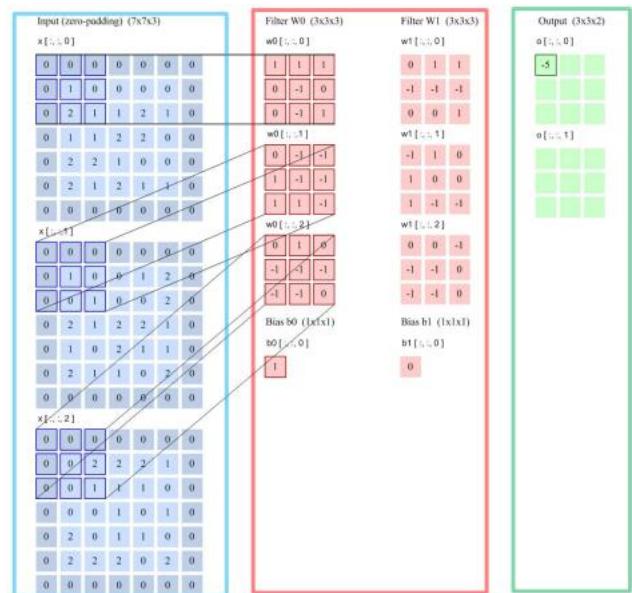
PyTorch 采用 $[N \times C \times H \times W]$

MXNet 可以设定参数在两种格式中选择

f) 将 $[N \times H_{in} \times W_{in} \times C_{in}]$ 的输入变换为 $[N \times H_{out} \times W_{out} \times C_{out}]$ 的输出，共需要 $C_{in} \times C_{out}$ 个卷积核

g) 卷积核的类型和大小决定了 $H_{in} \times W_{in}$ 和 $H_{out} \times W_{out}$ 之间的对应关系

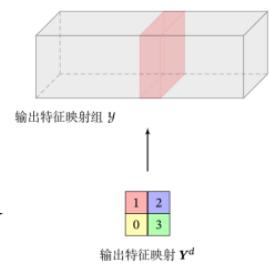
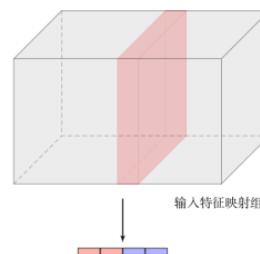
e. 一般形式



4) 汇聚层

a. 早期的 CNN 中通常还会加入一些汇聚层 (pooling layer, 或称池化层)

b. 用来快速减少神经元的数目



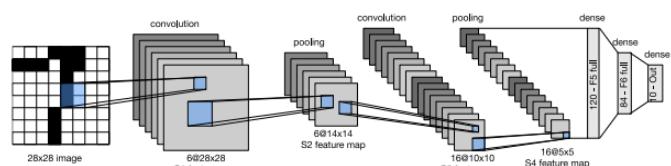
c. 当前，汇聚层有被步长 > 1 的卷积层替代的趋势

5) LeNet-5

a. LeNet-5 是一个早期的 CNN 架构

b. 结合了卷积层、汇聚层和全连接层

c. 结合手写识别数据 MNIST，成为了经典的 CNN 入门模型



lec6-conv2d

```
import numpy as np
v = [3, -2, 1]
w = [-5, 4]
np.convolve(v, w)
array([-15, 22, -13, 4])
```

```
from PIL import Image
im = Image.open("sufe.jpg")
im
```



```
import numpy as np
im_arr = np.array(im)
print(im_arr.shape)
(748, 1197, 3)
```

```
print("Red channel:\n")
print(im_arr[:, :, 0])

print("\nGreen channel:\n")
print(im_arr[:, :, 1])

print("\nBlue channel:\n")
print(im_arr[:, :, 2])
```

Red channel:

```
[[68 69 70 ... 88 88 88]
 [68 69 71 ... 88 88 88]
 [69 70 71 ... 88 88 88]
 ...
 [95 68 84 ... 86 84 83]
 [87 65 75 ... 86 84 84]
 [83 67 70 ... 88 86 84]]
```

Green channel:

```
[[126 127 128 ... 138 138 138]
 [126 127 129 ... 138 138 138]
 [127 128 129 ... 138 138 138]
 ...
 [91 64 80 ... 31 34 36]
 [83 61 71 ... 31 34 36]
 [78 62 67 ... 31 34 36]]
```

Blue channel:

```
[[201 202 203 ... 209 209 209]
 [200 201 203 ... 209 209 209]
```

```
[201 202 203 ... 209 209 209]
```

...

```
[ 90  63  79 ...  24  25  26]
```

```
[100  76  86 ...  24  25  26]
```

```
[101  85  88 ...  22  23  24]]
```

```
# Convert to a grey image
# Use the formula in
https://stackoverflow.com/a/12201744
#  $Y = 0.2989 R + 0.5870 G + 0.1130 B$ 
im_grey = 0.2989 * im_arr[:, :, 0] + 0.5870 * im_arr[:, :, 1]
+ 0.1140 * im_arr[:, :, 2]
print(im_grey.shape)
(748, 1197)
```

`Image.fromarray(np.uint8(im_grey))`



```
# The code is adapted from
https://github.com/ashushekhar/image-convolution-from-scratch
```

```
def convolve2d_channel(image, kernel):
```

""

This function takes an image and a kernel and returns the convolution of them.

:param image: a numpy array of size [image_height, image_width].

:param kernel: a numpy array of size [kernel_height, kernel_width]

:return: a numpy array of size [image_height, image_width] (convolution output)

"""

Flip the kernel

```
kernel = np.flip(kernel, axis=(0, 1))
```

Various dimensions

```
img_w = image.shape[1]
```

```
img_h = image.shape[0]
```

```

ker_w = kernel.shape[1]
ker_h = kernel.shape[0]
out_w = img_w - ker_w + 1
out_h = img_h - ker_h + 1
# Convolution output
output = np.empty((out_h, out_w))
# Loop over pixels
for j in range(out_w):
    for i in range(out_h):
        out[i, j] = (kernel * image[i:(i + ker_h), j:(j + ker_w)]).sum()
return output

```

```

def convolve2d(image, kernel):
    if image.ndim == 2:
        return convolve2d_channel(image, kernel)
    elif image.ndim == 3:
        res = [convolve2d_channel(image[:, :, k], kernel)
               for k in range(image.shape[2])]
    else:
        raise Exception("invalid image shape")

```

np.flip

翻转也称镜像，是指将图像沿轴线进行轴对称变换。水平镜像是将图像沿垂直中轴线进行左右翻转，垂直镜像是将图像沿水平中轴线进行上下翻转，水平垂直镜像是水平镜像和垂直镜像的叠加。

```
cv.flip(src, flipCode[, dst]) -> dst
```

np.empty

```
numpy.empty(shape, dtype=float, order='c', like=None)
```

```
*
```

np.empty()是依据给定形状和类型(shape,[dtype, order])返回一个新的空数组。由于 np.empty()返回一个随机元素的矩阵，所以使用的时候需要手工把每一个值重新定义，否则该值是接近零的随机数。

```

kernel = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
kernel
array([[-1, -1, -1],
       [-1, 8, -1],
       [-1, -1, -1]])

```

```

im1 = convolve2d(im_grey, kernel)
im1.shape
(746, 1195)

```

```

def normalize(x):
    xmin = np.quantile(x, 0.01)
    xmax = np.quantile(x, 0.99)
    x = np.clip(x, xmin, xmax)

```

```
return np.uint8(255.0 * (x - xmin) / (xmax - xmin))
```

```
Image.fromarray(normalize(im1))
```



np.quantile

分位数(Quantile)是一个统计学概念,用于描述一个数据集中的位置特性。在一个有序的数据集中,第 q 个 q 分位数是一个值,使得数据集中至少有 $q \times 100\%$ 的数据点小于或等于这个值,并且至少有 $(1 - q) \times 100\%$ 的数据点大于或等于这个值。

```
logo = Image.open("logo.png")
```

```
logo
```



```

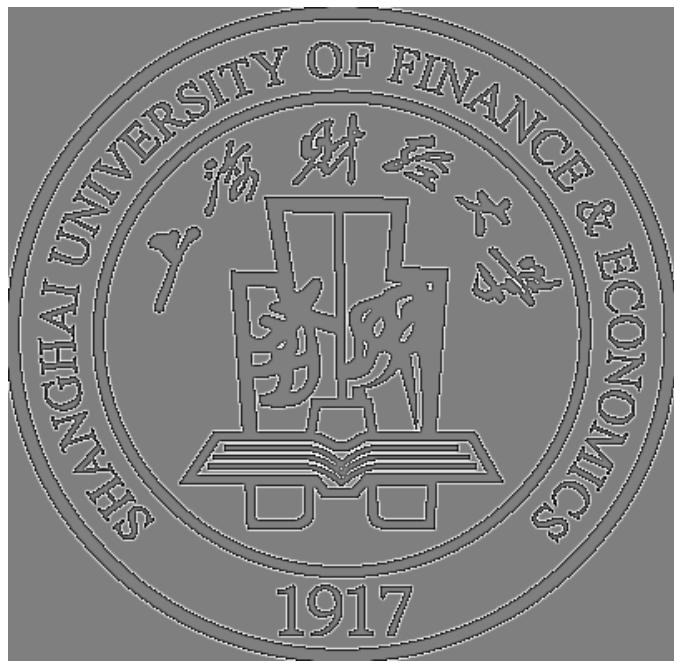
logo = np.array(logo)
print(logo.shape)
logo = 0.2989 * logo[:, :, 0] + 0.5870 * logo[:, :, 1] +
0.1140 * logo[:, :, 2]
print(logo.shape)
Image.fromarray(np.uint8(logo))
(395, 405, 3)
(395, 405)

```



```
logo = convolve(logo, kernel)
print(logo.shape)
Image.fromarray(normalize(logo))

(393, 403)
```



```
kernel = np.ones((9, 9)) / 81.0
print(np.round(kernel, 3))
```

```
im2 = convolve2d(im_arr, kernel)
Image.fromarray(np.uint8(im2))

[[0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
 [0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]]
```

```
[0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
[0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]
[0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012]]
```



```
gr = np.linspace(-3, 3, num=9)
den = np.exp(-0.2 * gr * gr)
den = den.reshape(-1, 1) / den.sum()
kernel = den.dot(den.T)
print(np.round(kernel, 3))
```

```
im3 = convolve2d(im_arr, kernel)
Image.fromarray(np.uint8(im3))

[[0.001 0.002 0.004 0.006 0.006 0.006 0.004 0.002 0.001]
 [0.002 0.005 0.009 0.012 0.014 0.012 0.009 0.005 0.002]
 [0.004 0.009 0.016 0.022 0.024 0.022 0.016 0.009 0.004]
 [0.006 0.012 0.022 0.03 0.034 0.03 0.022 0.012 0.006]
 [0.006 0.014 0.024 0.034 0.038 0.034 0.024 0.014 0.006]
 [0.006 0.012 0.022 0.03 0.034 0.03 0.022 0.012 0.006]
 [0.004 0.009 0.016 0.022 0.024 0.022 0.016 0.009 0.004]
 [0.002 0.005 0.009 0.012 0.014 0.012 0.009 0.005 0.002]
 [0.001 0.002 0.004 0.006 0.006 0.006 0.004 0.002 0.001]]
```



```
gr = np.linspace(-5, 5, num=25)
den = np.exp(-0.1 * gr * gr)
den = den.reshape(-1, 1) / den.sum()
kernel = den.dot(den.T)
```

```
im4 = convolve2d(im_arr, kernel)
Image.fromarray(np.uint8(im4))
```



lec7-convert

本次作业将练习卷积神经网络，利用卷积层和全连接层实现手写数字的识别。

1、目标

通过对 MNIST 数据进行训练，构建一个简单地图像分类模型，对图片中的数字进行识别。你将利用该模型对自己真实手写出的数字进行预测，观察模型效果。

2、主要步骤

1. 获取数据
2. 定义模型结构
3. 创建模型类
4. 定义损失函数
5. 编写训练循环
6. 实施预测

2.1 获取数据

我们使用知名的 MNIST 数据集，它可以从 PyTorch 中利用工具函数下载得到。原始的 MNIST 数据训练集大小为 60000，我们随机抽取其中的 10000 个观测进行简单的训练，以及 10 个观测进行预测展示。以下函数会在当前目录建立一个名为 data 的文件夹，其中会包含下载得到的数据集。

**注意：请在任何程序的最开始加上随机数种子的设置。
请保持这一习惯。**

```
import numpy as np
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader

np.random.seed(123456)
torch.manual_seed(123456)

mnist = datasets.MNIST(
    root = "data"
    train = True,
    download = True,
```

```
    transform = ToTensor()
```

```
)  
loader = DataLoader(mnist, batch_size=10010,  
shuffle=True)
```

我们一次性取出随机抽取到的 10010 个观测，其中 x 是图片数据，y 是图片对应的数字。

```
x, y = next(iter(loader))
```

一个习惯性动作是查看数据的大小和维度。

```
print(x.shape)  
print(y.shape)  
torch.Size([10010, 1, 28, 28])  
torch.Size([10010])
```

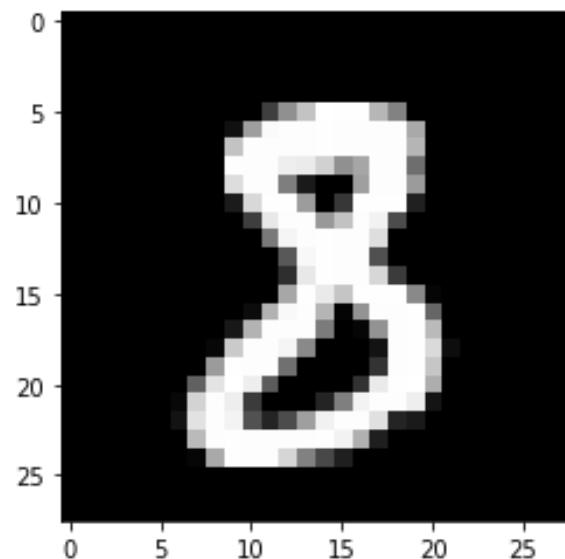
将最后的 10 张图片取为测试集：

```
xtest = x[-10:]  
ytest = y[-10:]  
x = x[:-10]  
y = y[:-10]  
print(x.shape)  
print(xtest.shape)  
torch.Size([10000, 1, 28, 28])  
torch.Size([10, 1, 28, 28])
```

我们可以利用下面的函数展示图片的内容。如选择第一张测试图片，先将其转换成 Numpy 数组，再绘制图形：

```
import matplotlib.pyplot as plt
```

```
img = xtest[0].squeeze().cpu().numpy()  
print(img.shape)  
plt.imshow(img, cmap="gray")  
plt.show()  
(28, 28)
```



2.2 定义模型结构

作为演示，我们创建一个卷积层、一个汇聚层和一个全连接层。所有隐藏层的函数细节都可以在官方文档 (<https://pytorch.org/docs/stable/nn.html>) 中按分类找

到。每一个隐藏层本质上都是将一个数组转换成另一个数组的函数，因此为了确认编写的模型是正确的，可以先用一个小数据进行测试，观察输入和输出的维度。例如，我们先取出前 6 个观测，此时输入的维度是 [6,1,28,28]：

```
ns = 6
smallx = x[0:ns]
smally = y[0:ns]
print(smallx.shape)
print(smally.shape)
torch.Size([6, 1, 28, 28])
torch.Size([6])
```

接下来创建第 1 个卷积层，并测试输出的维度。注意到我们可以直接将隐藏层当成一个函数来调用。

```
conv1 = torch.nn.Conv2d(in_channels=1, out_channels=20, kernel_size=5, stride=2)
res1 = conv1(smallx)
print(res1.shape)
torch.Size([6, 20, 12, 12])
```

可以看到，输出的维度为 [20, 12, 12]（不包含第 1 位的数据批次维度）。

接下来我们创建一个最大汇聚层：

```
pool1 = torch.nn.MaxPool2d(kernel_size=2)
res2 = pool1(res1)
print(res2.shape)
assert res2.shape == (ns, 20, 6, 6), "pool 输出形状不对"
torch.Size([6, 20, 6, 6])
```

可以看出此时的输出维度变成了 [20, 6, 6]。

最后我们将得到的特征拉直，并输出 10 维的向量，用来计算每个类的概率预测：

```
# 输入: 20*6*6=720
# 输出: 10
fc1 = torch.nn.Linear(in_features=720, out_features=10)
res3 = fc1(torch.flatten(res2, start_dim=1))
print(res3.shape)
assert res3.shape == (ns, 10), "fc1 输出形状不对"
torch.Size([6, 10])
```

2.3 创建模型类

在确保隐藏层维度都正确后，将所有隐藏层封装到一个模型类中，其中模型结构在 `__init__()` 中定义，具体的计算过程在 `forward()` 中实现。此时需要加入激活函数。

```
class MyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(in_channels=1,
                                  out_channels=20, kernel_size=5, stride=2)
        self.pool1 = torch.nn.MaxPool2d(kernel_size=2)
```

```
        self.fc1 = torch.nn.Linear(in_features=720, out_features=10)

    def forward(self, x):
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.pool1(x)
        x = torch.flatten(x, start_dim=1)
        x = self.fc1(x)
        x = torch.nn.functional.softmax(x, dim=1)
        return x
```

再次测试输入输出的维度是否正确。如果模型编写正确，输出的维度应该是 [6,10]，且输出的结果为 0 到 1 之间的概率值。

```
np.random.seed(123)
torch.manual_seed(123)

model = MyModel()
pred = model(smallx)
print(pred.shape)
print()
print(pred)
print()
print(torch.sum(pred, dim=1))
torch.Size([6, 10])

tensor([[0.1019, 0.1074, 0.1031, 0.1050, 0.0917, 0.1077, 0.1088, 0.0985, 0.0936, 0.0823],
        [0.1053, 0.0975, 0.1009, 0.1096, 0.0899, 0.1213, 0.1105, 0.0947, 0.0856, 0.0847],
        [0.0971, 0.1062, 0.1118, 0.1027, 0.0784, 0.1124, 0.1005, 0.1123, 0.0808, 0.0979],
        [0.1053, 0.1318, 0.0929, 0.1080, 0.0824, 0.1061, 0.0934, 0.1020, 0.0958, 0.0823],
        [0.1065, 0.0973, 0.1026, 0.1064, 0.0962, 0.1096, 0.1056, 0.1046, 0.0833, 0.0879],
        [0.1014, 0.1099, 0.1216, 0.1017, 0.0796, 0.1058, 0.0993, 0.1093, 0.0770, 0.0943]],
       grad_fn=<SoftmaxBackward0>

tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000], grad_fn=<SumBackward1>)
```

`pred` 的每一行加总为 1，其中每一个元素代表对应类别的预测概率。

我们还可以直接打印模型对象，观察隐藏层的结构

```
print(model)
MyModel(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(2, 2))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
  ceil_mode=False)
  (fc1): Linear(in_features=720, out_features=10, bias=True)
)
```

2.4 定义损失函数

对于分类问题，损失函数通常选取负对数似然函数。在 PyTorch 中，可以使用 `torch.nn.NLLoss` 来完成计算。其

用法是先定义一个损失函数对象，然后在预测值和真实标签上调用该函数对象。注意：损失函数对象的第一个参数是预测概率的对数值，第二个参数是真实的标签。

文档说明：

(<https://pytorch.org/docs/stable/generated/torch.nn.NLLLoss.html>)。

```
lossfn = torch.nn.NLLLoss()  
lossfn(torch.log(pred), smally)  
tensor(2.3334, grad_fn=<NLLLossBackward0>)
```

