

神经网络基础





目录

Contents



- ◆ 神经网络
- ◆ 损失函数
- ◆ 网络优化方法
- ◆ 正则化方法
- ◆ 案例-价格分类案例

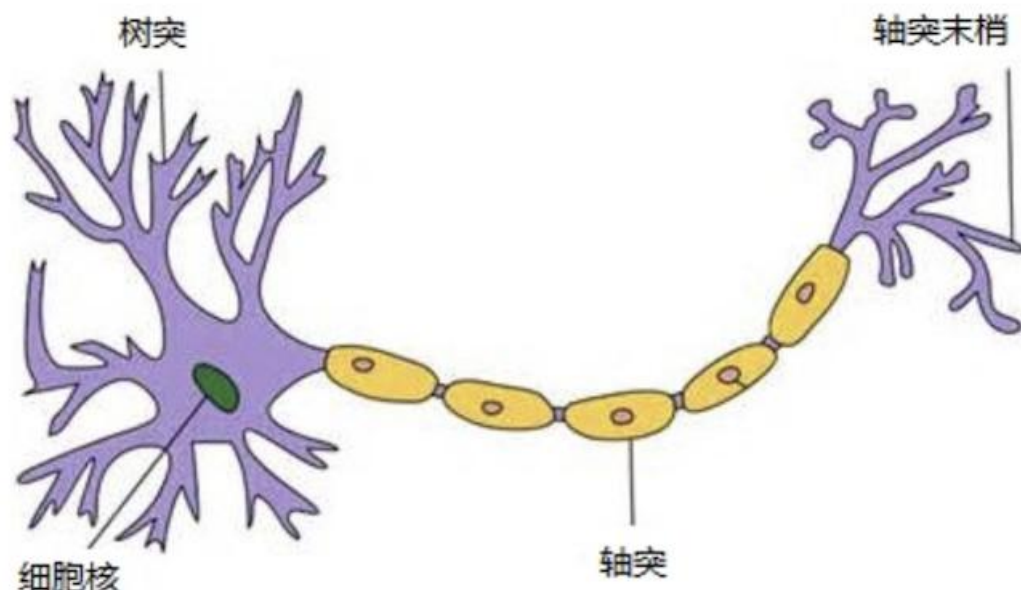
学习目标

Learning Objectives

1. 知道什么是人工神经网络
2. 知道常见的激活函数
3. 了解常见的参数初始化方法
4. 能够完成神经网络模型的搭建
5. 知道神经网络模型参数的计算方法

什么是神经网络

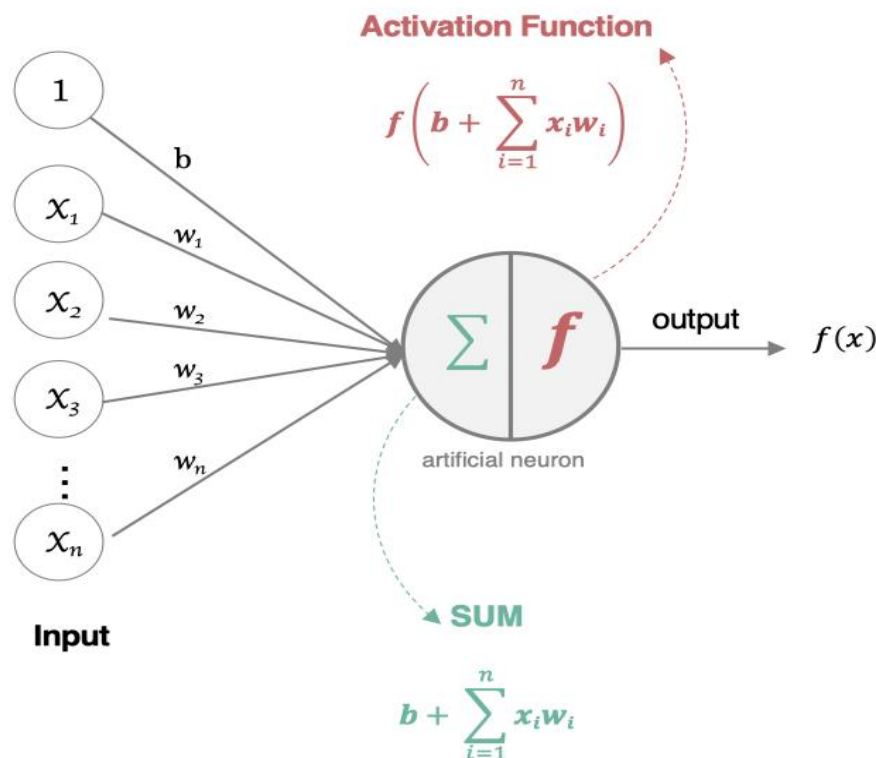
人工神经网络（Artificial Neural Network，简称为ANN）也简称为神经网络（NN），是一种模仿生物神经网络结构和功能的计算模型。人脑可以看做是一个生物神经网络，由众多的**神经元**连接而成。各个神经元传递复杂的电信号，树突接收到**输入信号**，然后对信号进行处理，通过轴突**输出信号**。下图是生物神经元示意图：