2.5 编写训练循环

利用课上介绍的循环模板和代码示例，对模型进行迭代训练。对于本数据，选取 mini-batch 大小为 200，共遍历数据 10 遍，优化器选为 Adam，学习率为 0.001。记录每个 mini-batch 下的损失函数值存放在列表 losses_sgd 中，然后画出损失函数的曲线。

```
nepoch = 30  
batch_size = 200  
lr = 0.001  
  
np.random.seed(123)  
torch.manual_seed(123)  
  
model = MyModel()  
losses = []  
opt = torch.optim.Adam(model.parameters(), lr=lr)  
  
n = x.shape[0]  
obs_id=np.arange(n) # [0, 1, ..., n-1]  
# Run the whole data set 'nepoch' times  
for i in range(nepoch):  
    # Shuffle observation IDs  
    np.random.shuffle(obs_id)  
  
    # Update on mini-batches  
    for j in range(0, n, batch_size):  
        # Create mini-batch  
        x_mini_batch = x[obs_id[j:(j + batch_size)]]  
        y_mini_batch = y[obs_id[j :(j + batch_size)]]  
        # Compute loss  
        pred = model(x_mini_batch)  
        lossfn = torch.nn.NLLLoss()  
        loss = lossfn(torch.log(pred), y_mini_batch)  
        # Compute gradient and update parameters  
        opt.zero_grad()  
        loss.backward()  
        opt.step()  
        losses.append(loss.item())  
  
        if (j // batch_size) % 20 == 0:
```

```
print(f"epoch {i}, batch {j // batch_size},  
      loss = {loss.item()}")  
epoch 0, batch 0, loss = 2.312232255935669  
epoch 0, batch 20, loss = 1.7555195093154907  
epoch 0, batch 40, loss = 1.10931396484375  
epoch 1, batch 0, loss = 0.7931170463562012  
epoch 1, batch 20, loss = 0.6293395757675171  
epoch 1, batch 40, loss = 0.49604567885398865  
epoch 2, batch 0, loss = 0.5026378035545349  
epoch 2, batch 20, loss = 0.4057575464248657  
epoch 2, batch 40, loss = 0.38254618644714355  
epoch 3, batch 0, loss = 0.3363669514656067  
epoch 3, batch 20, loss = 0.2684660255908966  
epoch 3, batch 40, loss = 0.27938300371170044  
epoch 4, batch 0, loss = 0.2825264036655426  
epoch 4, batch 20, loss = 0.22186315059661865  
epoch 4, batch 40, loss = 0.2930329442024231  
epoch 5, batch 0, loss = 0.25070205330848694  
epoch 5, batch 20, loss = 0.21505478024482727  
epoch 5, batch 40, loss = 0.16764219105243683  
epoch 6, batch 0, loss = 0.2063990831375122  
epoch 6, batch 20, loss = 0.178342804312706  
epoch 6, batch 40, loss = 0.22663486003875732  
epoch 7, batch 0, loss = 0.23608915507793427  
epoch 7, batch 20, loss = 0.16675056517124176  
epoch 7, batch 40, loss = 0.17813687026500702  
epoch 8, batch 0, loss = 0.27633199095726013  
epoch 8, batch 20, loss = 0.17944306135177612  
epoch 8, batch 40, loss = 0.15992097556591034  
epoch 9, batch 0, loss = 0.28502729535102844  
epoch 9, batch 20, loss = 0.1843848079442978  
epoch 9, batch 40, loss = 0.17882069945335388  
epoch 10, batch 0, loss = 0.13444605469703674  
epoch 10, batch 20, loss = 0.1853785365819931  
epoch 10, batch 40, loss = 0.15365129709243774  
epoch 11, batch 0, loss = 0.2231629192829132  
epoch 11, batch 20, loss = 0.10797496140003204  
epoch 11, batch 40, loss = 0.17474615573883057  
epoch 12, batch 0, loss = 0.12791325151920319  
epoch 12, batch 20, loss = 0.07508761435747147  
epoch 12, batch 40, loss = 0.17308630049228668  
epoch 13, batch 0, loss = 0.13825419545173645  
epoch 13, batch 20, loss = 0.11354640871286392  
epoch 13, batch 40, loss = 0.13732516765594482  
epoch 14, batch 0, loss = 0.1437186300754547  
epoch 14, batch 20, loss = 0.08950509130954742  
epoch 14, batch 40, loss = 0.12481299042701721  
epoch 15, batch 0, loss = 0.12439898401498795  
epoch 15, batch 20, loss = 0.1215452179312706
```

```

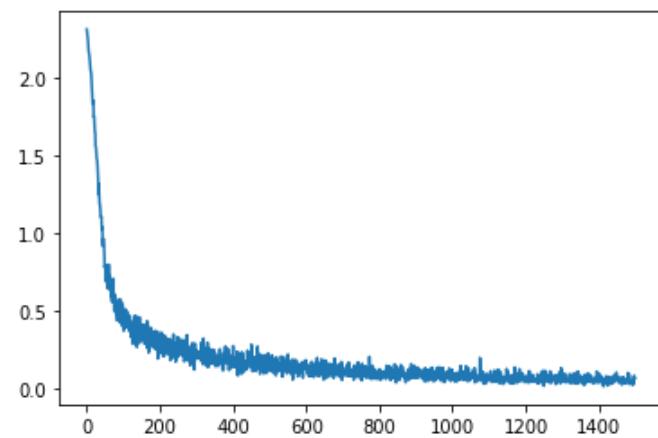
epoch 15, batch 40, loss = 0.0775536522269249
epoch 16, batch 0, loss = 0.12715835869312286
epoch 16, batch 20, loss = 0.14986908435821533
epoch 16, batch 40, loss = 0.07287914305925369
epoch 17, batch 0, loss = 0.07606033235788345
epoch 17, batch 20, loss = 0.0694250538945198
epoch 17, batch 40, loss = 0.07662598788738251
epoch 18, batch 0, loss = 0.06969526410102844
epoch 18, batch 20, loss = 0.06844912469387054
epoch 18, batch 40, loss = 0.07449617236852646
epoch 19, batch 0, loss = 0.07136600464582443
epoch 19, batch 20, loss = 0.11972198635339737
epoch 19, batch 40, loss = 0.06785032153129578
epoch 20, batch 0, loss = 0.07200051099061966
epoch 20, batch 20, loss = 0.11937542259693146
epoch 20, batch 40, loss = 0.05456458032131195
epoch 21, batch 0, loss = 0.06120741739869118
epoch 21, batch 20, loss = 0.05998872220516205
epoch 21, batch 40, loss = 0.06688737124204636
epoch 22, batch 0, loss = 0.08760158717632294
epoch 22, batch 20, loss = 0.07654212415218353
epoch 22, batch 40, loss = 0.05038575828075409
epoch 23, batch 0, loss = 0.04823886230587959
epoch 23, batch 20, loss = 0.13898977637290955
epoch 23, batch 40, loss = 0.10197748243808746
epoch 24, batch 0, loss = 0.044452033936977386
epoch 24, batch 20, loss = 0.08776171505451202
epoch 24, batch 40, loss = 0.08801954984664917
epoch 25, batch 0, loss = 0.018981067463755608
epoch 25, batch 20, loss = 0.03800363093614578
epoch 25, batch 40, loss = 0.058318842202425
epoch 26, batch 0, loss = 0.049480944871902466
epoch 26, batch 20, loss = 0.09590239822864532
epoch 26, batch 40, loss = 0.07519640028476715
epoch 27, batch 0, loss = 0.08464951813220978
epoch 27, batch 20, loss = 0.05308246612548828
epoch 27, batch 40, loss = 0.05427286773920059
epoch 28, batch 0, loss = 0.0792817622423172
epoch 28, batch 20, loss = 0.055106207728385925
epoch 28, batch 40, loss = 0.0491580069065094
epoch 29, batch 0, loss = 0.05872231349349022
epoch 29, batch 20, loss = 0.034123703837394714
epoch 29, batch 40, loss = 0.029261356219649315

```

```

plt.plot(losses)
[<matplotlib.lines.Line2D at 0x2b9e1414c10>]

```



2.6 实施预测

为了验证模型的效果，我们对 10 个测试观测（即之前生成的 testx）进行预测。

```

ytest = model(xtest)
print(np.round(ytest.detach().cpu().numpy(), 3))
print(ytest)
[[0.  0.  0.001 0.035 0.   0.  0.   0.  0.963 0. ]
 [0.  0.  0.   0.   0.   0.002 0.   0.024 0.   0.973]
 [0.  0.  1.   0.   0.   0.   0.   0.   0.   0. ]
 [0.  0.  1.   0.   0.   0.   0.   0.   0.   0. ]
 [0.  1.   0.   0.   0.   0.   0.   0.   0.   0. ]
 [0.  0.   0.   0.001 0.   0.982 0.   0.   0.   0.017]
 [0.978 0.   0.005 0.   0.   0.   0.001 0.   0.016 0. ]
 [0.  0.   0.   0.001 0.008 0.   0.   0.   0.   0.991]
 [0.  0.   0.   0.   0.   0.   0.   0.   1.   0. ]
 [0.  0.   0.   0.   0.   0.   0.   1.   0.   0. ]]
tensor([8, 9, 2, 2, 1, 5, 0, 9, 8, 7])

```

如果模型搭建和训练都正常，那么每一行中概率最大的取值所在的位置应该正好对应真实的标签。我们也可以让 PyTorch 自动找到最大值的位置。

```

torch.argmax(ytest, dim=1)
tensor([8, 9, 2, 2, 1, 5, 0, 9, 8, 7])

```

最后我们用模型对一些真实的手写数字图片进行预测。以下是一个例子：



接下来利用 Pillow 软件包读取图片：

```

from PIL import Image
im = Image.open("digits/sample2.png")
im

```



此时如果直接将其转为 Numpy 数组会得到三个通道：

```
im_arr = np.array(im)
print(im_arr.shape)
(100, 100, 3)
```

因此，我们先强制转换为灰度图片（单通道），再缩放至模型的图片大小 28*28：

```
im = im.convert("L")
im.thumbnail((28, 28))
im_arr = np.array(im)
print(im_arr.shape)
im
(28, 28)
```

2

为了传递给模型对象，还需要先将数值归一化到[0,1]区间，转换为 PyTorch 的 Tensor 类型，并增加一个批次和一个通道的维度：

```
test0 = torch.tensor(im_arr / 255, dtype=torch.float32).
view(1, 1, 28, 28)
print(test0.shape)
torch.Size([1, 1, 28, 28])
```

最后对图片标签进行预测：

```
pred0 = model(test0)
print(np.round(pred0.detach().cpu().numpy(), 3))
[[0. 0. 0.998 0.002 0. 0. 0. 0. 0.]]
```

DL-lec7

二、一般化建模方法

1、神经网络建模

1) 从建模的角度看，CNN 与前馈神经网络并没有本质上的不同

2) 神经元+参数=>新的神经元

2、模块化

1) 可以将各种类型的层抽象成具有统一接口的模块

a. 输入=>输出

b. 可学习的参数

c. 导数计算机制

2) 全连接层

a. 线性变换

b. 参数为权重矩阵和偏置向量

3) 激活函数层

a. 固定的非线性变换

b. 无参数（某些会带少量参数）

4) 卷积层

a. 卷积运算（**也是一种线性变换**）

b. 参数为卷积核和偏置向量

3、一般流程

1) 搭建模型

模型=结构+参数

2) 计算损失函数（标量，通常按输入取平均）

3) 对参数求导

4) 最优化、参数更新

5) 重复、迭代

4、问题

1) 模型怎么选？

2) 损失函数怎么选？

3) 导数怎么求？

4) 最优化方法怎么选？

5、模型

1) 根据数据和问题特点选择层的类型

2) 例如卷积层的引入正是为了处理图片数据

3) 神经元的数目、层的深度等超参数通常需要微调和试验，也可借鉴已有文献成果

4) 一些最新的研究在关注自动化的神经网络架构搜索（neural architecture search, NAS）

6、损失函数

1) 本质上是一个统计问题

2) 大部分情况下根据极大似然准则导出

3) 也有许多其他的损失函数，如 SVM

7、导数

1) 手动计算反向传播

2) 现代软件框架的自动微分

8、最优化

1) 基础的随机梯度下降

2) 更多改良的优化算法

二、优化方法

1、传统模型

1) 在传统统计模型中，经常使用牛顿法迭代

2) 同时利用一阶导和二阶导的信息

3) 如果参数数量是 p

4) 一阶导是梯度 g ，大小为 $p \times 1$

5) 二阶导是 Hessian 矩阵 H ，大小为 $p \times p$

6) 牛顿法需计算 $H^{-1}g$ ，复杂度为 $O(p^3)$

2、一阶方法

1) 对于深度学习模型，参数数量非常多

2) 几乎只能依赖于梯度

3) 通常称为一阶优化方法

3、GD

1) Gradient descent

2) 梯度下降法，或最速下降法

3) $\theta^{(k+1)} = \theta^{(k)} - \eta^{(k)} \cdot \nabla_{\theta} L(\theta^{(k)})$

η 称为步长或学习率

$\nabla_{\theta} L(\theta^{(k)})$ 是精确的梯度，由所有观测计算

4、SGD

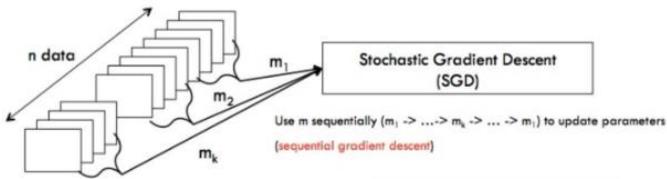
1) Stochastic gradient descent

2) 随机梯度下降

3) $\theta^{(k+1)} = \theta^{(k)} - \eta^{(k)} \cdot \nabla_{\theta} l(\theta^{(k)})$

$\nabla_{\theta} l(\theta^{(k)})$ 是 $\nabla_{\theta} L(\theta^{(k)})$ 的一个无偏估计

4) 例如通过 mini-batch 计算而来



5) 思考题

- 数据共有 n 个观测，每个 mini-batch 大小为 m ，如有剩余则单独形成一个 mini-batch
- 如何简单计算 mini-batch 的数量？
- 用 Python 实现
- SGD 的重要意义在于，即使利用随机的、不精确的梯度
- 在一定条件下也可以保证收敛到导数为 0 的点
- $\sum_{k=1}^{\infty} \eta^{(k)} = \infty$, $\sum_{k=1}^{\infty} (\eta^{(k)})^2 < \infty$
- 由统计学家 Herbert Robbins 和 Sutton Monro 在 1951 年提出
- A Stochastic Approximation Method
- $\eta^{(k)}$ 的选取

令 $S_n = \frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{n}$, $T_n = \frac{1}{1^2} + \frac{1}{2^2} + \dots + \frac{1}{n^2}$

则 $S_n - \log(n) \rightarrow 0.577216 \dots$, 故 $S_n \rightarrow \infty$

$$T_n \rightarrow \frac{\pi^2}{6} < \infty$$

12) 常用模板

```

obs_id = np.arange(n) # [0, 1, ..., n-1]
# Run the whole data set 'nepoch' times
for i in range(nepoch):
    # Shuffle observation IDs
    np.random.shuffle(obs_id)

    # Update on mini-batches
    for j in range(0, n, batch_size):
        # Create mini-batch
        x_mini_batch = x[obs_id[j:(j + batch_size)]]
        # Compute loss
        loss = model(x_mini_batch)
        # Compute gradient and update parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

DL-lec8

一、优化方法 (改进 SGD)

1. 改进 SGD

- SGD 虽然具有较好的理论性质
- 但在实际中会遇到各种挑战，如：
- 确定合适的学习率 η ，过大导致优化不收敛，过小耗费大量迭代次数

- 对每个参数使用了相同的学习率 η

2、Adagrad

- 核心思想是对每一个参数计算一个独立的 η
- 令 $g_i^{(k)}$ 为第 i 个参数在第 k 次迭代的导数

$$3) \text{ SGD 即为 } \theta_i^{(k+1)} = \theta_i^{(k)} - \eta \cdot g_i^{(k)}$$

$$4) \text{ Adagrad 为 } \theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{\sqrt{G_i^{(k)} + \epsilon}} \cdot g_i^{(k)}$$

$$\text{其中 } G_i^{(k)} = (g_i^{(1)})^2 + \dots + (g_i^{(k)})^2$$

5) 优点

- 学习率自动衰减
- 每个参数使用自适应的学习率
- 缺点

学习率衰减非常快，后期动力不足

3、RMSprop

- 对 Adagrad 加以改进
- 使用滑动平均计算 G_i

$$3) \text{ } G_i^{(k)} = \gamma G_i^{(k-1)} + (1 - \gamma) (g_i^{(k)})^2$$

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{\sqrt{G_i^{(k)} + \epsilon}} \cdot g_i^{(k)}$$

4、Adadelta

- 与 RMSprop 独立地对 Adagrad 加以改进
- 加入分子的滑动平均，保持量纲一致

$$3) \text{ } G_i^{(k)} = \gamma G_i^{(k-1)} + (1 - \gamma) (g_i^{(k)})^2$$

$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\sqrt{D_i^{(k)} + \epsilon}}{\sqrt{G_i^{(k)} + \epsilon}} \cdot g_i^{(k)}$$

$$4) \text{ 分子为 } D_i^{(k)} = \gamma D_i^{(k-1)} + (1 - \gamma) (\Delta \theta_i^{(k)})^2$$

5、Adam

- 对梯度也进行滑动平均
- $m_i^{(k)} = \beta_1 m_i^{(k-1)} + (1 - \beta_1) g_i^{(k)}$
- $v_i^{(k)} = \beta_2 v_i^{(k-1)} + (1 - \beta_2) (g_i^{(k)})^2$
- 修整偏差 $\hat{m}_i^{(k)} = m_i^{(k)} / (1 - \beta_1^k)$
- $\hat{v}_i^{(k)} = v_i^{(k)} / (1 - \beta_2^k)$

$$4) \text{ } \theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\eta}{\sqrt{\hat{v}_i^{(k)} + \epsilon}} \cdot \hat{m}_i^{(k)}$$

5、对比

- 理论性质的研究尚在进行
- 实际应用那种往往比 SGD 更快收敛
- 前期用自适应的方法，后期用 SGD

4) 见 lec8-optimizer.ipynb

5) Ali Rahimi 的演讲

<https://www.bilibili.com/video/BV1BW411Y78t>

13:27-16:00

6) 练习

第四次作业 lec8-convnet-training.ipynb

二、计算环境

1. 计算效率

1) 神经网络的计算效率由众多不同的因素决定

a. 算法收敛速度

b. 软件实现

c. 硬件设备

d. ...

2) 软件

a. 主流的深度学习框架都对核心运算，如矩阵乘法、卷积等进行了高度优化

b. Tensorflow

a) <https://www.tensorflow.org/>

b) 免费、开源

c) 主要由 Google 开发维护



TensorFlow

c. PyTorch

a) <https://pytorch.org/>

b) 免费、开源

c) 主要由 Facebook 开发维护



d. MXNet

a) <https://mxnet.apache.org/>

b) 免费、开源

c) 由陈天奇、李沐等人发起

d) 当前主要由 Apache 软件基金会和 Amazon 团队开发维护



e. MindSpore

a) <https://www.mindspore.cn/>

b) 免费、开源

c) 由华为开发维护



f. 对比

a) 不同的软件框架之间并无绝对的优劣之分

b) 语法通常非常相似

c) 个人使用可依喜好选择

d) 商业应用往往考虑兼容性和维护成本等因素

3) 硬件

a. 像绝大多数应用程序一样，神经网络模型可以运行在 CPU 上

b. 随着深度学习的流行，更多专用设备如 GPU、TPU 等被用来加速计算

c. GPU 的优势在于可以高度并行，特别适合神经网络的结构

	GeForce RTX 4090	GeForce RTX 4080	GeForce RTX 4070 Ti
GPU Engine Specs:			
NVIDIA CUDA Cores	16384	9728	7680
Boost Clock (GHz)	2.52	2.51	2.61
Base Clock (GHz)	2.23	2.21	2.31
Technology Support:			
Ray Tracing Cores	3rd Generation	3rd Generation	3rd Generation
Tensor Cores	4th Generation	4th Generation	4th Generation
NVIDIA Architecture	Ada Lovelace	Ada Lovelace	Ada Lovelace

d. 但并非 GPU 上运行的神经网络就一定比 CPU 上快

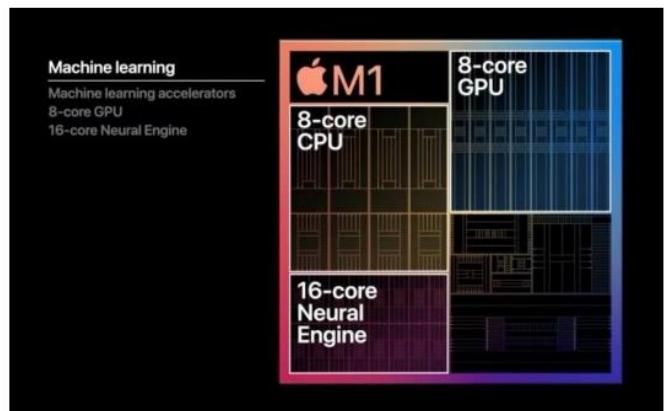
e. 数据传输到 GPU 的计算核心需要时间

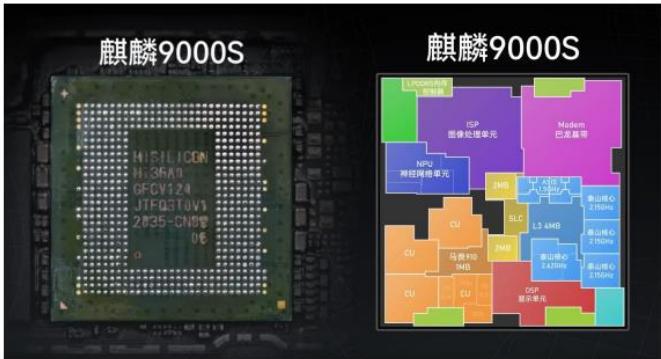
f. GPU 单核的性能一般不如 CPU

g. 数据量少、网络简单时并行效果不明显

h. 对于较复杂的网络，GPU 的运算效率往往有很大的提升

i. 一些集成芯片上还带有专门的神经网络处理器





j. Google Colab

- a) <https://colab.research.google.com/>
- b) 免费计算资源
- c) 可使用 GPU

lec8-optimizer

```
import numpy as np
import torch
from torch import nn
import torch.nn.functional as F
```

```
x = torch.tensor([[0.0, 0.0],
                  [0.0, 1.0],
                  [1.0, 0.0],
                  [1.0, 1.0]])
y = torch.tensor([[0.0],
                  [1.0],
                  [1.0],
                  [0.0]])

n = x.shape[0]
p = x.shape[1]
d = y.shape[1]
r = 5

class MyModel(nn.Module):
    def __init__(self, in_dim, out_dim, hidden_dim):
        super(MyModel, self).__init__()
        self.fc1 = nn.Linear(in_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, out_dim)

    def forward(self, x):
        z1 = self.fc1(x)
        a1 = F.softplus(z1)
        z2 = self.fc2(a1)
        phat = torch.sigmoid(z2)
        return phat

def get_losses(OptClass, nepoch=1000, lr=0.1):
    model = MyModel(in_dim=p, out_dim=d, hidden_dim=r)
    opt = OptClass(model.parameters(), lr=lr)
```

```
losses = []
```

```
for i in range(nepoch):
    phat = model(x)
    loss = torch.mean(-y * torch.log(phat) - (1.0 - y) * torch.log(1.0 - phat))

    opt.zero_grad()
    loss.backward()
    opt.step()

    losses.append(loss.item())

return losses
```

```
nepoch = 2000
```

```
lr = 0.1
```

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
losses_sgd = get_losses(torch.optim.SGD, nepoch, lr)
```

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
losses_adagrad = get_losses(torch.optim.Adagrad, nepoch, lr)
```

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
losses_adadelta = get_losses(torch.optim.Adadelta, nepoch, lr)
```

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
losses_rmsprop = get_losses(torch.optim.RMSprop, nepoch, lr)
```

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
losses_adam = get_losses(torch.optim.Adam, nepoch, lr)
```

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
```

```
import seaborn as sns
```

```
dat = pd.DataFrame({"x": range(nepoch), "sgd": losses_sgd,
                    "adagrad": losses_adagrad, "adadelta": losses_adadelta,
                    "rmsprop": losses_rmsprop, "adam": losses_adam,
                    "sgd_log": np.log(losses_sgd), "adagrad_log":
```

```

np.log(losses_adagrad),      "adadelta_      log":      |
np.log(losses_adadelta),     "rmsprop_log":      |
np.log(losses_rmsprop),      "adam_log":      |
np.log(losses_adam)})}

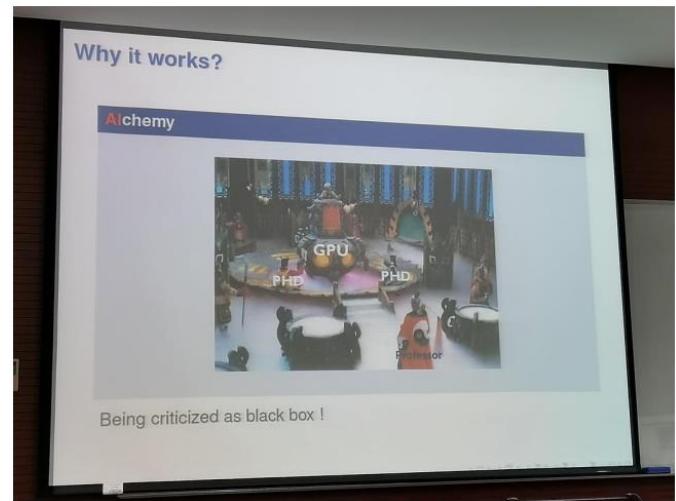
fig = plt.figure(figsize=(12, 12))
sub = fig.add_subplot(211)
sns.lineplot(data=dat, x="x", y="sgd", label="SGD")
sns.lineplot(data=dat, x="x", y="adagrad", label="Adagrad")
sns.lineplot(data=dat, x="x", y="adadelta", label="Adadelta")
sns.lineplot(data=dat, x="x", y="rmsprop", label="RMSprop")
sns.lineplot(data=dat, x="x", y="adam", label="Adam")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.ylim(0, 1)

sub = fig.add_subplot(212)
sns.lineplot(data=dat, x="x", y="sgd_log", label="SGD")
sns.lineplot(data=dat, x="x", y="adagrad_log", label="Adagrad")
sns.lineplot(data=dat, x="x", y="adadelta_log", label="Adadelta")
sns.lineplot(data=dat, x="x", y="rmsprop_log", label="RMSprop")
sns.lineplot(data=dat, x="x", y="adam_log", label="Adam")
plt.xlabel("Epoch")
plt.ylabel("Log-scale Loss")

dat

```

DL-lec9



深度学习·炼丹入门

李沐 深度学习 (Deep Learning) 话题下的优秀答主
1,525 人赞同了该文章

所谓炼丹，就是将大量灵材使用丹炉将其凝炼成丹。炼成的灵丹蕴含灵材的大部分特性，方便携带，容易吸收。高级仙丹在炼制中更是能吸收天地灵气从而引发天地异象。

深度学习的模型训练就是炼丹。把精选原始数据，按照神经网络的规定法则通过计算框架提炼，从而得到一个远小于数据数倍的模型。一个好的模型不仅能抓取数据中的模式，更是能提取更加一般化规则从而可以用来预测新的数据。

精选评论 (1)

知乎用户 2016-11-19
卷积三钱、全联二钱，盗梦半钱，极池一分。四味和匀，先大火煮沸，再以文火慢炖七日七夜。择良辰停火取药，滤去西梯西、弃参等杂质，以叉八六泰格拉送服。可以开天窍、晓阴阳、发混蒙、妙不可言。
208 查看回复

Ali Rahimi 的演讲

<https://www.bilibili.com/video/BV1BW411Y78t>

11:00-13:18

一、“炼金术”

1、为什么深度学习被戏称为炼金术？

1) 正面含义：

- 有用
 - 催生新的技术和问题
- 2) 负面含义：
- 系统性的科学认识还不足
 - 过度包装、泛化

2、当前目标

1) 吸取实践中积累的经验

2) 尽可能去解释背后的原因和动机

3、实践技巧

- 数据预处理
- 参数初始化
- 激活函数
- 特殊隐藏层
- 正则化方法
- 计算环境

二、数据预处理

实践

1、对数据做中心化

- 按观测进行平均 (每个观测减去观测平均值)
- 按图形通道进行平均 (每个通道减去所有通道的平均值)

原因

1、一些权重和激活函数对输入数据的数值范围比较敏感

三、参数初始化

实践

- 对于以零为中心的激活函数使用 Xavier 初始化方法
- 对于 ReLU 使用 Kaiming 初始化方法

原因

1、对输出的方法约等于输入的方差

2、针对 ReLU 进行系数修正

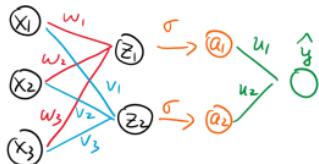
1、简单初始化

1) 对于简单的网络，可以用固定的分布随机初始化

2) 例如 $Unif(-0.01, 0.01)$ 和 $N(0, 0.01^2)$

3) **不能全部初始化为 0!**

2、对称初始化



If $w_1 = v_1, w_2 = v_2, w_3 = v_3, u_1 = u_2$, then $z_1 = z_2, a_1 = a_2$

$$u_1 = u_2 \Rightarrow \frac{dl}{da_1} = \frac{dl}{da_2}, \frac{dl}{du_1} = \frac{dl}{du_2}$$

$$\frac{dl}{dz_1} = \frac{dl}{da_1} \cdot \sigma'(z_1) = \frac{dl}{da_2} \cdot \sigma'(z_2) = \frac{dl}{dz_2}$$

$$\frac{dl}{dw_1} = \frac{dl}{dz_1} \cdot \frac{dz_1}{dw_1} = \frac{dl}{dz_1} \cdot x_1 = \frac{dl}{dz_2} \cdot x_1 = \frac{dl}{dw_1}$$

$$\dots$$

$$\frac{dl}{dw_i} = \frac{dl}{dz_1}$$

1) 结论

a. 不光是不能将参数初始化为 0

b. 即使是对称也不行

c. 需要使用随机初始化

2) 初始化影响

a. 对于高维、深层的网络，考虑 Xavier 和 Kaiming 初始化

b. 见 weight_INITIALIZATION.ipynb

四、激活函数

实践

1、默认使用 ReLU

2、尝试使用 Leaky ReLU、ELU 等 ReLU 的变种

3、尽量避免 Sigmoid 和 Tanh (输出层除外)

原因

1、梯度饱和

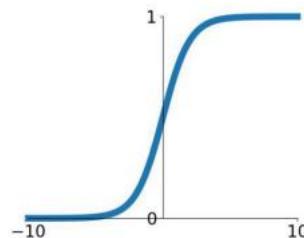
2、以零为中心

3、计算复杂度

1、Sigmoid

1) x 在 $[-5, 5]$ 之外几乎是平坦取值

2) 导数“饱和”了



Sigmoid

3) 回顾 BP

$$a = \sigma(z)$$

$$b. \frac{dl}{dz} = \frac{dl}{da} \circ \sigma'(z)$$

上游导数

$$\boxed{\frac{dl}{dz}} = \boxed{\frac{dl}{da}} \circ \boxed{\sigma'(z)}$$

下游导数 **局部导数**

c. 当局部导数接近 0 时，下游导数也几乎为 0

4) Sigmoid 的输出范围是 $[0, 1]$

5) 不是关于 0 对称的

6) 如果输入永远是正的会发生什么？

7) $z = Wx + b, x \geq 0$

$$8) \frac{dl}{dw} = \frac{dl}{dz} x^T = \left[\frac{dl}{da} \circ \sigma'(z) \right] x^T$$

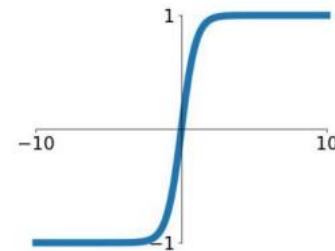
$$\left[\frac{dl}{da} \circ \boxed{\sigma'(z)} \right] \boxed{x^T} \geq 0 \geq 0$$

9) W 导数的符号几乎被上游导数的符号制约

2、Tanh

1) Tanh 解决了输出范围中心化的问题

2) 但导数饱和的问题依然存在



tanh(x)

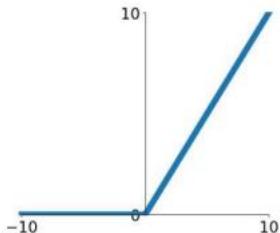
3、ReLU

1) 好处

a. 在 > 0 的区域永远有梯度

b. 计算简单

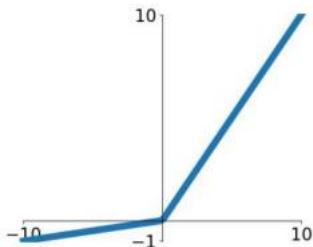
- 2) 不足
 a. 值域非以 0 为中心
 b. <0 的导数为 0



ReLU (Rectified Linear Unit)

4、Leaky ReLU

- 1) 在所有区域都有梯度
- 2) 计算简单
- 3) 输出有正有负

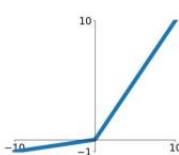


Leaky ReLU

$$f(x) = \max(0.01x, x)$$

5、PReLU

- 1) 让 Leaky ReLU 中的系数变成可学习参数
- 2) 反向传播时需计算 α 的导数并更新取值



Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

五、特殊隐藏层

实践

- 1、使用归一化层
- 2、使用残差网络

原因

- 1、保持一定的尺度不变性，有利于优化
- 2、以线性函数为基础，拟合残差

1、归一化层

- 1) 批量归一化 (Batch normalization)
- 2) 层归一化 (Layer normalization)

2、批量归一化

- 1) 类似于数据预处理中的中心化和标准化



计算每一列的均值和方差

- 2) x_{ij} 表示第 j 个神经元的第 i 个观测
- 3) $\mu_j = \frac{1}{N} \sum_{i=1}^N x_{ij}, \sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{ij} - \mu_j)^2$
- 4) $\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$
- 5) 加入可学习参数向量 β, γ
- 6) $z_{ij} = \gamma_j \hat{x}_{ij} + \beta_j$
- 7) 思考题
 - a. BN 归一化的参数依赖于 mini-batch 的选择
 - b. 对单个观测作预测时怎么计算？
 - a) 计算所有观测的均值和方差
 - b) 或在训练时计算均值和方差的滑动平均
 - c) 这里可能出现不一致性！
 - 8) PyTorch
 - a. <https://pytorch.org/docs/stable/nn.html#normalization-layers>
 - b. PyTorch 中模型可以设置状态
model.train() 用于训练
model.eval() 用于预测

Batch Normalization for
fully-connected networks

$\mathbf{x}: N \times D$

Normalize ↓

$\mu, \sigma: 1 \times D$

$\gamma, \beta: 1 \times D$

$y = \gamma(x - \mu) / \sigma + \beta$

Batch Normalization for
convolutional networks
(Spatial Batchnorm, BatchNorm2D)

$\mathbf{x}: N \times C \times H \times W$

Normalize ↓

$\mu, \sigma: 1 \times C \times 1 \times 1$

$\gamma, \beta: 1 \times C \times 1 \times 1$

$y = \gamma(x - \mu) / \sigma + \beta$

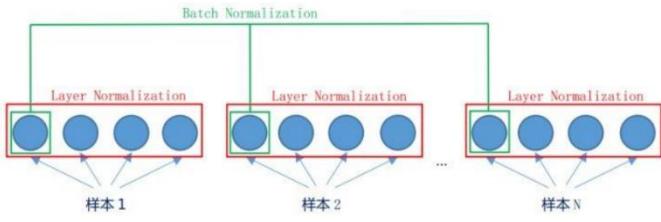
3、层归一化

- 1) 对每个观测的所有变量做归一化



批量归一化：
计算每一列的均值和方差

层归一化：
计算每一行的均值和方差



4、残差网络

- 1) 在早期的深度学习模型中出现反常的现象
- 2) 深度网络比浅层网络的训练误差更大

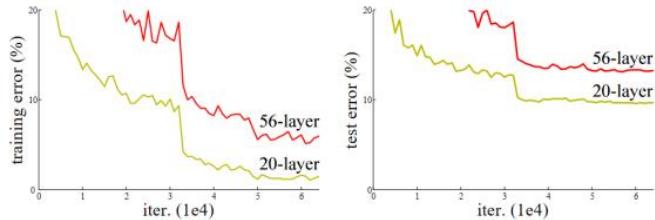


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

- 3) 可能的原因是深层网络的目标函数更复杂、更难以优化
- 4) 而线性函数是非常稳定的结构
- 5) 残差网络的思想就是先用线性函数去拟合目标，再用非线性神经网络去拟合残差

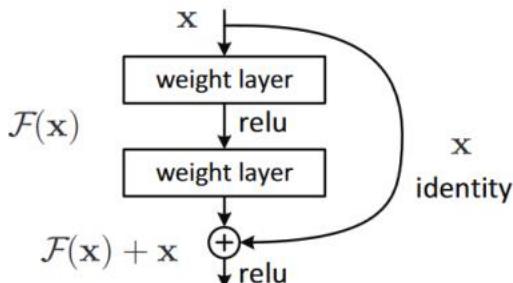


Figure 2. Residual learning: a building block.

- 6) 实现非常简单
- 7) 但要注意维度的匹配

```
p = 3
d = 10
fc1 = nn.Linear(in_features=p, out_features=d)
fc2 = nn.Linear(in_features=d, out_features=p)
```

```
n = 5
x = torch.randn(n, p)

z1 = fc1(x)
a1 = torch.relu(z1)
z2 = fc2(a1)
z2_2 = x + z2
a2 = torch.relu(z2_2)
```

- 8) 延伸阅读
 - a. 李沐《精读论文》系列

b. <https://www.bilibili.com/video/BV1Fb4y1h73E>

c. <https://www.bilibili.com/video/BV1P3411y7nn>

六、正则化方法

实践

- 1、目标函数中加入正则项
- 2、使用提前停止、丢弃法等机制

原因

- 1、减小过拟合风险
- 2、增强模型泛化能力

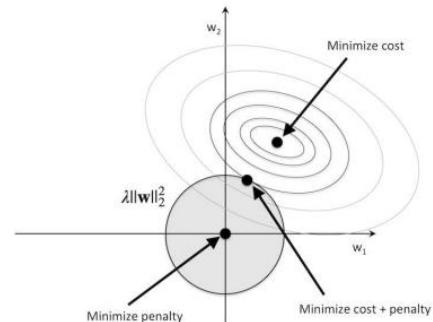
1、模型泛化

- 1) 神经网络往往具有非常多的参数
- 2) 根据通用近似定理，按照一定的规则增加参数可以获得更好的拟合效果
- 3) 但同时增加了过拟合的风险
- 4) 我们不希望网络只是记忆已知的数据
- 5) 而是有一定的泛化能力

2、正则化

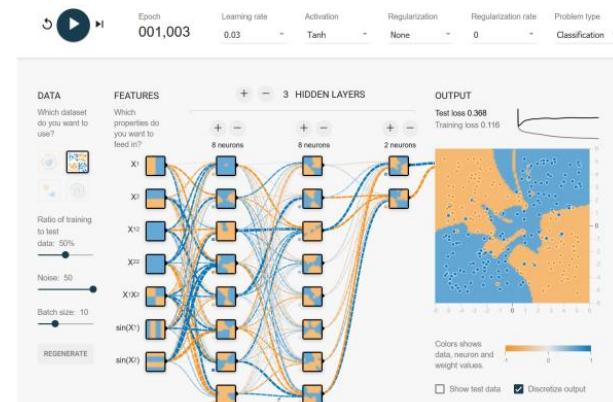
- 1) 正则化可以认为是对模型的参数加入一些约束条件，使模型的复杂度降低
- 2) 显示正则化
损失函数加入正则项
- 3) 隐式正则项
 - a. 提前停止 (Early stopping)
 - b. 丢弃法 (Dropout)
- 4) 正则项

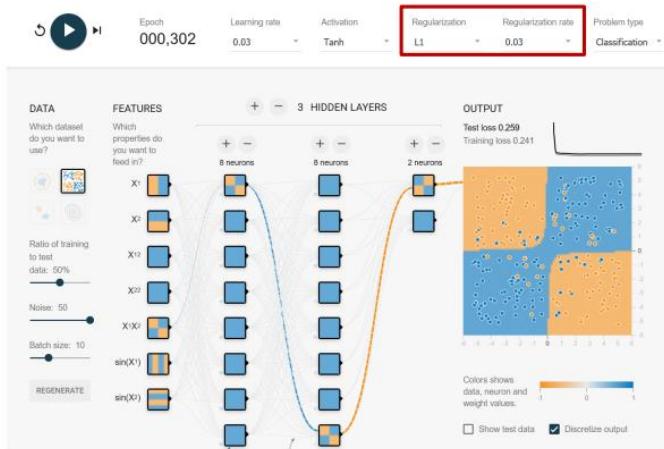
$$\theta^* = \arg \min_{\theta} \frac{1}{N} \sum_{n=1}^N \mathcal{L}(y^{(n)}, f(\mathbf{x}^{(n)}, \theta)) + \lambda \ell_p(\theta)$$



- 5) 演示

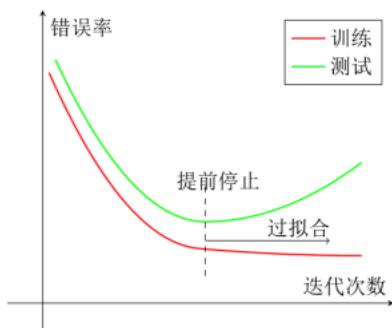
<https://playground.tensorflow.org>





6) 提前停止

训练时划分出一个验证集，当验证集误差率不再下降时就停止迭代



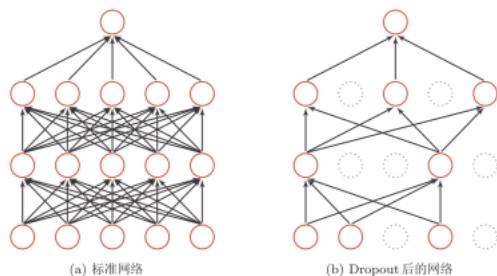
7) 丢弃法

a. 当数据通过隐藏层时，随机丢弃一些神经元

$$b. y = \sigma(Wd(x) + b)$$

$$d(x) = \begin{cases} m \odot x & \text{当训练阶段时} \\ px & \text{当测试阶段时} \end{cases}$$

c. $m \in \{0,1\}^d$ 利用 Bernoulli 分布生成



8) PyTorch

a. <https://pytorch.org/docs/stable/nn.html#dropout-layers>

b. PyTorch 中对于 Dropout 同样要注意设置模型状态

- a) model.train() 用于训练
- b) model.eval() 用于预测

lec9-weight initialization

```
import math
import numpy as np
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
```

```
np.random.seed(123)
torch.random.manual_seed(123)
<torch._C.Generator at 0x1b38c6edb0>
```

n = 10

p = 1000

x = torch.randn(n, p)

fcs = [nn.Linear(p, p, bias=False) for i in range(5)]

print(fcs)

```
[Linear(in_features=1000, out_features=1000, bias=False),
 Linear(in_features=1000, out_features=1000, bias=False),
 Linear(in_features=1000, out_features=1000, bias=False),
 Linear(in_features=1000, out_features=1000, bias=False),
 Linear(in_features=1000, out_features=1000, bias=False)]
```

一、固定方差初始化

sigma = 0.02

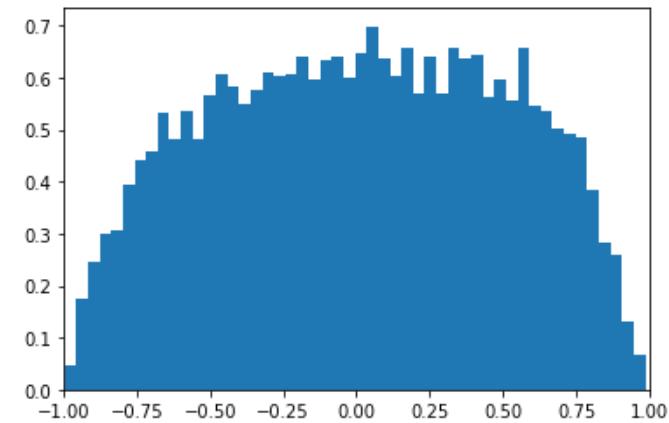
sigma = 0.1

nn.init.normal_(fcs[0].weight, mean=0.0, std=sigma)

a1 = torch.tanh(fcs[0](x))

```
plt.hist(a1.detach().numpy().reshape(-1), bins=50,
 density=True)
plt.xlim(-1, 1)
```

(-1.0, 1.0)

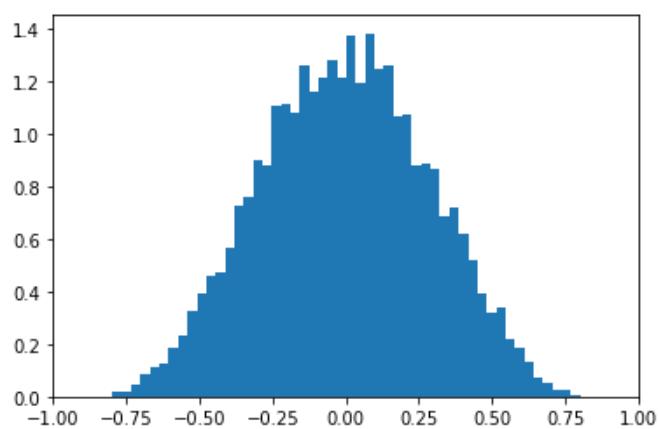


nn.init.normal_(fcs[1].weight, mean=0.0, std=sigma)

a2 = torch.tanh(fcs[1](a1))

```
plt.hist(a2.detach().numpy().reshape(-1), bins=50,
 density=True)
plt.xlim(-1, 1)
```

(-1.0, 1.0)



```

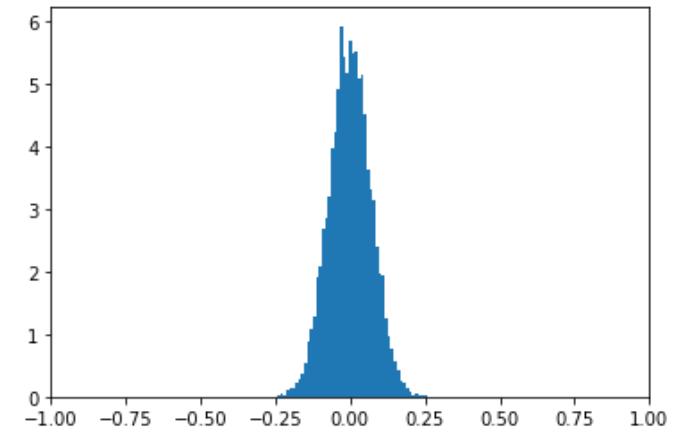
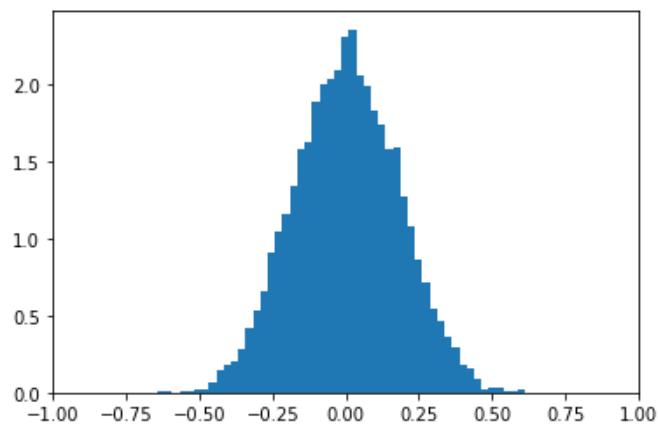
nn.init.normal_(fcs[4].weight, mean=0.0, std=sigma)
a5 = torch.tanh(fcs[4](a4))
plt.hist(a5.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)

```