当电信号通过树突进入到细胞核时，会逐渐聚集电荷。达到一定的电位后，细胞就会被激活，通过轴突发出电信号。

如何构建神经网络

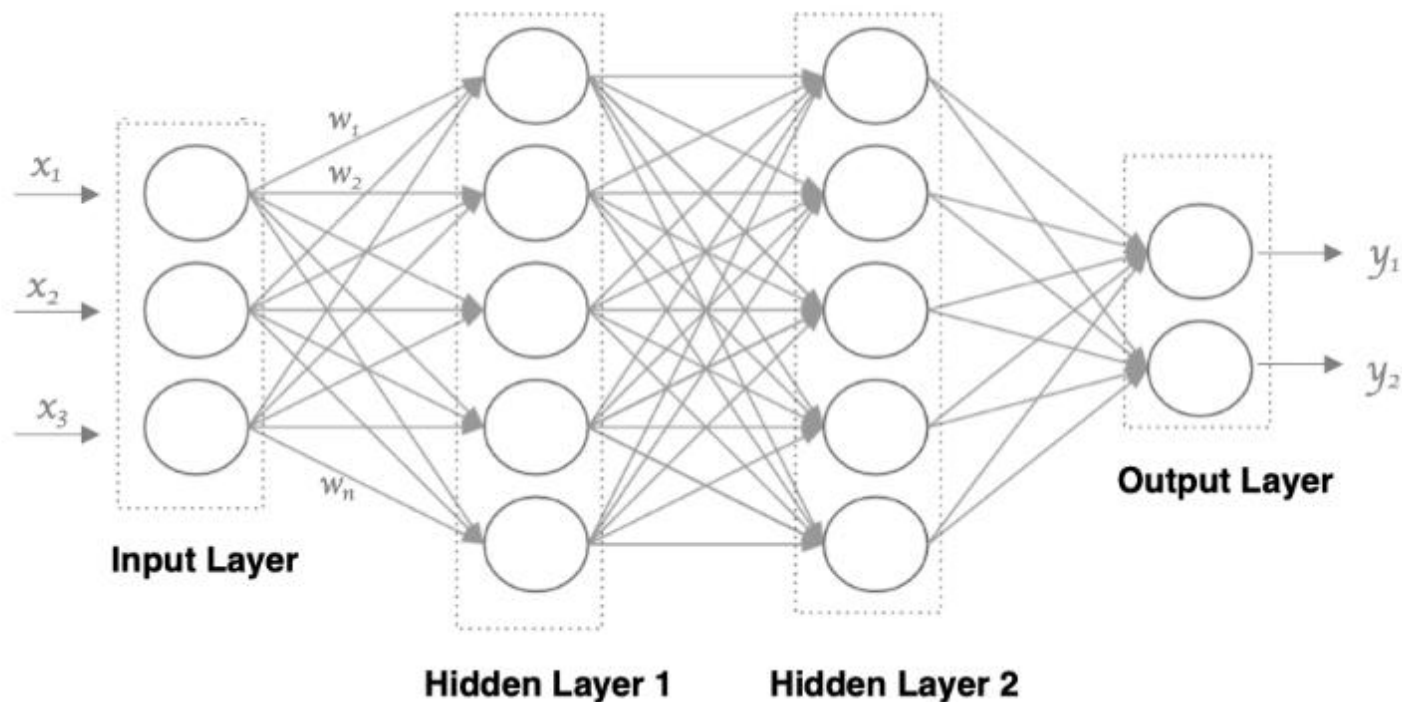
神经网络是由多个神经元组成，构建神经网络就是在构建神经元。以下是神经网络中神经元的构建说明：