```

nn.init.normal_(fcs[2].weight, mean=0.0, std=sigma)
a3 = torch.tanh(fcs[2](a2))
plt.hist(a3.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)

```



二、Xavier 初始化

```

np.random.seed(123)
torch.random.manual_seed(123)

```

```

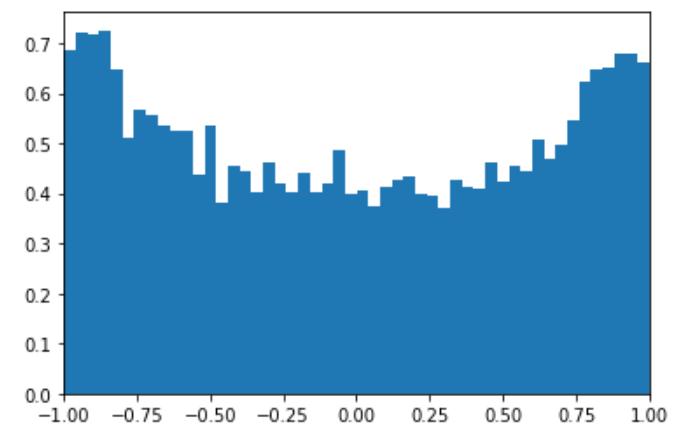
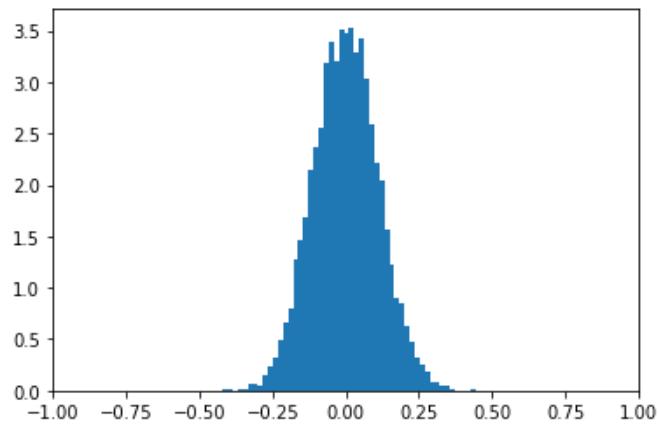
in_dim = fcs[0].weight.shape[1]
nn.init.normal_(fcs[0].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a1 = torch.tanh(fcs[0](x))
plt.hist(a1.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)

```

```

nn.init.normal_(fcs[3].weight, mean=0.0, std=sigma)
a4 = torch.tanh(fcs[3](a3))
plt.hist(a4.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)

```

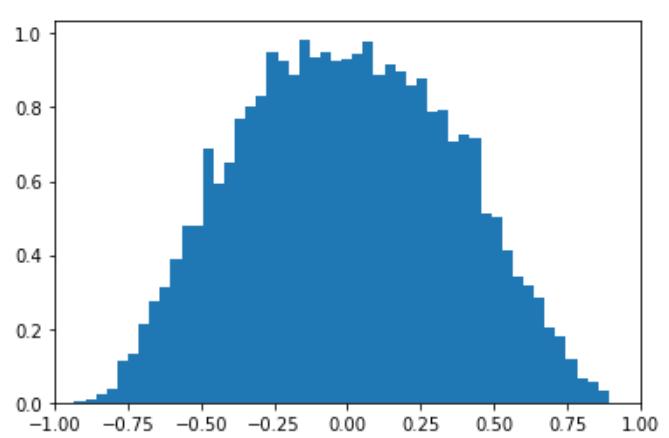
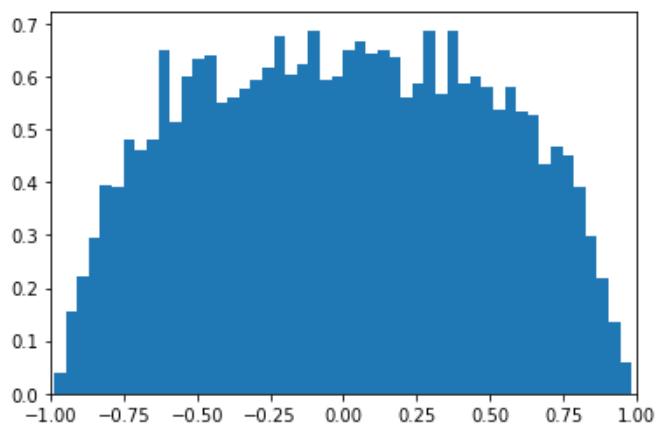


```

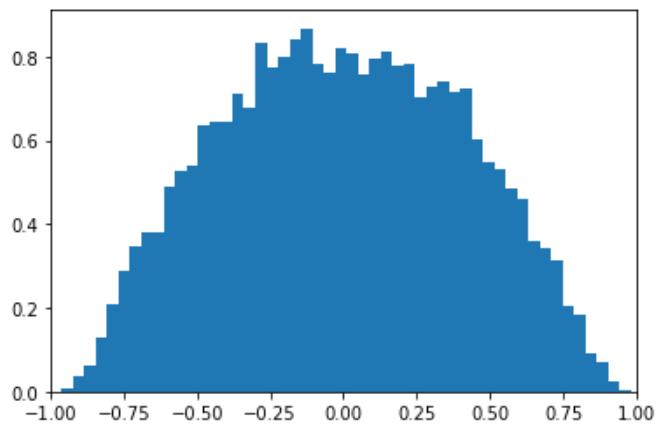
in_dim = fcs[1].weight.shape[1]
nn.init.normal_(fcs[1].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a2 = torch.tanh(fcs[1](a1))
plt.hist(a2.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)

```

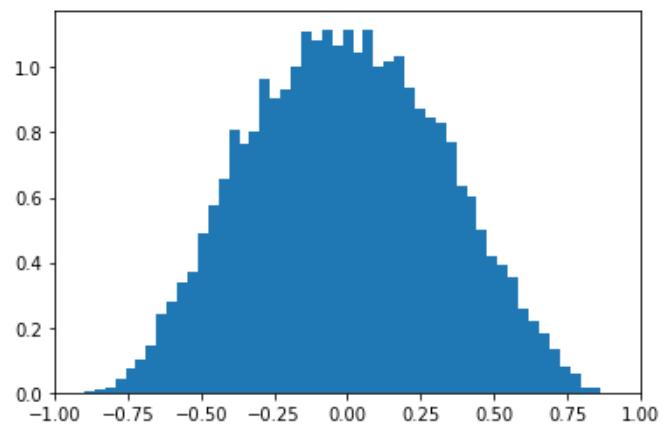
(-1.0, 1.0)



```
in_dim = fcs[2].weight.shape[1]
nn.init.normal_(fcs[2].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a3 = torch.tanh(fcs[2](a2))
plt.hist(a3.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)
```



```
in_dim = fcs[4].weight.shape[1]
nn.init.normal_(fcs[4].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a5 = torch.tanh(fcs[4](a4))
plt.hist(a5.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)
```

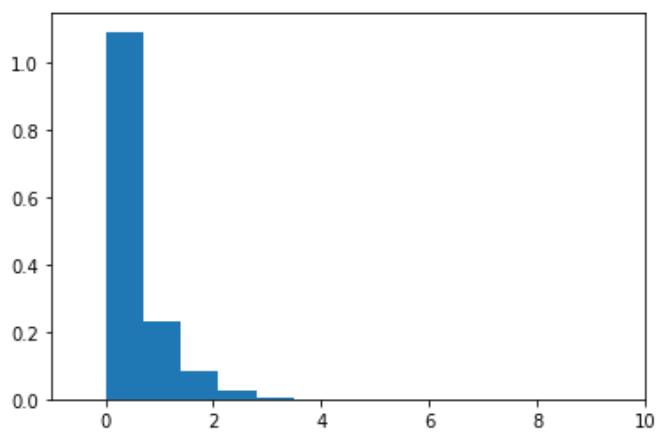


```
in_dim = fcs[3].weight.shape[1]
nn.init.normal_(fcs[3].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a4 = torch.tanh(fcs[3](a3))
plt.hist(a4.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 1)
(-1.0, 1.0)
```

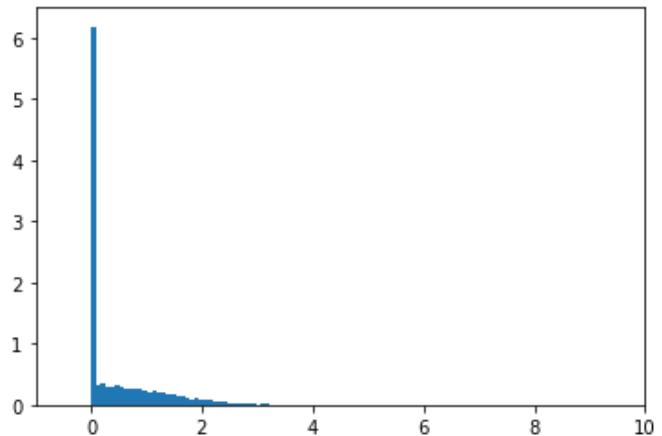
三、以 ReLU 为激活函数的 Xavier 初始化

```
np.random.seed(123)
torch.random.manual_seed(123)
```

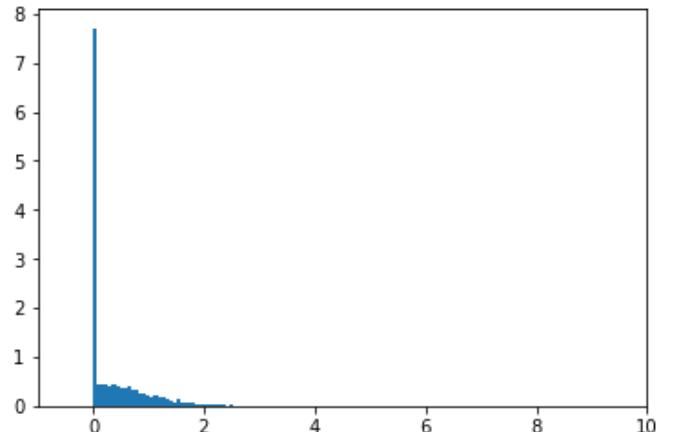
```
in_dim = fcs[0].weight.shape[1]
nn.init.normal_(fcs[0].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a1 = torch.relu(fcs[0](x))
plt.hist(a1.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)
```



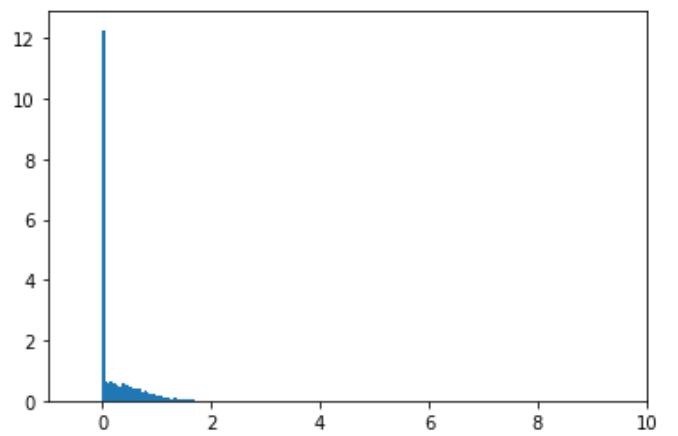
```
in_dim = fcs[1].weight.shape[1]
nn.init.normal_(fcs[1].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a2 = torch.relu(fcs[1](a1))
plt.hist(a2.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)
```



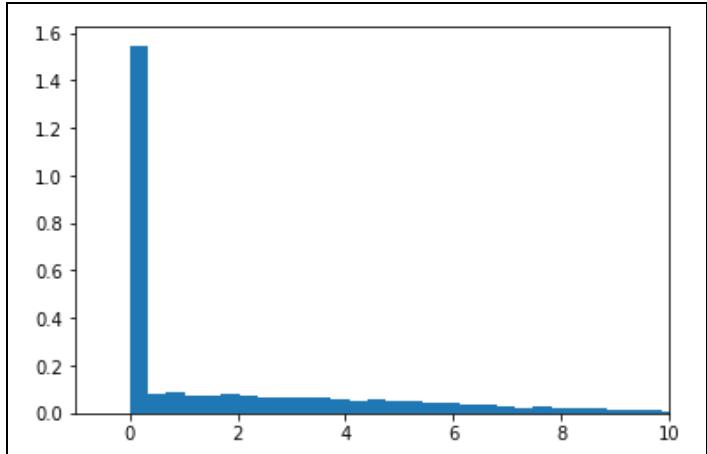
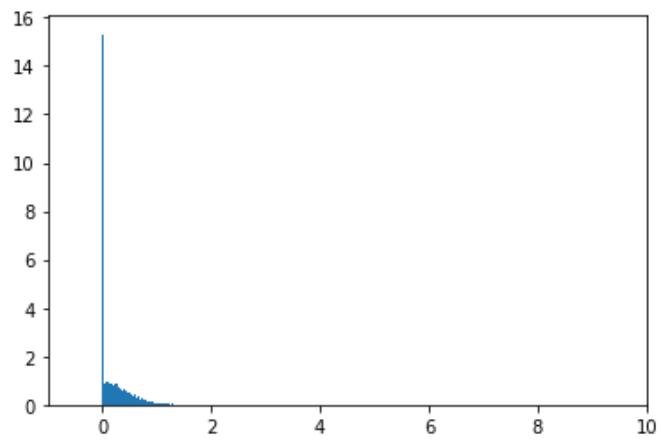
```
in_dim = fcs[2].weight.shape[1]
nn.init.normal_(fcs[2].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a3 = torch.relu(fcs[2](a2))
plt.hist(a3.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)
```



```
in_dim = fcs[3].weight.shape[1]
nn.init.normal_(fcs[3].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a4 = torch.relu(fcs[3](a3))
plt.hist(a4.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)
```



```
in_dim = fcs[4].weight.shape[1]
nn.init.normal_(fcs[4].weight, mean=0.0, std=1.0 / 
math.sqrt(in_dim))
a5 = torch.relu(fcs[4](a4))
plt.hist(a5.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)
```



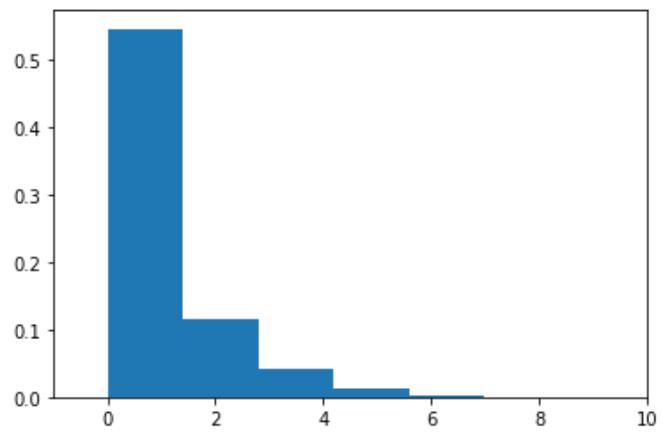
四、Kaiming 初始化

```

np.random.seed(123)
torch.random.manual_seed(123)

in_dim = fcs[0].weight.shape[1]
nn.init.normal_(fcs[0].weight, mean=0.0, std=2.0 / 
math.sqrt(in_dim))
a1 = torch.relu(fcs[0](x))
plt.hist(a1.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)

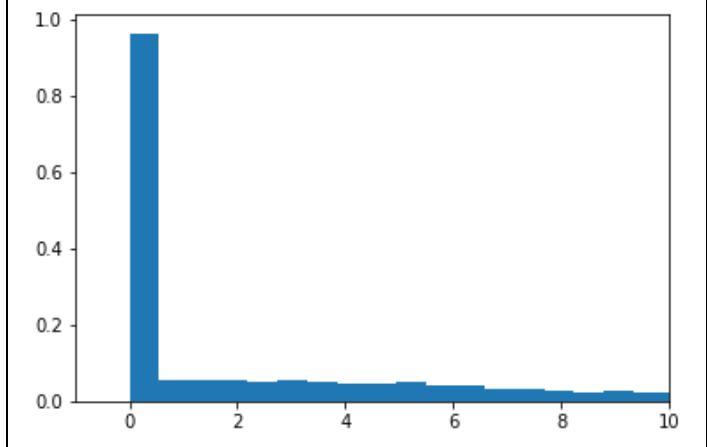
```



```

in_dim = fcs[2].weight.shape[1]
nn.init.normal_(fcs[2].weight, mean=0.0, std=2.0 / 
math.sqrt(in_dim))
a3 = torch.relu(fcs[2](a2))
plt.hist(a3.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)

```



```

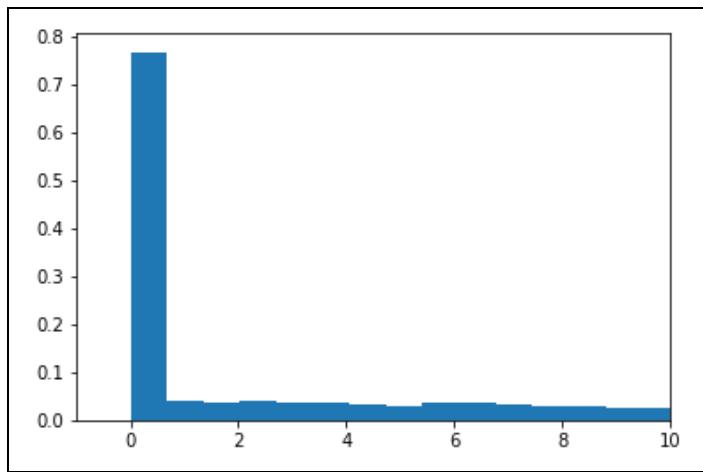
in_dim = fcs[1].weight.shape[1]
nn.init.normal_(fcs[1].weight, mean=0.0, std=2.0 / 
math.sqrt(in_dim))
a2 = torch.relu(fcs[1](a1))
plt.hist(a2.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)

```

```

in_dim = fcs[3].weight.shape[1]
nn.init.normal_(fcs[3].weight, mean=0.0, std=2.0 / 
math.sqrt(in_dim))
a4 = torch.relu(fcs[3](a3))
plt.hist(a4.detach().numpy().reshape(-1), bins=50,
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)