这个过程就像，来源不同树突(树突都会有不同的权重)的信息，进行的加权计算，输入到细胞中做加和，再通过激活函数输出细胞值。

如何构建神经网络

接下来，我们使用多个神经元来构建神经网络，相邻层之间的神经元相互连接，并给每一个连接分配一个强度，如下图所示：



如何构建神经网络

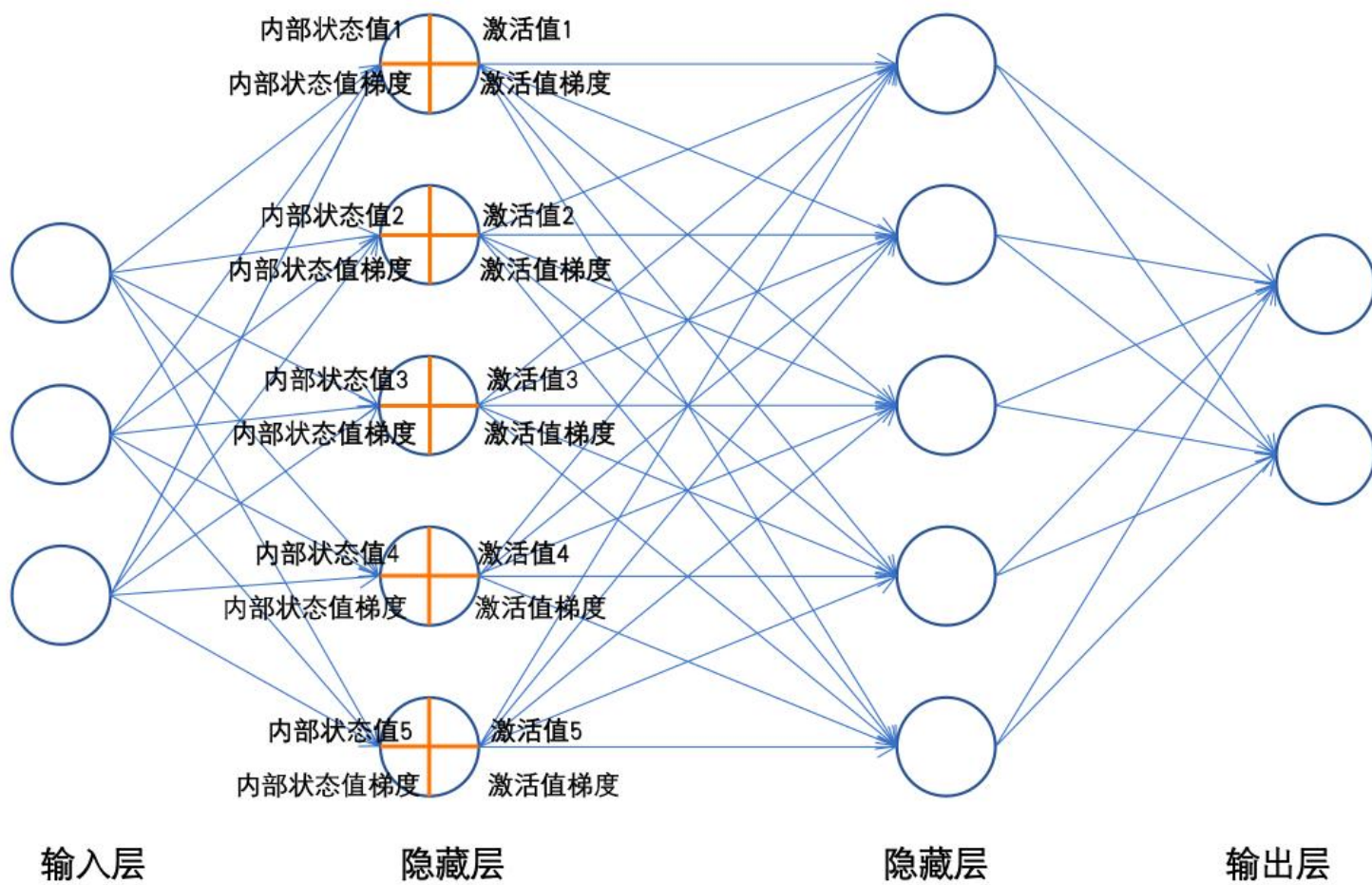
神经网络中信息只向一个方向移动，即从输入节点向前移动，通过隐藏节点，再向输出节点移动。其中的基本部分是：