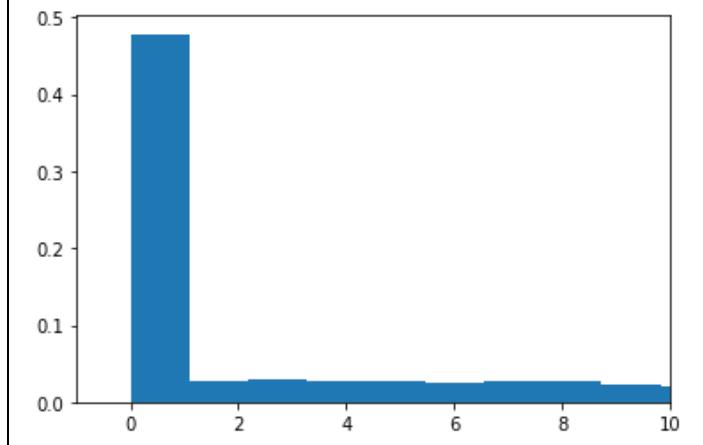
```



```

in_dim = fcs[4].weight.shape[1]
nn.init.normal_(fcs[4].weight, mean=0.0, std=2.0 / 
math.sqrt(in_dim))
a5 = torch.relu(fcs[4](a4))
plt.hist(a5.detach().numpy().reshape(-1), bins=50, 
density=True)
plt.xlim(-1, 10)
(-1.0, 10.0)

```

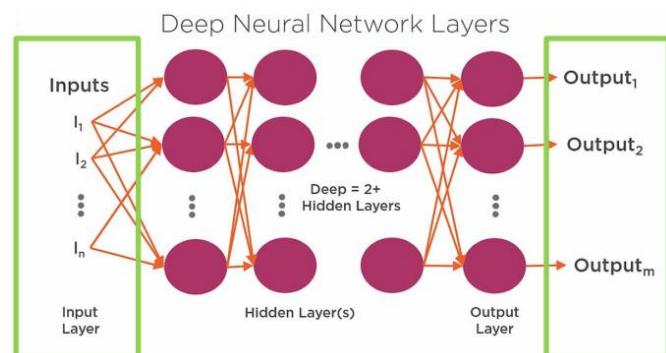


DL-lec10

一、循环神经网络（应用）

1、FNN/CNN

1) 定长输入=>定长输出



n 维输入

m 维输出

2) 28×28 个像素=>10 个类别

3) 如果输入长度不确定怎么办？

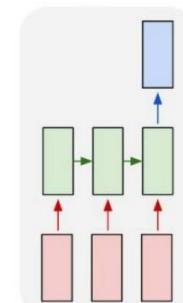
4) 例：用名字预测性别

- (张三, 男)
- (李清照, 女)
- (工藤新一, 男)
- (克里斯蒂娜, 女)
- (柯尔莫哥洛夫, 男)

2、序列数据

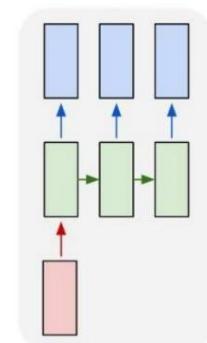
1) 可以把名字看成是由单个字符组成的序列

- a. 名字=>性别
- b. 不定长序列=>定长类别



2) 序列数据可以衍生出许多类型的问题

- a. 定长数据=>不定长序列
- b. 例：图片描述



A cat sitting on a suitcase on the floor



A cat is sitting on a tree branch



A dog is running in the grass with a frisbee



Two people walking on the beach with surfboards



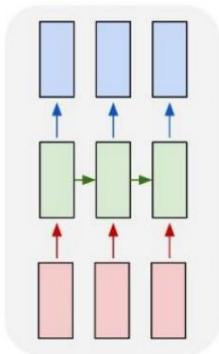
A tennis player in action on the court



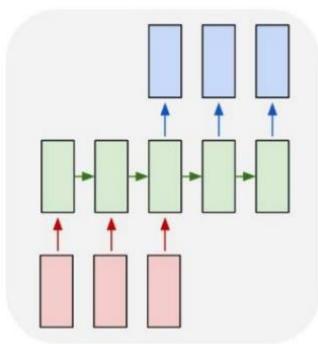
Two giraffes standing in a grassy field

3) 序列数据可以衍生出许多类型的问题

- a. 不定长序列=>不定长序列
- b. 输入输出一一对应
- c. 例：影片逐帧分类



- 4) 序列数据可以衍生出许多类型的问题
- 不定长序列=>不定长序列
 - 输入输出无对应关系
 - 例：机器翻译



DeepL Translator DeepL Pro Download for Windows Login

Translate text Translate .docx & .pptx files

Translate from English (detected) Into Chinese Glossary

Across the Great Wall we can reach every corner in the world. 穿过长城，我们可以到达世界上每个角落。

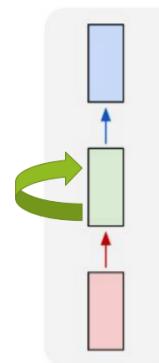
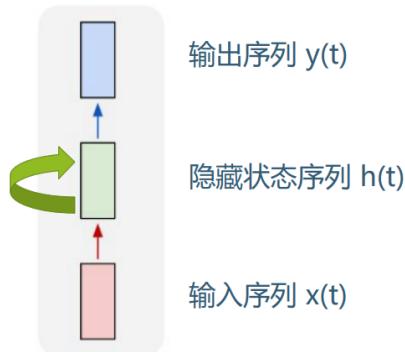
二、循环神经网络

1、循环神经网络

- Recurrent Neural Networks (RNN)
- 核心是对序列数据进行建模

2、RNN 结构

1)



$y(t)$ 是 $h(t)$ 的函数

$h(t)$ 是 $x(t)$ 和 $h(t-1)$ 的函数

输入序列 $x(t)$

2)

新状态

旧状态

$$h_t = f_W(h_{t-1}, x_t)$$

由参数 W
决定的函数

当前输入变量

参数 W 不随时间变化!

3) 回顾

- “序列数据”好像听着有些耳熟?
- 经典统计模型中有哪些方法刻画序列数据?
- 时间序列分析中的向量自回归模型 (VAR)

$$y_t = c + Ay_{t-1} + \varepsilon_t$$

误差项
线性变换

- 如果令 $h_t = E[Y_t]$, 则 $h_t = c + Ah_{t-1}$
- 假设 c 还和其他的序列 x_t 相关, 如 $c = Bx_t$
- 那么可以写成 $h_t = Bx_t + Ah_{t-1}$

是 $h_t = f_W(h_{t-1}, x_t)$ 的特例!

- 如果我们使用非线性变换呢?
- 线性回归模型=>前馈神经网络
- 线性自回归模型=>RNN

4) 常见 RNN

新状态

旧状态

$$h_t = f_W(h_{t-1}, x_t)$$

由参数 W
决定的函数

当前输入变量

$$a. h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$b. y_t = W_{hy}h_t$$

所有的 W 参数都不随时间变化!

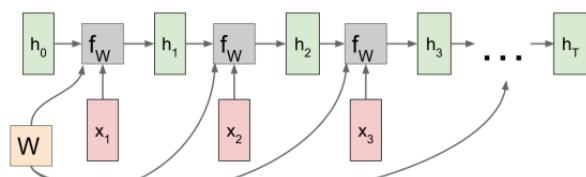
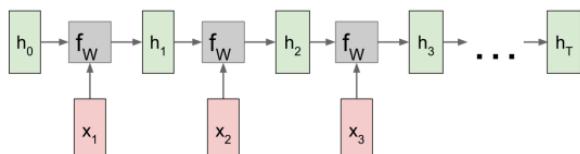
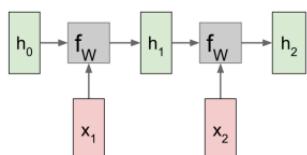
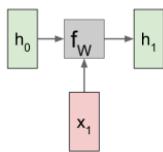
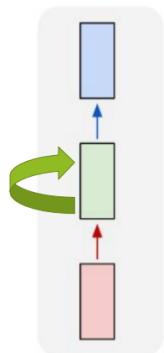
3、基本原理

- RNN 的基本假定是序列中前后的数据是“相关”的
 - 上海财_?
 - 上海市财_?
- 隐藏状态 $h(t)$ 起到了“记忆”的作用
汇总了当前以及过去的信息
- $h(t)$ 的变化模式是稳定的
参数 W 不随时间变化

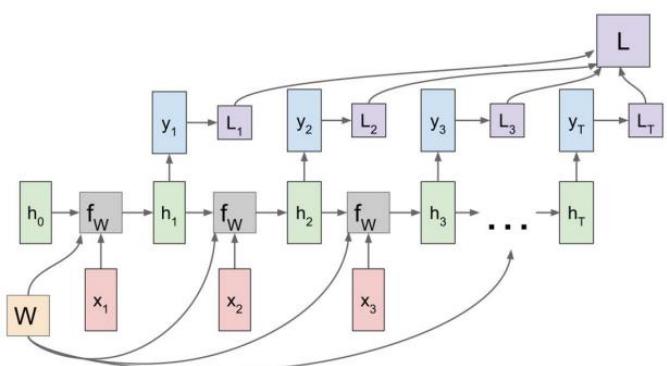
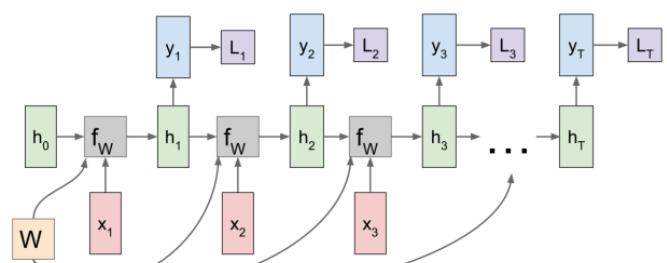
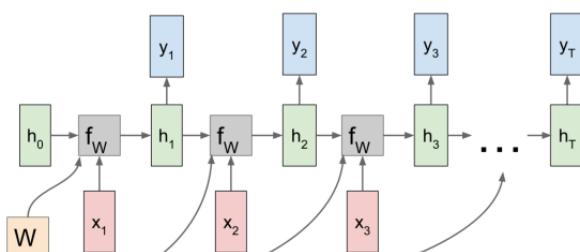
三、直观理解

1、理解 RNN

- 1) 这张图其实隐藏了很多细节
- 2) 我们可以把循环展开
- 3) 从而更好地理解 RNN 的建模过程

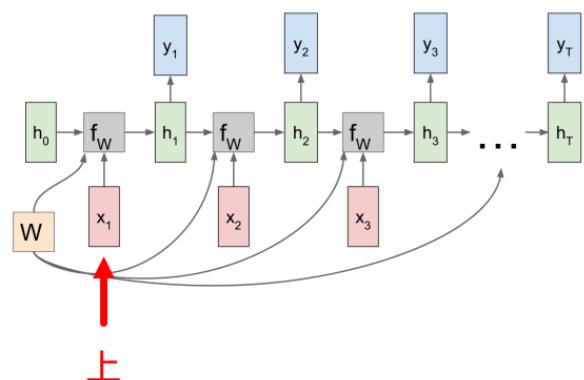


权重共用



损失 L_1

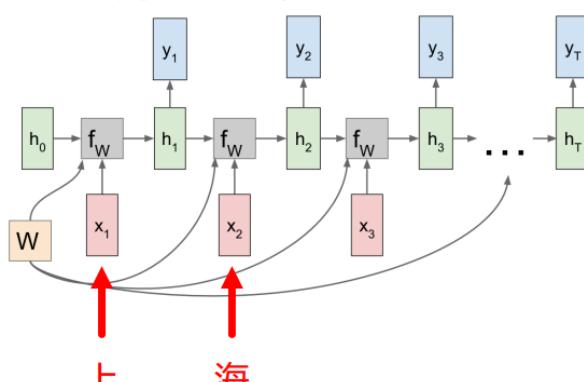
**预测：财
实际：海**



上

损失 L_1

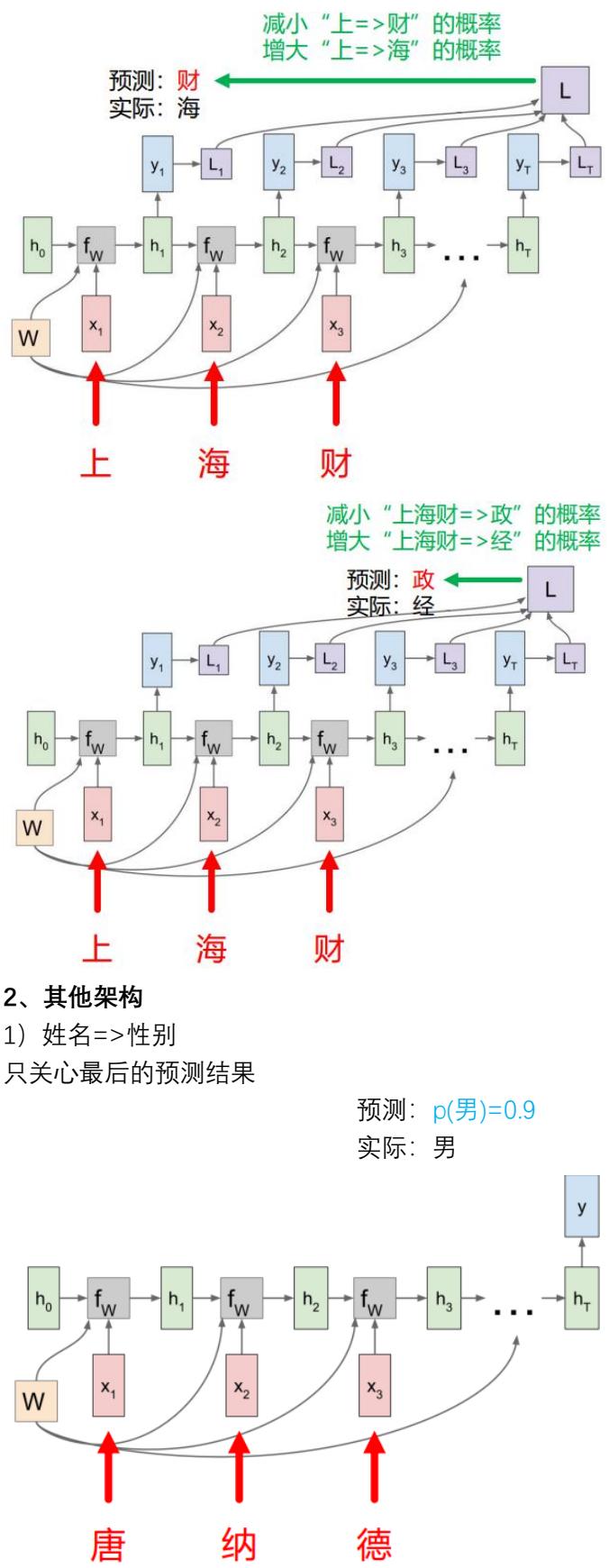
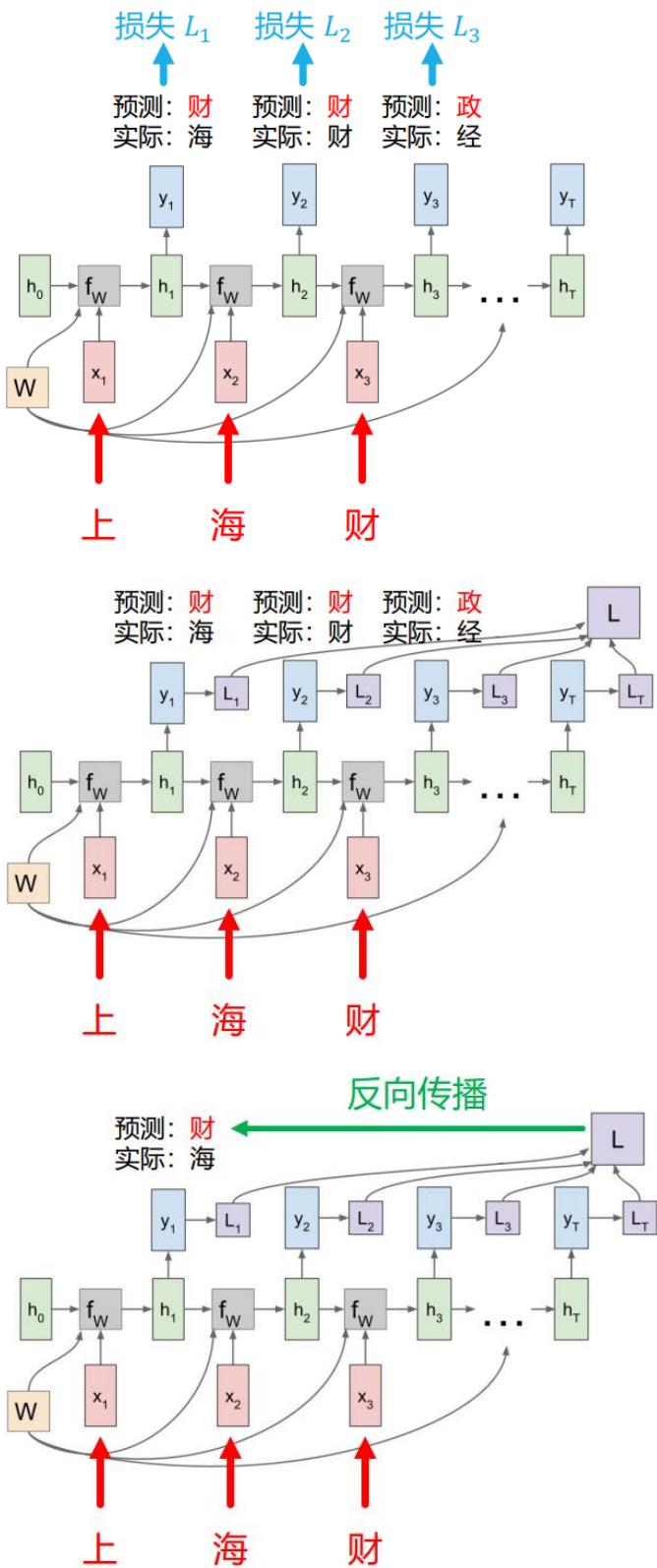
**预测：财
实际：海**



上

损失 L_2

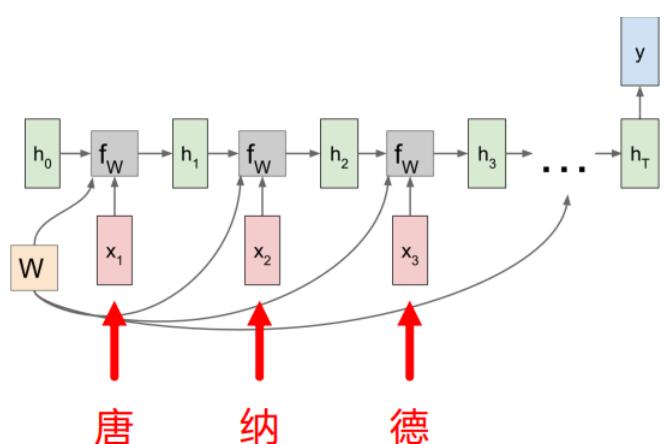
**预测：财
实际：财**



2、其他架构

- 1) 姓名=>性别
只关心最后的预测结果

预测: $p(\text{男})=0.9$
实际: 男



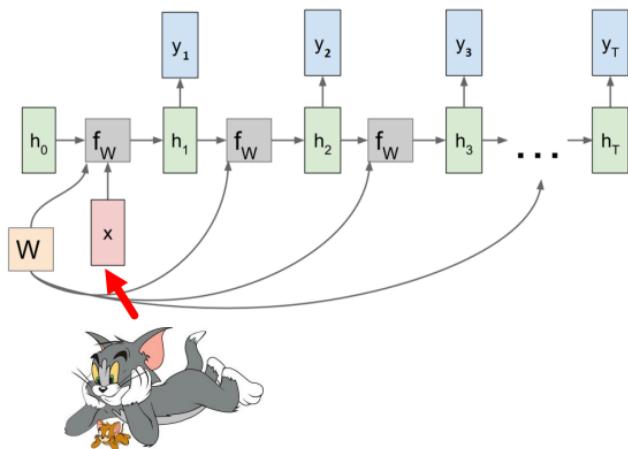
- 2) 图片=>文字描述

预测: 猫
实际: 猫

预测: 与
实际: 和

预测: 老
实际: 老

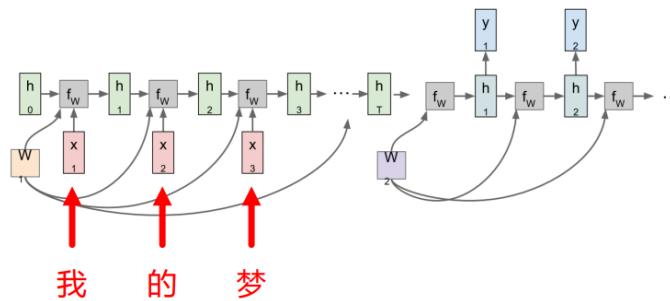
预测: 鼠
实际: 鼠



3) 中文=>英文

预测: My
实际: My

预测: Dream
实际: Dream



DL-lec11

循环神经网络 (续)

一、实例演示

1. 性别预测

1) 数据来源

<https://github.com/wainshine/Chinese-Names-Corpus>

2) >114 万条已标注性别的人名

dict,sex
阿安,男
阿彬,未知
阿斌,男
阿滨,男
阿冰,女
阿冰冰,女
阿兵,男
阿婵,女
阿超,男
阿朝,男
阿琛,女
阿臣,男
阿辰,未知
阿晨,未知

石晓彦,女
石晓艳,女
石晓燕,女
石晓艺,女
石晓英,女
石晓莹,女
石晓颖,女
石晓影,女
石晓勇,男
石晓宇,男
石晓玉,女
石晓云,女
石晓泽,男
石晓珍,女
石筱,女

闫志慧,女
闫志坚,男
闫志江,男
闫志杰,男
闫志娟,女
闫志军,男
闫志君,未知
闫志丽,女
闫志利,男
闫志亮,男
闫志林,男
闫志玲,女
闫志龙,男
闫志梅,女
闫志民,男

2. 预处理

- 1) 删除未知类别
- 2) 计算每个字出现的频率
- 3) 选取前 500 个高频字
- 4) 将范围限定在 500 个常用字中

char	freq
636 王	50390
926 李	49078
1086 张	47089
1203 陈	41512
1394 刘	39986
...	...
2170 墙	1
2171 楼	1
2172 周	1
2173 据	1
1962 莽	1

3、建立字典

1) 将 500 个常用字作为字典

```
array(['王', '李', '张', '刘', '文', '林', '明', '杨', '华', '黄', '吴', '金',
       '周', '晓', '国', '赵', '玉', '伟', '海', '志', '徐', '丽', '红', '建', '朱',
       '孙', '平', '军', '英', '春', '龙', '胡', '水', '荣', '德', '云', '成', '郭',
       '东', '郑', '高', '芳', '马', '何', '梅', '新', '杰', '辉', '生', '秀', '玲',
       '江', '俊', '洪', '强', '世', '光', '罗', '艳', '燕', '兰', '子', '庆', '峰',
       '忠', '染', '宇', '凤', '谢', '霞', '美', '宋', '祥', '清', '立', '兴', '萍',
       '许', '叶', '雪', '良', '安', '慧', '娟', '福', '宝', '佳', '方', '家', '唐']
```

2) 将汉字用 one-hot 向量编码

a. 王=[1,0,0,...]

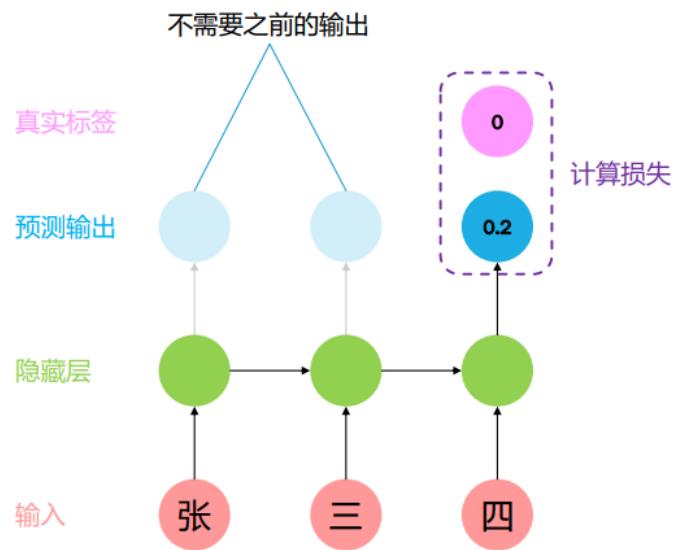
b. 李=[0,1,0,...]

c. 张=[0,0,1,...]

3) 每个名字看作是一个序列 $x = (x_1, x_2, \dots)$

a. 每个 x_t 是一个 one-hot 向量

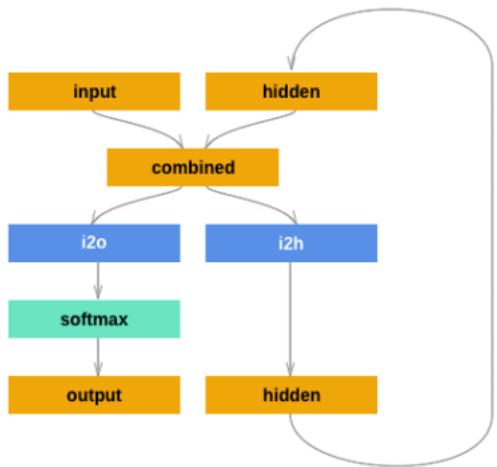
4、建模原理



5、建立模型

1) 参考:

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html



6、测试结果

- 1) 简单构建训练集（1万）和测试集（1万）
- 2) CPU 上训练 10 秒钟
- 3) 测试集准确率可达 97.9%

7、代码实现

参见 name_classify.ipynb

lec11-name classify

```
import time
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import itertools
import collections
import matplotlib.pyplot as plt

# 读取数据
df = pd.read_csv("Chinese_Names_Corpus_Gender(120W).txt", header=2)
df
```

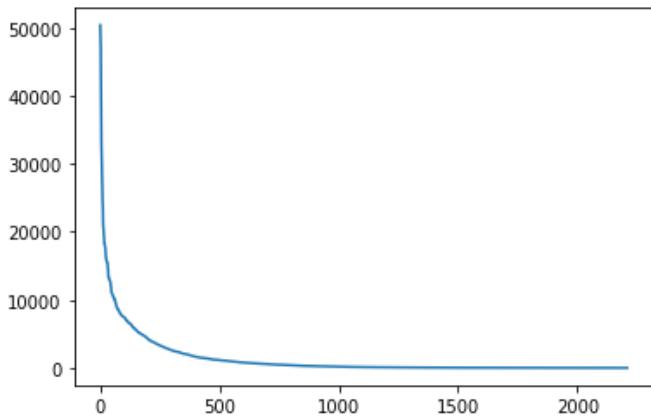
	dict	sex
0	阿安	男
1	阿彬	未知
2	阿斌	男
3	阿滨	男
4	阿冰	女
...
1145004	佐腾	男
1145005	佐威	男
1145006	佐为	男
1145007	佐樱	女
1145008	佐子	男
1145009 rows × 2 columns		