1. **输入层 (Input Layer)**：即输入 x 的那一层（如图像、文本、声音等）。每个输入特征对应一个神经元。输入层将数据传递给下一层的神经元。
2. **输出层 (Output Layer)**：即输出 y 的那一层。输出层的神经元根据网络的任务（回归、分类等）生成最终的预测结果。
3. **隐藏层 (Hidden Layers)**：输入层和输出层之间都是隐藏层，神经网络的“深度”通常由隐藏层的数量决定。隐藏层的神经元通过加权和激活函数处理输入，并将结果传递到下一层。

特点是：

- 同一层的神经元之间没有连接
- 第 N 层的每个神经元和第 $N-1$ 层 的所有神经元相连（这就是full connected的含义），这就是全连接神经网络
- 全连接神经网络接收的样本数据是二维的，数据在每一层之间需要以二维的形式传递
- 第 $N-1$ 层神经元的输出就是第 N 层神经元的输入
- 每个连接都有一个权重值（ w 系数和 b 系数）

神经网络内部状态值和激活值



神经网络内部状态值和激活值

每一个神经元工作时，前向传播会产生两个值，**内部状态值（加权求和值）**和**激活值**；反向传播时会产生激活值梯度和内部状态值梯度。

内部状态值

神经元或隐藏单元的内部存储值，它反映了当前神经元接收到的输入、历史信息以及网络内部的权重计算结果。

$$z=W \cdot x+b$$

W：权重矩阵

x：输入值

b：偏置

激活值

通过激活函数（如 ReLU、Sigmoid、Tanh）对内部状态值进行非线性变换后得到的结果。激活值决定了当前神经元的输出。

$$a=f(z)$$

f：激活函数

z：内部状态值



总结

1. 知道神经网络是什么

一种模仿生物神经网络结构和功能的计算模型，由神经元构成（加权和+激活函数）

2. 神经网络的构成

输入层，隐藏层和输出层

3. 知道什么是内部状态和激活值

内部状态值：神经元接收到的输入、历史信息以及网络

内部的权重计算结果

激活值：激活函数对内部状态值进行非线性变换后得到

的结果

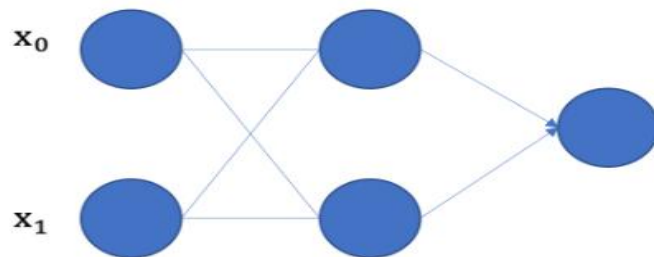
激活函数

激活函数用于对每层的**输出数据进行变换**，进而为整个网络注入了**非线性因素**。此时，神经网络就可以拟合各种曲线。

1. 没有引入非线性因素的网络等价于使用一个线性模型来拟合
2. 通过给网络输出增加激活函数，实现引入非线性因素，使得网络模型可以逼近任意函数，提升网络对复杂问题的拟合能力.

激活函数

如果不使用激活函数，整个网络虽然看起来复杂，其本质还相当于一种**线性模型**，如下公式所示：



$$f_{c1} = w_0^{c1}x_0 + w_1^{c1}x_1 + b^{c1}$$

$$f_{c2} = w_0^{c2}x_0 + w_1^{c2}x_1 + b^{c2}$$

$$f_{out} = w_0^{out}f_{c1} + w_1^{out}f_{c2} + b^{out}$$

$$\begin{aligned} &= w_0^{out}(w_0^{c1}x_0 + w_1^{c1}x_1 + b^{c1}) + w_1^{out}(w_0^{c2}x_0 + w_1^{c2}x_1 + b^{c2}) + b^{out} \\ &= (w_0^{out}w_0^{c1} + w_1^{out}w_0^{c2})x_0 + (w_0^{out}w_1^{c1} + w_1^{out}w_1^{c2})x_1 + (w_0^{out}b^{c1} + w_1^{out}b^{c2} + b^{out}) \end{aligned}$$

$$\begin{aligned} k_0 &= w_0^{out}w_0^{c1} + w_1^{out}w_0^{c2} \\ k_1 &= w_0^{out}w_1^{c1} + w_1^{out}w_1^{c2} \\ c &= w_0^{out}b^{c1} + w_1^{out}b^{c2} + b^{out} \end{aligned}$$

$$= k_0x_0 + k_1x_1 + c$$

[神经网络演示](#)

常见的激活函数-sigmoid 激活函数

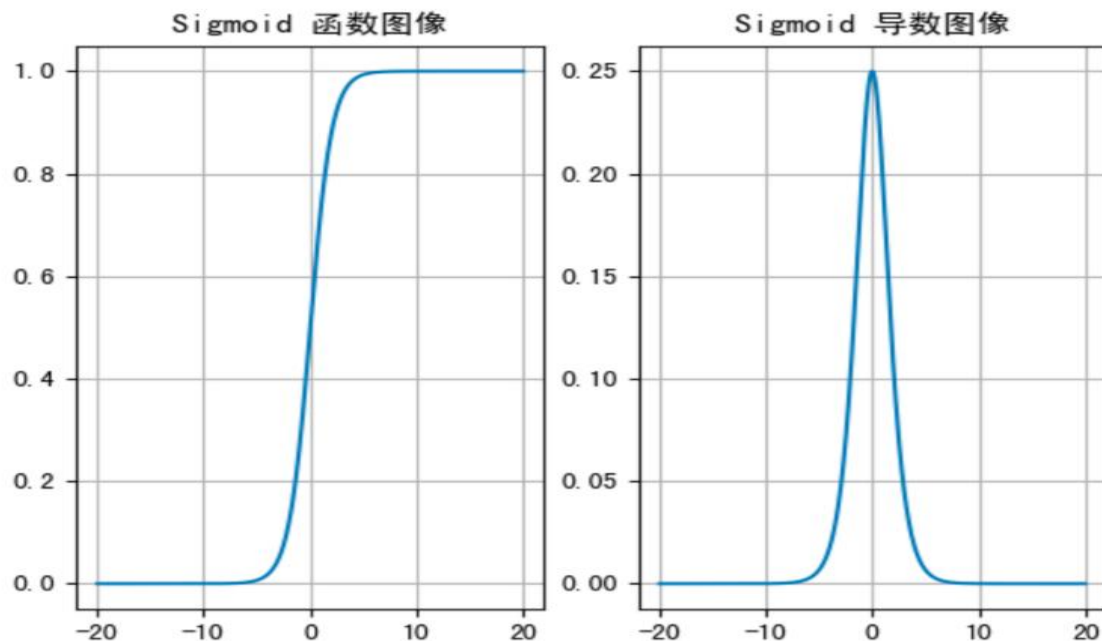
激活函数公式:

$$f(x) = \frac{1}{1 + e^{-x}}$$

激活函数求导公式:

$$f'(x) = \left(\frac{1}{1+e^{-x}}\right)' = \frac{1}{1+e^{-x}} \left(1 - \frac{1}{1+e^{-x}}\right) = f(x)(1 - f(x))$$

sigmoid 激活函数的函数图像如下:



常见的激活函数-sigmoid 激活函数

- sigmoid 函数可以将任意的输入映射到 $(0, 1)$ 之间，当输入的值大致在 <-6 或者 >6 时，意味着输入任何值得到的激活值都是差不多的，这样会丢失部分的信息。比如：输入 100 和输出 10000 经过 sigmoid 的激活值几乎都是等于 1 的，但是输入的数据之间相差 100 倍的信息就丢失了。
- 对于 sigmoid 函数而言，输入值在 $[-6, 6]$ 之间输出值才会有明显差异，输入值在 $[-3, 3]$ 之间才会有比较好的效果。
- 通过上述导数图像，我们发现导数数值范围是 $(0, 0.25)$ ，当输入 <-6 或者 >6 时，sigmoid 激活函数图像的导数接近为 0，此时网络参数将更新极其缓慢，或者无法更新。
- 一般来说，sigmoid 网络在 5 层之内就会产生梯度消失现象。而且，该激活函数并不是以 0 为中心的，所以在实践中这种激活函数使用的很少。sigmoid 函数一般只用于二分类的输出层。