```
df = df[df.sex != "未知"]
names = df["dict"].values

# 计算单字频率
chars = [list(name) for name in names]
chars_flatten = list(itertools.chain(*chars))
freq = collections.Counter(chars_flatten)
freq = pd.DataFrame(freq.items(), columns=["char", "freq"])
freq = freq.sort_values(by="freq", ascending=False)
freq
```

	char	freq
636	王	50390
926	李	49078
1086	张	47089
1203	陈	41512
1394	刘	39986
...
2170	墙	1
2171	棱	1
2172	禹	1
2173	据	1
1962	莽	1
2210 rows × 2 columns		

```
# 频率分布  
char_rank = np.arange(freq.shape[0])  
char_freq = freq["freq"].values  
plt.plot(char_rank, char_freq)  
/ <matplotlib.lines.Line2D at 0x2271f622e50>
```



```
# 由常见字组成的名字数据
dict_size = 500
dict = listr(freq["char"].values[:dict_size])
dict_set = set(dict)
filtered = list(filter(lambda item: set(item[1]).issubset(dict_set), enumerate(names)))
ind = [idx for idx, name in filtered]
dat = df.iloc[ind]
dat["y"] = np.where(dat[« sex» ] == « 男», 0, 1)
dat

C:\Users\DELL\AppData\Local\Temp\ipykernel_11444\1483093082.py:8:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user\_guide/indexing.html#returning-a-view-versus-a-copy
dat["y"] = np.where(dat["sex"] == "男", 0, 1)
```

		dict	sex	y
401	艾爱	女	1	
402	艾爱国	男	0	
403	艾爱平	女	1	
404	艾安	男	0	
405	艾白	女	1	
...
1144983	左宗明	男	0	
1144984	左宗申	男	0	
1144985	左宗胜	男	0	
1144986	左宗堂	男	0	
1144989	左祖敏	男	0	

```
# 划分训练集和测试集
train = dat.sample(n=10000, random_state=123)
test = dat.sample(n=1000, random_state=321)

# One-hot 编码 # 给一个字符知道字符在字典中是第
几个
def char2index(char):
    return dict.index(char)

def name2index(name):
    return [char2index(char) for char in name]

def name2tensor(name):
    tensor = torch.zeros(len(name), 1, dict_size)
    for i, char in enumerate(name):
        tensor[i, 0, char2index(char)] = 1 # tensor 是一
    个三维数组 1*d
    return tensor

char2index("李")
```

```
name2index("李兴")  
[1, 76]
```

```
# 建立模型 RNN
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.h2o = nn.Linear(hidden_size, 1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), dim=1)
        hidden = torch.tanh(self.i2h(combined))
        output = torch.sigmoid(self.h2o(hidden))
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)
```

```
# 测试模型输入输出
n_hidden = 128
rnn = RNN(dict_size, n_hidden)
input = name2tensor("李兴")
hidden = rnn.init_hidden()
output, next_hidden = rnn(input[0], hidden)
print(output)
print(next_hidden)

tensor([[0.5039]], grad_fn=<SigmoidBackward0>)
tensor([[ 3.9781e-02,  2.8147e-02, -3.6969e-02,  1.3793e-02, -5.8002e-02,
        -1.4093e-02, -4.9785e-02,  5.8509e-02,  1.9775e-02, -3.1678e-03,
        2.2597e-02, -2.8383e-02,  1.5317e-02, -7.8083e-03,  3.6871e-03,
       -2.2734e-02,  2.4379e-02,  2.1663e-02, -3.6015e-02,  1.7164e-02,
        2.6967e-03, -3.3535e-02, -6.8762e-02,  4.8796e-03,  1.1621e-02,
        2.0426e-02,  1.7782e-02,  1.7175e-02, -3.1590e-02,  3.8382e-02,
       6.1524e-02,  1.8383e-02,  2.2076e-02, -2.0587e-02,  2.2252e-02,
       2.2557e-02, -1.6937e-02, -2.0218e-02,  3.8881e-03, -2.5070e-02,
      -7.1744e-02, -7.0943e-03,  5.6245e-02,  1.5940e-02,  3.1001e-02,
       2.4812e-02,  6.1076e-02, -4.8246e-02, -3.3198e-02, -9.2560e-03,
      3.8047e-02,  3.5928e-02, -4.0104e-02, -3.4599e-02, -4.1972e-02,
```

```

-1.7890e-02, 1.0700e-02, -4.8431e-02, 9.9346e-03, -1.0565e-02,
-5.0752e-02, 3.4963e-02, 1.5125e-02, -1.9436e-03, -3.8121e-03,
1.4695e-02, -1.4500e-02, -1.2248e-02, 6.4194e-03, 1.4094e-02,
-2.3500e-02, -2.6200e-02, 3.0263e-02, -2.7684e-02, 1.6095e-02,
-3.1570e-02, 1.3331e-02, -2.1008e-02, 8.9486e-03, -4.2247e-02,
-2.2698e-02, 4.3483e-03, -3.4317e-02, -4.5343e-02, 4.1974e-04,
8.1832e-04, 3.8085e-02, -4.8860e-03, -7.0697e-03, 2.8365e-03,
-5.9845e-02, 2.8062e-03, -1.1644e-02, 1.1859e-02, 1.3732e-02,
-4.6501e-02, -5.4161e-03, -2.0738e-02, -1.5813e-02, -1.1125e-02,
-1.4246e-02, -1.7573e-02, -2.8714e-02, -4.5249e-03, 1.5643e-02,
-1.7980e-02, 6.9877e-02, -6.8763e-02, 5.2029e-02, 2.6713e-04,
1.7348e-03, 1.6518e-02, 2.8594e-02, -1.2944e-02, -3.0261e-02,
-1.7221e-02, -1.9359e-02, 2.4823e-02, -6.5224e-03, 2.2660e-03,
-1.9247e-02, 1.4287e-02, 2.4821e-03, 7.2185e-02, 4.3747e-02,
-1.0037e-02, 6.1270e-05, -2.4147e-02]], grad_fn=<TanhBackward0>)

```

```

hidden = next_hidden
output, next_hidden = rnn(input[1], hidden)
print(output)
print(next_hidden)
tensor([[0.5067]], grad_fn=<SigmoidBackward0>)
tensor([[[-0.0138, -0.0338, -0.0071, -0.0495, -0.0489, 0.0478, -0.0397, 0.0367,
0.0769, -0.0320, -0.0305, -0.0017, -0.0545, 0.0306, 0.0659, 0.0190,
0.0228, 0.0104, -0.0789, 0.0160, 0.0141, -0.0240, -0.0419, -0.0370,
0.0067, 0.0328, 0.0679, 0.0059, -0.0010, 0.0177, 0.0046, 0.0168,
0.0300, -0.0210, -0.0067, 0.0691, 0.0207, -0.0203, -0.0077, 0.0113,
-0.0617, 0.0519, 0.0628, 0.0387, 0.0276, 0.0155, -0.0054, -0.0096,
-0.0235, 0.0400, -0.0412, -0.0157, -0.0056, 0.0225, -0.0019, -0.0065,
0.0347, -0.0210, 0.0049, -0.0299, -0.0184, -0.0031, 0.0002, -0.0479,
-0.0435, 0.0440, 0.0037, -0.0599, -0.0230, -0.0415, 0.0002, 0.0056,
0.0170, 0.0314, 0.0086, -0.0421, 0.0425, 0.0025, 0.0288, 0.0041,
0.0012, -0.0464, -0.0210, -0.0488, 0.0551, 0.0270, 0.0233, -0.0429,
0.0260, -0.0143, 0.0052, 0.0071, -0.0342, 0.0514, 0.0212, -0.0650,
0.0594, 0.0083, -0.0199, -0.0017, -0.0183, -0.0285, -0.0129, -0.0414,
-0.0522, 0.0131, 0.0285, -0.0131, 0.0461, 0.0350, -0.0201, 0.0287,
0.0151, 0.0019, 0.0093, 0.0485, -0.0136, 0.0216, 0.0407, -0.0112,
-0.0215, 0.0066, 0.0124, 0.0347, 0.0174, -0.0323, 0.0222, 0.0193]], grad_fn=<TanhBackward0>)

```

正式模型训练

```
np.random.seed(123)
```

```
torch.random.manual_seed(123)
```

```
n = train.shape[0]
```

```
n_hidden = 64
```

```
nepoch = 5
```

```
bs = 100
```

```

rnn = RNN(dict_size, n_hidden)
opt = torch.optim.Adam(rnn.parameters(), lr=0.001)
train_ind = np.arange(n)
losses = []

t1 = time.time()
for k in range(nepoch): # 数据遍历多少次
    np.random.shuffle(train_ind)
    # Mini-batch 循环
    for j in range(0, n, bs): # minibatch
        mb = train.iloc[train_ind[j]: (j + bs)]]
        mb_size = mb.shape[0]
        loss = 0.0
        # 对 Mini-batch 中的每个名字进行循环
        # 如何计算损失函数 # 对每一个样本点单独写循环
        for i in range(mb_size):
            name = mb["dict"].values[i]
            input = name2tensor(name)
            hidden = rnn.init_hidden()
            y = mb["y"].values[i]
            # 对名字中的每个字进行循环
            for s in range(input.shape[0]):
                output, hidden = rnn(input[s], hidden)
                loss = loss - y * torch.log(output) - (1.0 - y) * torch.log(1.0 - output)

            loss = loss / mb_size
            opt.zero_grad()
            loss.backward()
            opt.step()

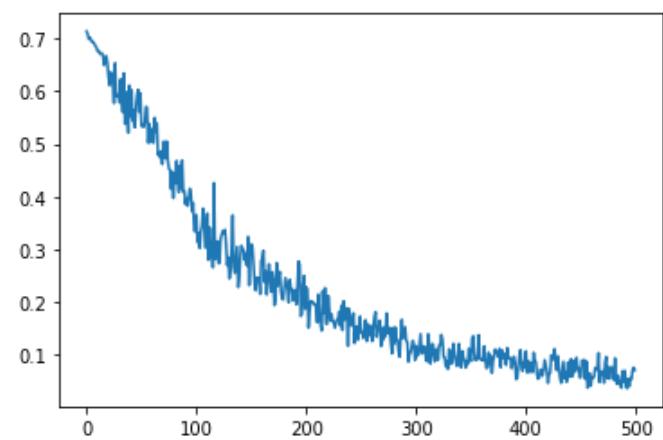
            losses.append(loss.item())
        if j // bs % 10 == 0:
            print(f'epoch {k}, batch {j // bs}, loss = {loss.item()}')
    t2 = time.time()
    print(t2 - t1)
epoch 0, batch 0, loss = 0.7133022546768188
epoch 0, batch 10, loss = 0.6761780381202698
epoch 0, batch 20, loss = 0.6261805891990662
epoch 0, batch 30, loss = 0.5781686305999756
epoch 0, batch 40, loss = 0.5512726902961731
epoch 0, batch 50, loss = 0.5336638689041138
epoch 0, batch 60, loss = 0.5058482885360718
epoch 0, batch 70, loss = 0.5043060779571533
epoch 0, batch 80, loss = 0.44378843903541565
epoch 0, batch 90, loss = 0.38597309589385986

```

```

epoch 1, batch 0, loss = 0.3651692569255829
epoch 1, batch 10, loss = 0.3681095838546753
epoch 1, batch 20, loss = 0.27375924587249756
epoch 1, batch 30, loss = 0.24464084208011627
epoch 1, batch 40, loss = 0.28715941309928894
epoch 1, batch 50, loss = 0.3098525106906891
epoch 1, batch 60, loss = 0.28342700004577637
epoch 1, batch 70, loss = 0.24363455176353455
epoch 1, batch 80, loss = 0.2275119423866272
epoch 1, batch 90, loss = 0.22295738756656647
epoch 2, batch 0, loss = 0.22859273850917816
epoch 2, batch 10, loss = 0.1615162491798401
epoch 2, batch 20, loss = 0.21181495487689972
epoch 2, batch 30, loss = 0.18197284638881683
epoch 2, batch 40, loss = 0.16902418434619904
epoch 2, batch 50, loss = 0.14238764345645905
epoch 2, batch 60, loss = 0.1258188784122467
epoch 2, batch 70, loss = 0.1283814162015915
epoch 2, batch 80, loss = 0.12115246057510376
epoch 2, batch 90, loss = 0.15273848176002502
epoch 3, batch 0, loss = 0.11695155501365662
epoch 3, batch 10, loss = 0.08770526200532913
epoch 3, batch 20, loss = 0.09042107313871384
epoch 3, batch 30, loss = 0.11094540357589722
epoch 3, batch 40, loss = 0.0924876406788826
epoch 3, batch 50, loss = 0.08804446458816528
epoch 3, batch 60, loss = 0.0918993353843689
epoch 3, batch 70, loss = 0.08881078660488129
epoch 3, batch 80, loss = 0.09335130453109741
epoch 3, batch 90, loss = 0.0866888165473938
epoch 4, batch 0, loss = 0.1070617288351059
epoch 4, batch 10, loss = 0.06310055404901505
epoch 4, batch 20, loss = 0.046603359282016754
epoch 4, batch 30, loss = 0.07689378410577774
epoch 4, batch 40, loss = 0.05823833495378494
epoch 4, batch 50, loss = 0.05822055786848068
epoch 4, batch 60, loss = 0.05797842517495155
epoch 4, batch 70, loss = 0.06920641660690308
epoch 4, batch 80, loss = 0.05269988998770714
epoch 4, batch 90, loss = 0.06993500888347626
106.07335042953491

```



```

# 对测试集预测
ntest = test.shape[0]
true_label = test["y"].values
pred = np.zeros(ntest)
rnn.eval()
for i in range(ntest):
    input = name2tensor(test["dict"].values[i])
    hidden = rnn.init_hidden()
    with torch.no_grad():
        for s in range(input.shape[0]):
            output, hidden = rnn(input[s], hidden)
            pred[i] = output.item()
    if i % 100 == 0:
        print(f"processed {i}")
    loss = -np.mean(true_label * np.log(pred) + (1.0 - true_label) * np.log(1.0 - pred))
    print(loss)
    pred_label = (pred > 0.5).astype(int)
    print(np.mean(pred_label == true_label))

processed 0
processed 100
processed 200
processed 300
processed 400
processed 500
processed 600
processed 700
processed 800
processed 900
0.07498759074511005
0.979

```

```

plt.plot(losses)
[<matplotlib.lines.Line2D at 0x2272a655a30>]

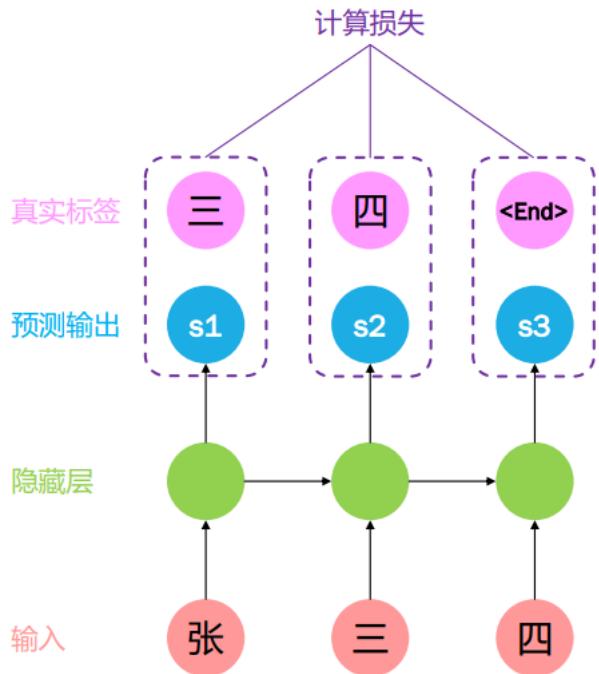
```

```

# 随机取样
np.random.seed(123)
torch.random.manual_seed(123)
ind = np.random.choice(ntest, 10)

```

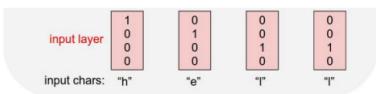

5、模型原理



Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



Fei-Fei Li, Ranjay Krishna, Danfei Xu

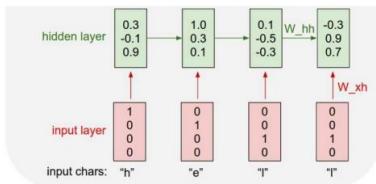
Lecture 10 - 40 May 7, 2020

Example: Character-level Language Model

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



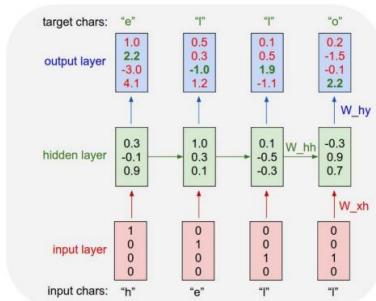
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 41 May 7, 2020

Example: Character-level Language Model

Vocabulary:
[h,e,l,o]

Example training sequence:
"hello"



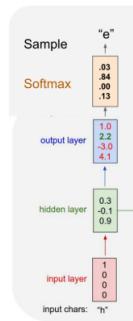
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 42 May 7, 2020

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



Fei-Fei Li, Ranjay Krishna, Danfei Xu

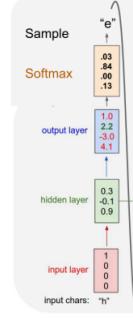
Lecture 10 - 43

May 7, 2020

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



Fei-Fei Li, Ranjay Krishna, Danfei Xu

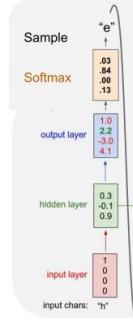
Lecture 10 - 44

May 7, 2020

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



Fei-Fei Li, Ranjay Krishna, Danfei Xu

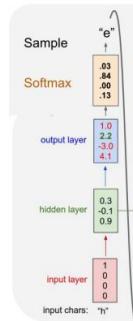
Lecture 10 - 45

May 7, 2020

Example: Character-level Language Model Sampling

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model



Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 46

May 7, 2020

6、预处理

1) 处于演示目的

2) 只考虑数据中最高频的 50 个字

3) 建立字典

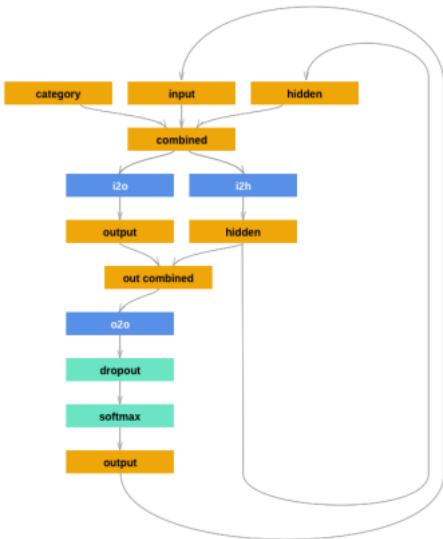
```
array(['尔', '斯', '拉', '克', '特', '德', '尼', '夫', '利', '科', '里', '卡', '亚',  
    '罗', '奇', '格', '布', '基', '维', '阿', '马', '伊', '奥', '纳', '雷', '洛',  
    '蒂', '塔', '诺', '瓦', '托', '莱', '巴', '内', '贝', '普', '萨', '娃', '耶',  
    '迪', '米', '西', '勒', '什', '达', '埃', '加', '库', '帕', '韦'], dtype='<U1')
```

4) 还需要在字典中加入一个特殊字符代表字符串结束

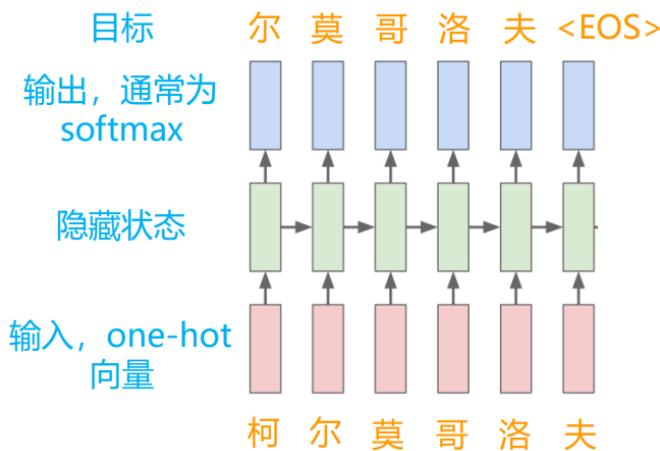
7、建立模型

1) 参考:

https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html



8、损失函数



9、训练结果

- 1) 比用序列预测分类更复杂
- 2) GPU 上训练~12 分钟

[‘雷斯塔’ ‘伊马斯’ ‘瓦尔德’ ‘耶尔维’ ‘西尔韦斯特里’ ‘布拉伊科夫斯基’ ‘库尔蒂耶’ ‘迪亚科’ ‘拉斯特里’ ‘罗斯托’ ‘维尔斯’ ‘马尔基奥’ ‘埃斯特拉德’ ‘维尔斯’ ‘韦尔尼’ ‘维尔斯’ ‘马尔基奥’ ‘格拉斯科’ ‘库尔蒂耶’ ‘尔德’ ‘萨尔瓦尼’ ‘维尔斯’ ‘巴尔巴里尼’ ‘内斯托拉’ ‘克拉斯尼奇’ ‘伊马斯’ ‘德拉斯科’ ‘诺尔贝格’ ‘伊马斯’ ‘德拉斯科’ ‘马尔基奥’ ‘勒布罗’ ‘达尔马斯’ ‘莱斯科’ ‘瓦尔德’ ‘西尔韦斯特里’ ‘罗斯托’ ‘托尔托拉’ ‘米尔科’ ‘特里斯塔尼’ ‘耶尔维’ ‘罗斯托’ ‘科’ ‘贝尔托利尼’ ‘克拉斯尼奇’ ‘迪亚科’ ‘利亚尔迪’ ‘亚尔马’ ‘塔尔迪’ ‘耶尔维’]

三、改进 RNN

1、优缺点

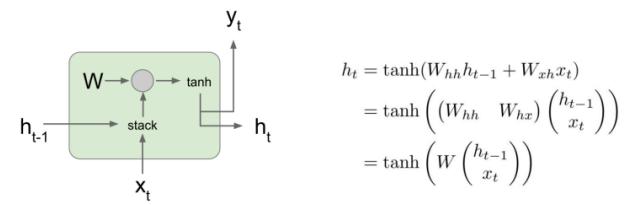
- 1) 相比于前馈神经网络和 CNN, RNN 有其独特的性质
- 2) 优点
 - a. 处理任意长的序列数据
 - b. 利用历史信息
 - c. 参数数量不随序列变长而增加
- 3) 缺点
 - a. 序列很长时计算量非常大
 - b. 梯度消失、爆炸问题

2、反向传播

- 1) 要理解 RNN 的局限, 就需要明白它的反向传播原理

Vanilla RNN Gradient Flow

Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al., “On the difficulty of training recurrent neural networks”, ICML 2013



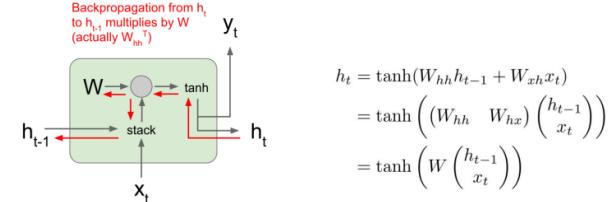
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 103 May 7, 2020

资料来自 <http://cs231n.stanford.edu/>

Vanilla RNN Gradient Flow

Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al., “On the difficulty of training recurrent neural networks”, ICML 2013

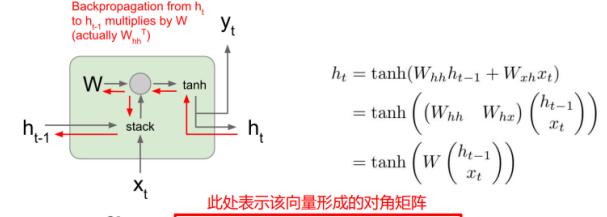


Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 104 May 7, 2020

Vanilla RNN Gradient Flow

Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al., “On the difficulty of training recurrent neural networks”, ICML 2013



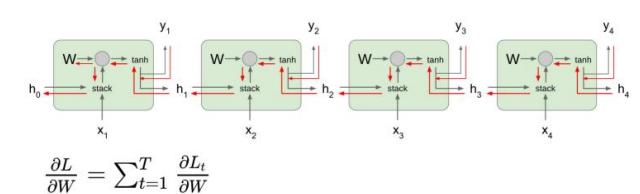
$$\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh}h_{t-1} + W_{xh}x_t)W_{hh}$$

Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 105 May 7, 2020

Vanilla RNN Gradient Flow

Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al., “On the difficulty of training recurrent neural networks”, ICML 2013



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

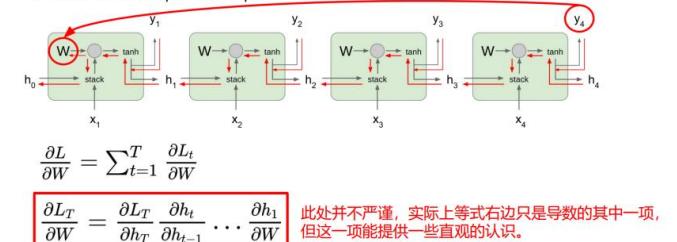
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 106 May 7, 2020

Vanilla RNN Gradient Flow

Gradients over multiple time steps:

Bengio et al., “Learning long-term dependencies with gradient descent is difficult”, IEEE Transactions on Neural Networks, 1994
Pascanu et al., “On the difficulty of training recurrent neural networks”, ICML 2013



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_T}{\partial h_{T-1}} \cdots \frac{\partial h_1}{\partial W}$$

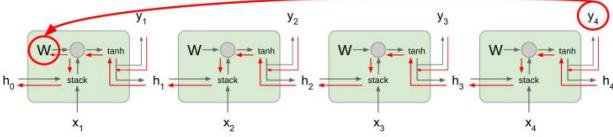
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 107 May 7, 2020

此处并不严谨，实际上等式右边只是导数中的其中一项，但这一项能提供一些直观的认识。

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

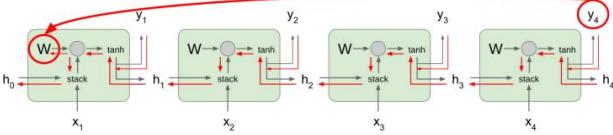
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 108

May 7, 2020

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



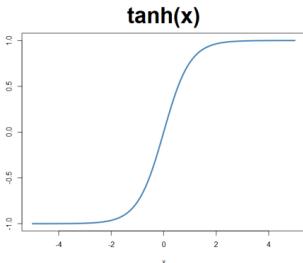
$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W} \quad \boxed{\frac{\partial h_t}{\partial h_{t-1}} = \tanh'(W_{hh} h_{t-1} + W_{xh} x_t) W_{hh}}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_1}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \frac{\partial h_t}{\partial h_{t-1}} \right) \frac{\partial h_1}{\partial W}$$

Fei-Fei Li, Ranjay Krishna, Danfei Xu

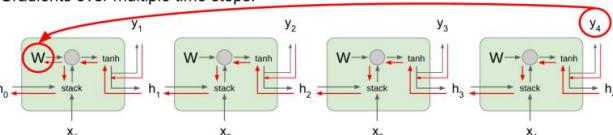
Lecture 10 - 109

May 7, 2020



Vanilla RNN Gradient Flow

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W} \quad \text{Almost always } < 1 \quad \text{Vanishing gradients}$$

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \left(\prod_{t=2}^T \tanh'(W_{hh} h_{t-1} + W_{xh} x_t) \right) W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

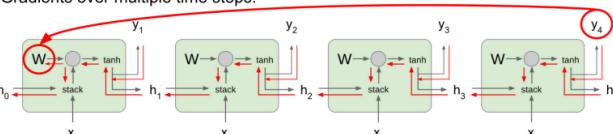
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 110

May 7, 2020

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W} \quad \text{What if we assumed no non-linearity?}$$

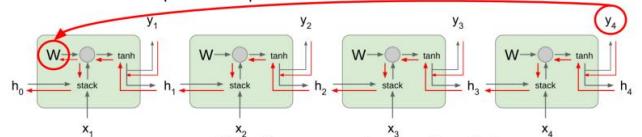
Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 111

May 7, 2020

Vanilla RNN Gradient Flow

Gradients over multiple time steps:



$$\frac{\partial L}{\partial W} = \sum_{t=1}^T \frac{\partial L_t}{\partial W}$$

Largest singular value > 1:
Exploding gradients

$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} \boxed{W_{in}^{T-1}} \frac{\partial h_1}{\partial W}$ Largest singular value < 1:
Vanishing gradients

Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 10 - 112

May 7, 2020

3. 改进方法

- 1) 梯度爆炸: 对梯度的上限进行截断
- 2) 梯度消失: 更改 RNN 架构

4、LSTM

- 1) 长短期记忆神经网络
- 2) Long Short-Term Memory
- 3) Hochreiter and Schmidhuber (1997). Long Short Term Memory, Neural Computation.

4) Vanilla RNN

$$h_t = \tanh \left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix} \right)$$

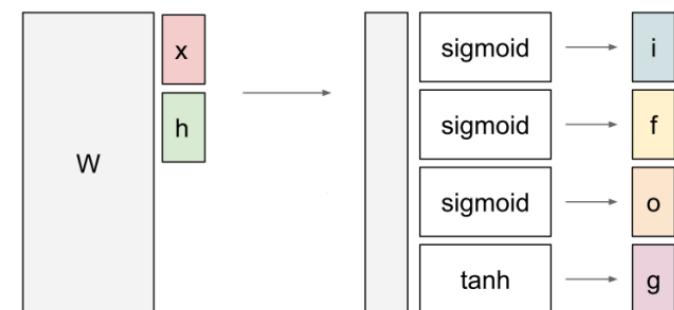
LSTM

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

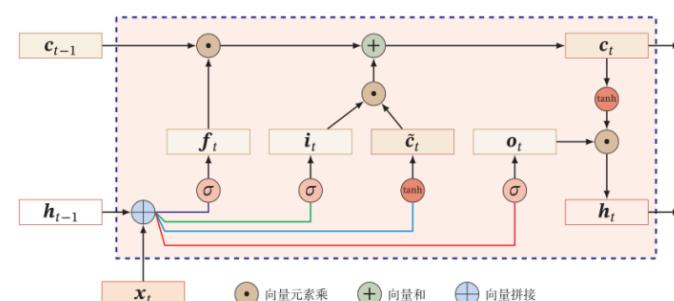
5)



$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

6)



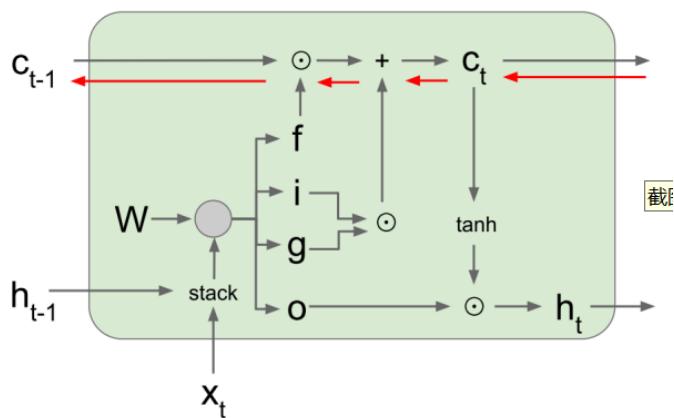
a. i : 是否写入当前单元

b. g/\tilde{c} : 多大程度上写入当前单元

c. f : 是否遗忘上一个单元

d. o : 多大程度上将单元转成隐藏状态

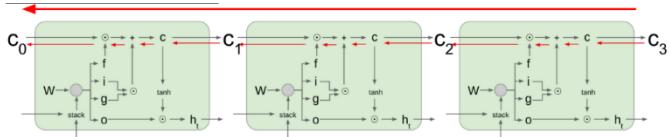
7) 梯度从 c_t 传向 c_{t-1} 时只牵涉到与 f 的逐元素相乘, 没有直接的矩阵乘法



8) 可以更好地对梯度进行远距离传播

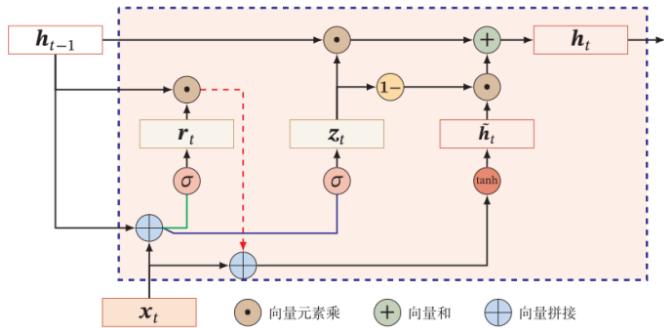
“没有中间商赚差价”

可以类比于残差神经网络



5、GRU

1)



重置门

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

$$\tilde{h}_t = \tanh(W_c x_t + U(r_t \odot h_{t-1}) + b_c)$$

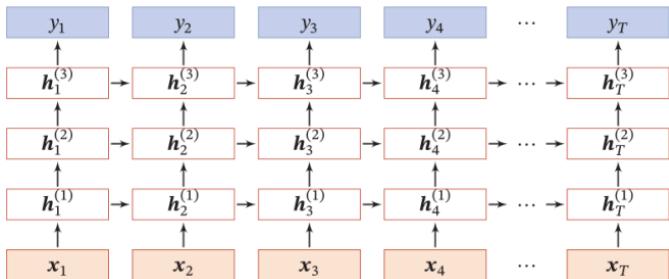
$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t$$

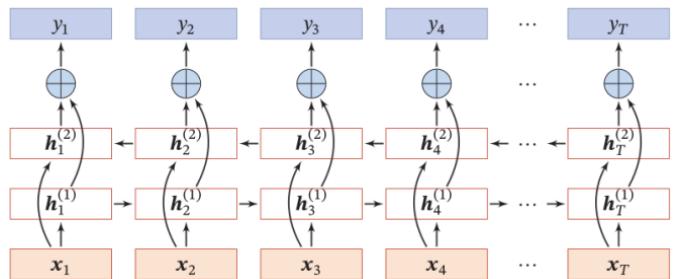
更新门

6、其他架构

1) 多层堆叠 RNN



2) 双向 RNN



lec12-gen_name_en

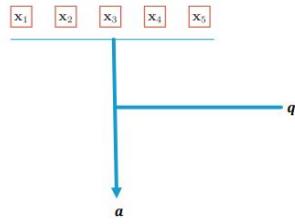
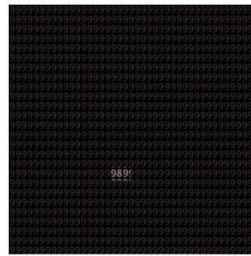
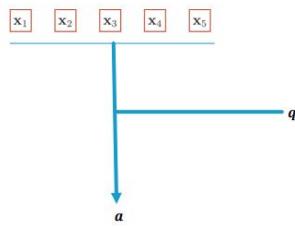
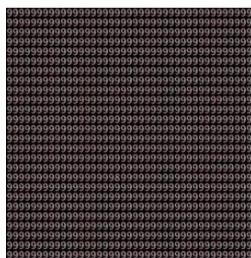
import time

import

DL-lec13

今天的主题

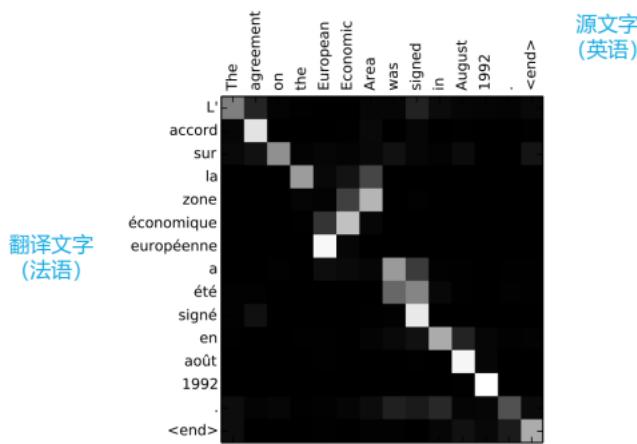
1. 注意力机制与 Transformer
2. 大语言模型初探



一、注意力机制与 Transformer

1、注意力机制

- 1) 在机器翻译中，注意力机制可以理解为一种文字“对齐”的方法



- 2) 实现方法

- a. 基于 Neural Machine Translation by jointly Learning to Align and Translate. ICLR 2015
- b. 回顾翻译文字的生成机制
- c. $p(y_i | y_1, \dots, y_{i-1}, x) = g(y_{i-1}, s_i, c_i)$

- d. y_i 是当前翻译出的单词

- e. s_i 是 RNN 当前的隐藏层向量, $s_i = f(s_{i-1}, y_{i-1}, c_i)$

- f. c_i 是当前上下文向量, 相当于对输入序列进行压缩后的结果

- g. 注意力机制体现在 c_i 的选择和构建上

- h. 整体而言, c_i 是对输入序列隐藏层的加权平均

- i. $c_i = \sum_{j=1}^T a_{ij} h_j$

- j. 权重 a_{ij} 代表了第 j 个输入文字对当前翻译输出的“重要性”

- k. $(a_{i1}, \dots, a_{iT}) = \text{softmax}(e_{i1}, \dots, e_{iT})$

- l. $e_{ij} = a(s_{i-1}, h_j)$ 是一个打分函数, 参数数量固定

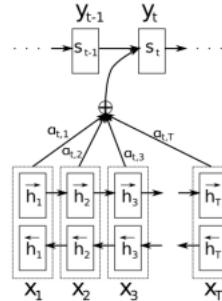


Figure 1: The graphical illustration of the proposed mode trying to generate the t -th tar get word y_t given a sourc sentence (x_1, x_2, \dots, x_T) .

3) 打分函数

- a. 打分函数有不同的形式

- b. 但核心在于参数数量固定, 不随输入序列长度影响

- a) 加性模型 $s(x, q) = v^T \tanh(Wx + Uq)$

- b) 点积模型 $s(x, q) = x^T q$

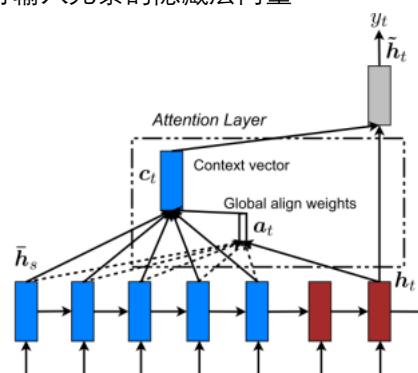
- c) 缩放点积模型 $s(x, q) = \frac{x^T q}{\sqrt{D}}$

- d) 双线性模型 $s(x, q) = x^T W q$

4) 全局注意力

- a. Effective Approaches to Attention-based Neural Machine Translation 一文对上述结构做了细微改动, 并提出“全局注意力”的概念

- b. “全局”的含义在于, 生成上下文向量 c_t 时利用到了所有输入元素的隐藏层向量

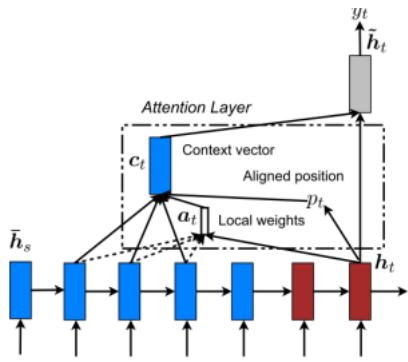


5) 局部注意力

- a. 与全局相对的, 是局部注意力

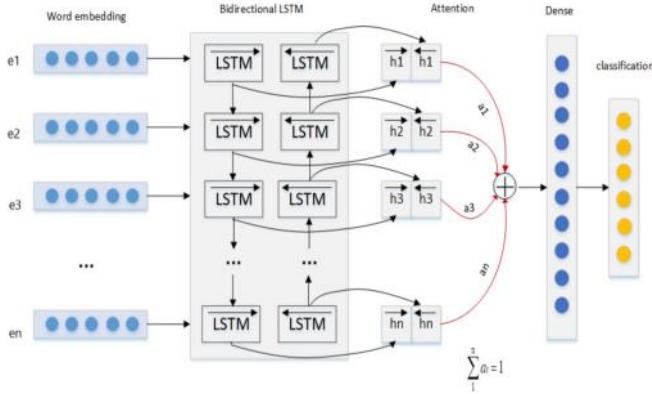
- b. a_t 计算的范围是一个固定宽度的滑动窗口

- c. 好处在于减少了计算量



6) 其他应用

a. 文本分类



b. 图像标注

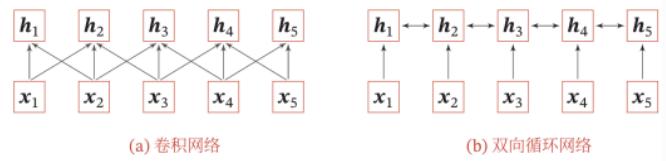
给定文字，识别出图片对应的区域



7) Attention Is All You Need (?)

8) Attention 热潮

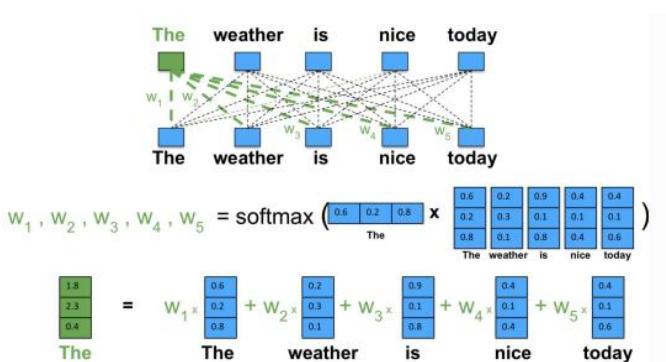
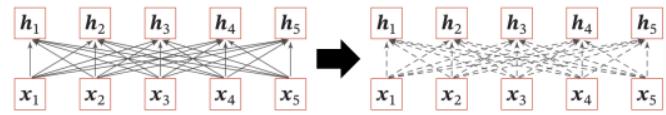
- a. 自注意力 (Self-attention)
- b. 多头注意力 (Multi-attention)
- c. KQV 模式 (Key-Query-Value)
- d. Transformer
- e. BERT
- f. GPT
- g. Vision Transformer
- 9) 自注意力
- a. 自注意力模型进一步放松了 RNN 的限制
- b. RNN 的主要作用在于利用固定数量的参数将不定长的序列映射为对应的隐藏表示
- c. 卷积网络也有类似的效果



d. 然而 RNN 和卷积主要利用的是局部信息

e. 全连接网络可以跨越较长的距离，但参数数量不固定

f. 自注意力模型可以看作是一种特殊的全连接层，权重由注意力机制生成，参数数量固定



10) KQV 模式

- a. 注意力机制还可以进一步抽象
- b. 输入序列的隐藏表示由三部分组成
- c. Key-Query-Value
- d. 回顾
- e. 上下文向量 $c_i = \sum_{j=1}^T a_{ij} h_j$
- f. 权重 $(a_{i1}, \dots, a_{iT}) = \text{softmax}(e_{i1}, \dots, e_{iT})$
- g. 打分函数 $e_{ij} = a(s_{i-1}, h_j)$

■ 回顾

隐藏表示是 Value 的加权平均

$$\text{上下文向量 } c_i = \sum_{j=1}^T a_{ij} h_j$$

$$\text{权重 } (a_{i1}, \dots, a_{iT}) = \text{softmax}(e_{i1}, \dots, e_{iT})$$

$$\text{打分函数 } e_{ij} = a(s_{i-1}, h_j)$$

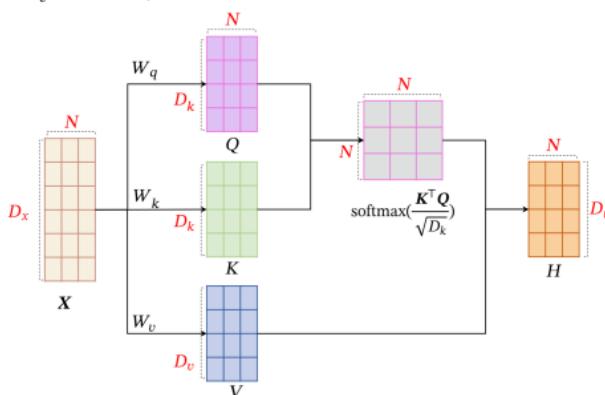
将 Query 去和 Key 进行匹配

$$Q = W_q X \in \mathbb{R}^{D_k \times N},$$

$$K = W_k X \in \mathbb{R}^{D_k \times N},$$

$$V = W_v X \in \mathbb{R}^{D_v \times N},$$

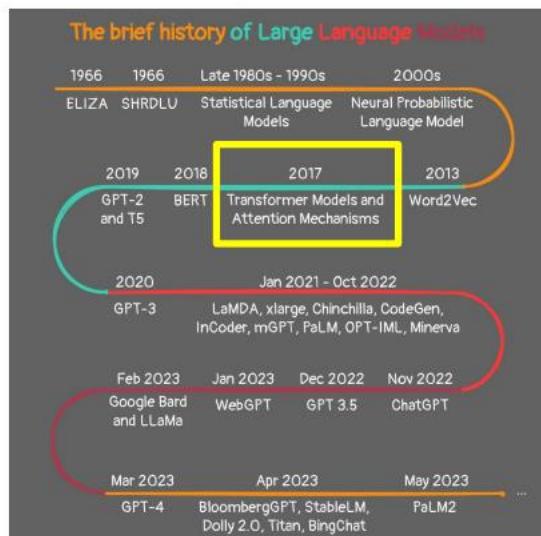
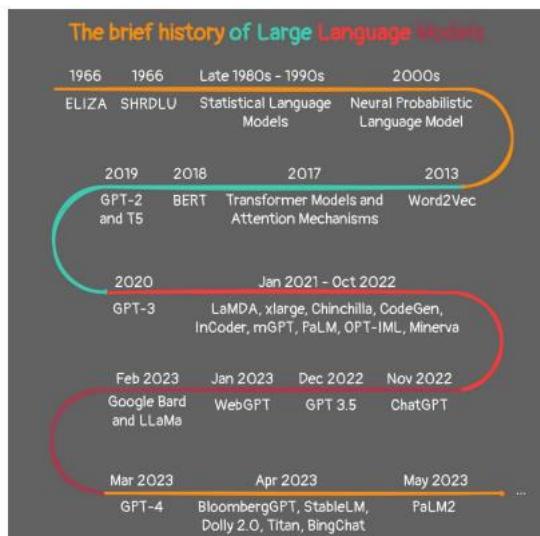
$$H = V \text{ softmax}\left(\frac{K^T Q}{\sqrt{D_k}}\right),$$



10) 思考

- a. 为什么要想尽办法计算隐藏表示？
- b. 为了更好地利用数据和问题的结构
- c. 不同的模型架构可能理论表达能力是等价的
- d. 但实际中要根据数据的特征进行设计

二、大语言模型初探



1. Transformer

- 1) 2017 年，一篇名为 Attention is all you need 的文章标志着 Transformer 横空出世
- 2) Transformer 可以看成是利用 Attention 来涉及的一种网络结构
- 3) 成为当今几乎所有大语言模型的基本构成单位

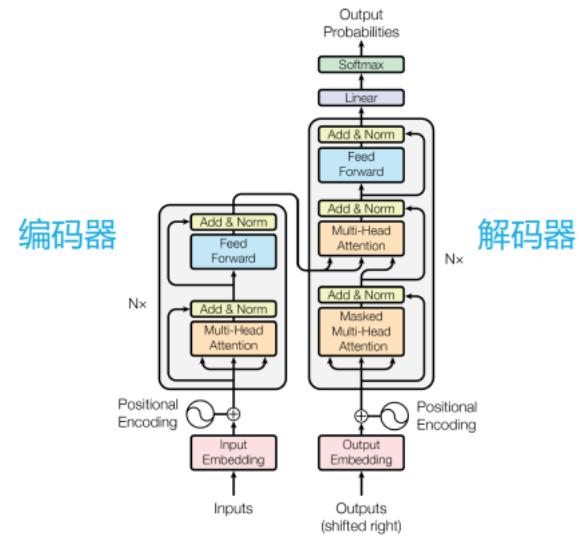
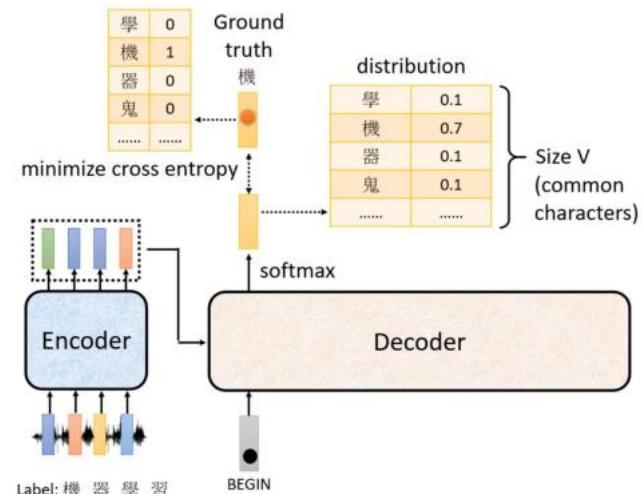
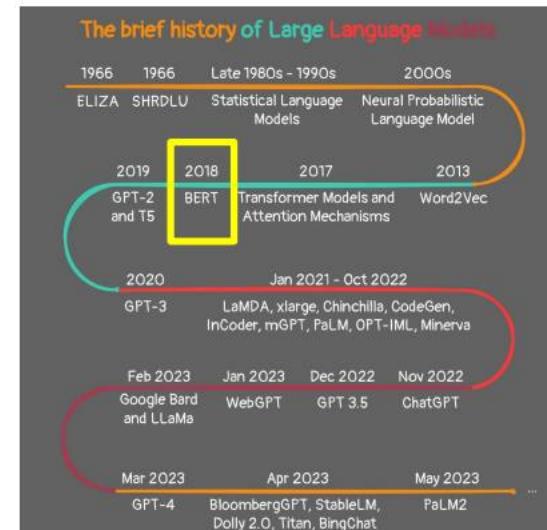


Figure 1: The Transformer - model architecture.



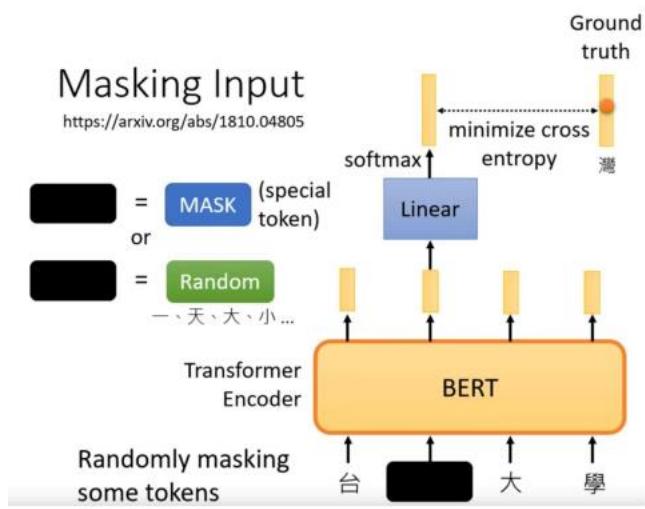
4) 扩展阅读

[Transformer 论文逐段精读【论文精读】_哔哩哔哩_bilibili](#)

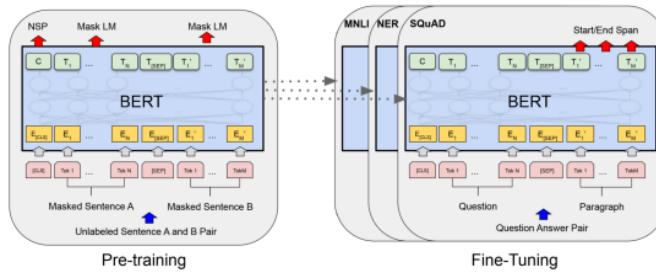


2. BERT

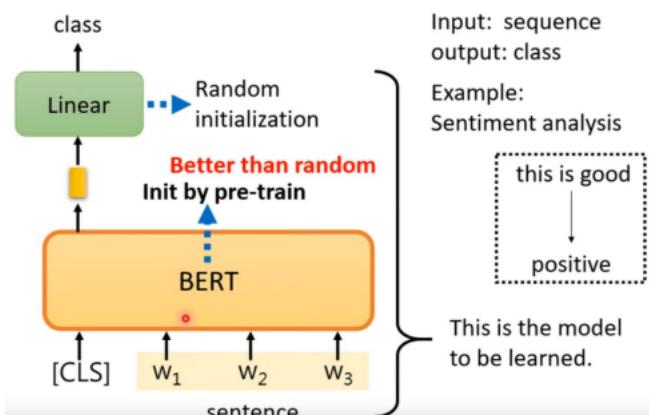
- 1) BERT 的全称为 Bidirectional Encoder Representations from Transformers
- 2) 同样是利用了 Transformer 结构构建出的语言模型
- 3) BERT 只利用了 Transformer 的编码器
- 4) 采用自监督学习的方法
- 5) “完形填空”: 随机选择一些词语, 将其盖住 (MASK)
- 6) 自监督学习的目的就是用文本剩余的部分来预测被盖住的词语
- 7) 使用双向信息: 不一定从左读到右



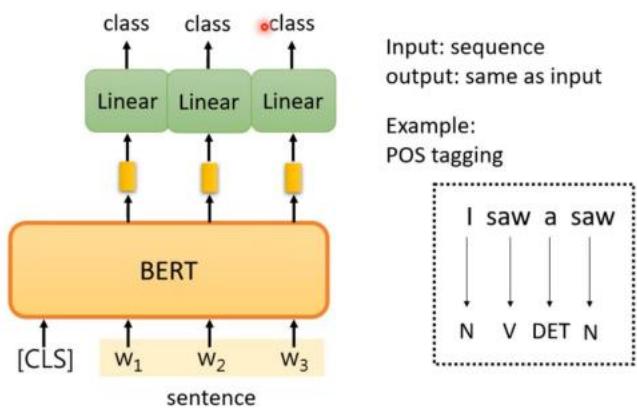
- 8) BERT 极大地普及了“预训练+微调”这一大模型的使用模式
- 9) 在训练阶段获取文本的嵌入向量表达
- 10) 微调阶段用向量表达完成各项 NLP 任务



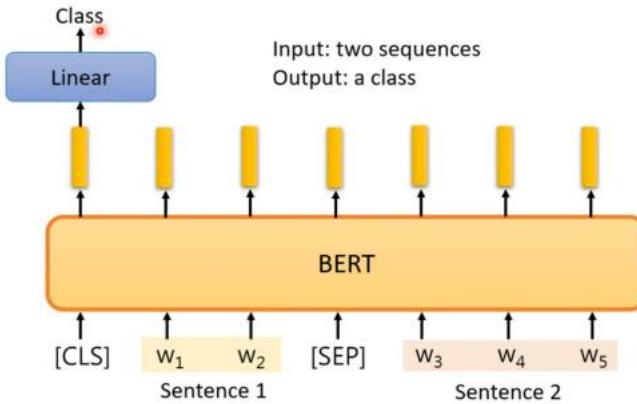
How to use BERT – Case 1



How to use BERT – Case 2

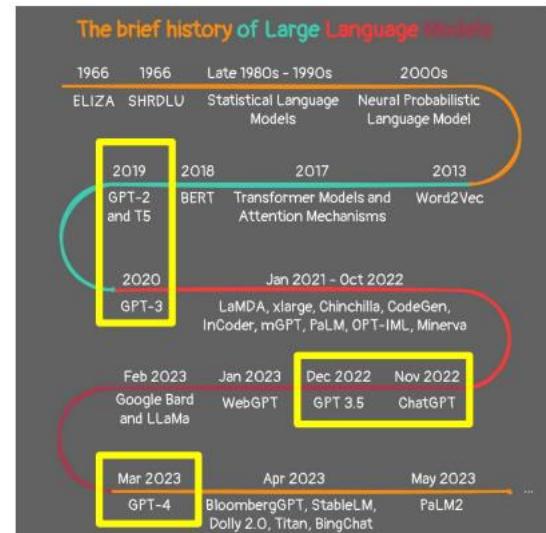


How to use BERT – Case 3



10) 扩展阅读

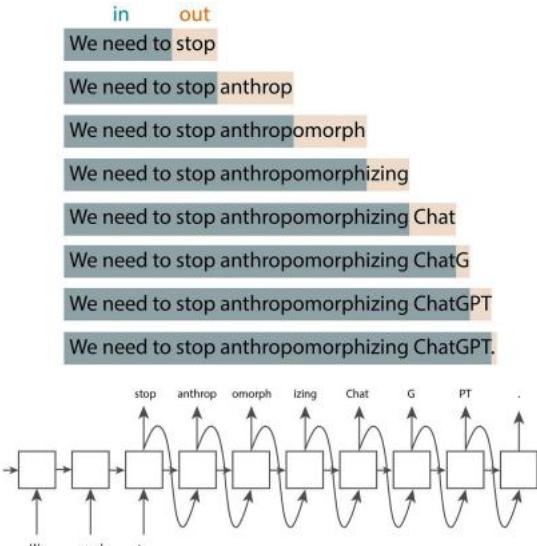
[BERT 论文逐段精读【论文精读】_哔哩哔哩_bilibili](#)



3. GPT

- 1) Generative Pre-trained Transformer
- 2) 由 OpenAI 主导开发的系列模型
- 3) GPT-1, GPT-2, GPT-3, ChatGPT, GPT-4...
- 4) BERT 解决了 NLP 任务的思想是“预训练+微调”
- 5) 其中微调需要针对具体任务进行设计
- 6) 而 GPT 的路线是用自然语言本身来对模型“下指令”
- 7) 由此产生了以 Prompt (提示词) 为代表的语言模型使用方法——把各类 NLP 任务转变成问答形式

- 7) 从训练的角度来说
- 8) BERT 的核心思想是“完形填空”
- 9) 而 GPT 的路线就是“文字接龙”
- 10) BERT 只利用了 Transformer 的编码器
- 11) GPT 只利用了 Transformer 的解码器



- 12) 无论模型如何演变，核心的统计准则往往是朴素的

3.1 Unsupervised pre-training

Given an unsupervised corpus of tokens $\mathcal{U} = \{u_1, \dots, u_n\}$, we use a standard language modeling objective to maximize the following likelihood:

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \dots, u_{i-1}; \Theta) \quad (1)$$

where k is the size of the context window, and the conditional probability P is modeled using a neural network with parameters Θ . These parameters are trained using stochastic gradient descent [51].

In our experiments, we use a multi-layer Transformer decoder [34] for the language model, which is a variant of the transformer [62]. This model applies a multi-headed self-attention operation over the input context tokens followed by position-wise feedforward layers to produce an output distribution over target tokens:

$$\begin{aligned} h_0 &= UW_e + W_p \\ h_l &= \text{transformer_block}(h_{l-1}) \forall l \in [1, n] \\ P(u) &= \text{softmax}(h_n W_e^T) \end{aligned} \quad (2)$$

where $U = (u_{-k}, \dots, u_{-1})$ is the context vector of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

- 13) GPT 系列的参数量以惊人的速度在不断增长



- 14) 早期的 GPT 模型 (1 和 2) 并没有显著比 BERT 更好，事实上在很多地方不如 BERT
- 15) 但令人惊讶的是，GPT 的“大力出奇迹”模式终于在 GPT-3 上大放异彩
- 16) 后续 OpenAI 推出的 ChatGPT 更是将公众对该系列模型的关注度提升到了顶峰
- 17) 不过与之相关的技术细节也公开得越来越少
- 18) 有猜测说 ChatGPT 在 GPT-3 的基础上融合了有

监督微调、强化学习、思维链等技术

19) GPT-4 被认为可能使用了专家模型

[GPT, GPT-2, GPT-3 论文精读【论文精读】_哔哩哔哩 bilibili](#)

[lec13-language-model.ipynb](#)

DL-lec14

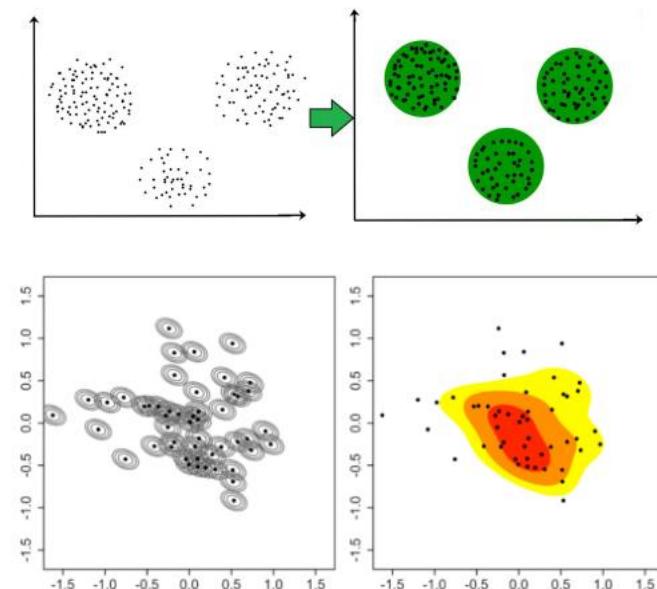
今天的主题

1. 深度生成模型
2. 变分自编码器
3. 生成对抗网络

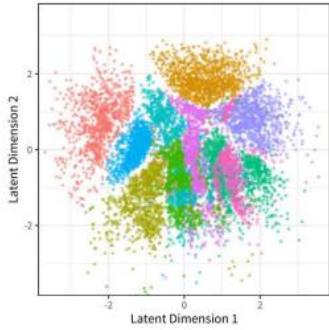
一、深度生成模型

1、无监督学习

- 1) 数据: X , 没有标记“标签”
- 2) 目标: 了解数据的分布或结构

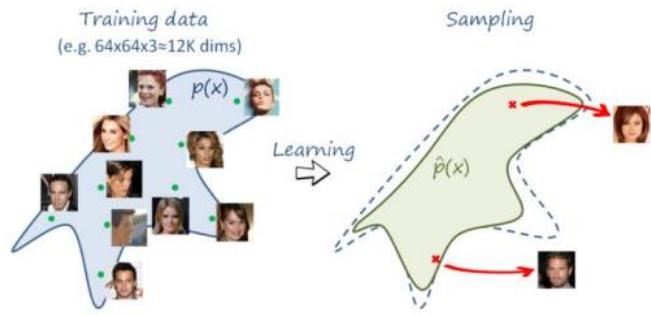


4 3 9 9 3 9 5 6 5 7
 9 6 4 1 0 0 1 6 8 4
 8 7 6 9 6 1 9 3 8 0
 9 8 9 5 4 2 6 1 5 4
 2 8 1 2 1 1 4 8 3 9
 0 8 1 6 1 3 7 4 7 2
 0 5 6 4 7 2 9 9 3 3
 5 0 1 9 1 1 4 6 9 0
 8 9 3 4 7 4 6 9 3 3
 8 4 3 1 1 7 7 2 6 4



2、生成模型

1) 给定样本数据，得到一个“生成器”，用以生成同分布的样本



- 2) 经典统计中的密度估计就是一种生成模型
- 3) 利用了神经网络结构的生成模型一般称为深度生成模型
- 4) 变分自编码器 (variational autoencoder)
- 5) 生成对抗网络 (generative adversarial network)
- 6) 流模型 (normalizing flow)
- 7) 扩散模型 (diffusion model)
- 8) 作用
 - a. 理解数据的统计分布 (统计建模的核心问题)
 - b. 数据扩充, 图像生成, 视觉艺术
 - c. 统计模拟
 - d. 提取特征

二、变分自编码器 (从统计学的角度)

1、VAE

- 1) Variational autoencoders, VAE
- 2) VAE 的文献通常是以编码器/解码器的视角来介绍该模型
- 3) 我们从更“统计”的角度来引入和理解
- 4) 建模目标
 - a. 回顾生成模型的建模目标
 - b. 给定数据 X_1, \dots, X_n , 希望刻画其分布 $p(x)$
 - c. 如何构建 $p(x)$, 使其能拟合复杂的数据?
- 5) 传统方法
 - a. 假设 $p(x)$ 来自某个分布族 $p_\theta(x)$
 - b. 例如正态分布 $N(\mu, \Sigma), \theta = (\mu, \Sigma)$
 - c. 缺点: 形式受限, 不够灵活
- 6) 核心思想 1
 - a. 混合分布建模

b. 将 $p_\theta(x)$ 设定为两个分布的混合

$$c. p_\theta(x) = \int \pi(z)p_\theta(x|z) dz$$

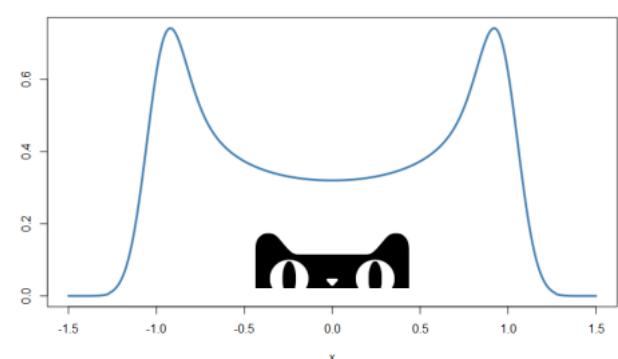
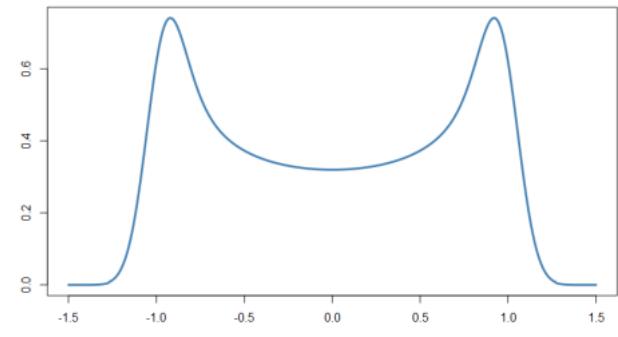
d. 简单分布+简单分布=>复杂分布

e. 简单分布+复杂分布=>更复杂的分布

f. 混合分布

$$Z \sim N(0, 1)$$

$$X | \{Z = z\} \sim N(\sin(z), 0.01)$$



7) VAE 建模

$$a. p_\theta(x) = \int \pi(z)p_\theta(x|z) dz$$

b. 在 VAE 中, $\pi(z)$ 固定为标准正态 $N(0, I)$

c. $p_\theta(x|z)$ 利用神经网络刻画

d. 连续数据: $p_\theta(x|z) = N(g_\theta(z), \tau^2 I)$, $g_\theta(z)$ 为神经网络, τ 为常数/超参数

e. 0-1 数据: $p_\theta(x|z) = \text{Bernoulli}(\sigma(g_\theta(z)))$, $g_\theta(z)$ 为神经网络, σ 为 Sigmoid 函数

8) 参数估计

a. 设定好模型后, 接下来的工作是估计参数 θ

b. 经典方法: 极大似然估计

$$c. l(\theta; x) = \log p_\theta(x)$$

$$d. \max_{\theta} n^{-1} \sum_{i=1}^n l(\theta; X_i)$$

e. $p_\theta(x) = \int \pi(z)p_\theta(x|z) dz$ 是一个复杂的积分, $l(\theta; x)$ 难以计算!

9) 核心思想 2

a. 似然函数不等式

b. 《关于 VAE, 记住这个公式就够了》

c. 反正大概率记不住

d.