常见的激活函数-sigmoid 激活函数

绘制激活函数图像时出现以下提示，需要将anaconda3/Lib/site-packages/torch/lib目录下的libiomp5md.dll文件删除

OMP: Error #15: Initializing libiomp5md.dll, but found libiomp5md.dll already initialized.

```
import torch
import matplotlib.pyplot as plt

# 创建画布和坐标轴
_, axes = plt.subplots(1, 2)

# 函数图像
x = torch.linspace(-20, 20, 1000)
# 输入值x通过sigmoid函数转换成激活值y
y = torch.sigmoid(x)
axes[0].plot(x, y)
axes[0].grid()
axes[0].set_title('Sigmoid 函数图像')
```

常见的激活函数-sigmoid 激活函数

```
# 导数图像
x = torch.linspace(-20, 20, 1000, requires_grad=True)
torch.sigmoid(x).sum().backward()
# x.detach():输入值x的数值
# x.grad:计算梯度，求导
axes[1].plot(x.detach(), x.grad)
axes[1].grid()
axes[1].set_title('Sigmoid 导数图像')
plt.show()
```

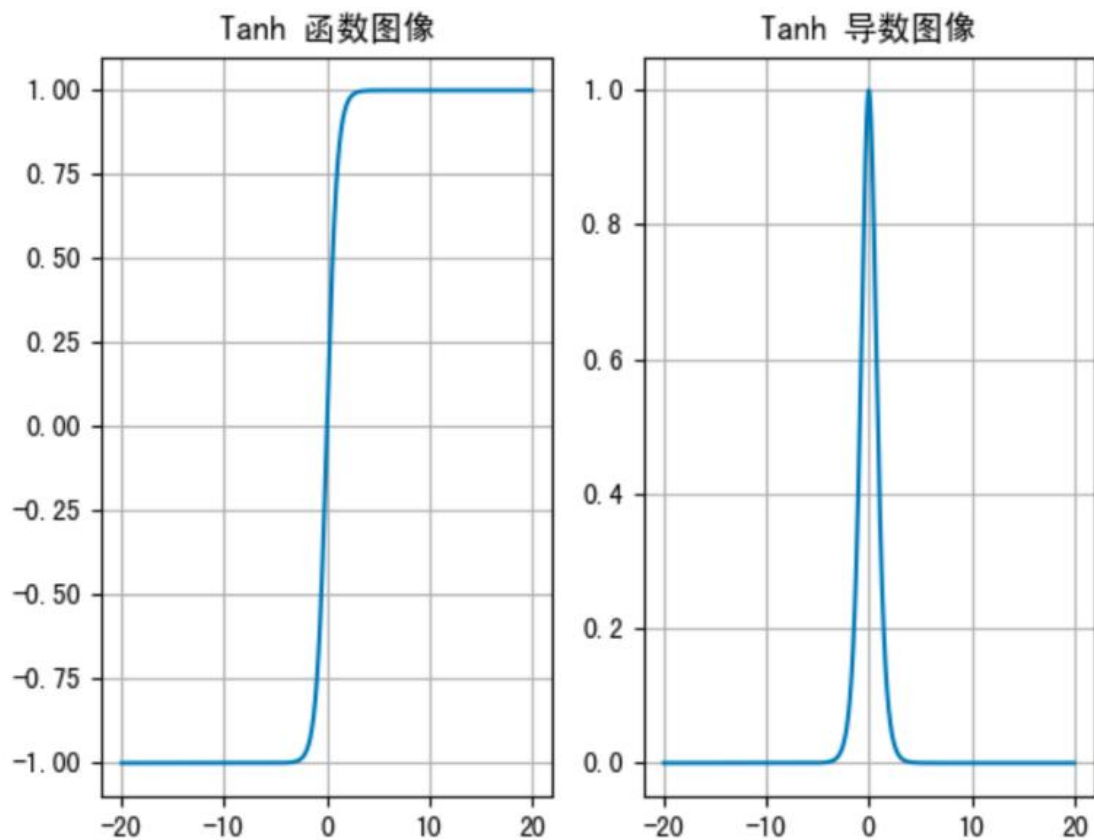

常见的激活函数 -tanh 激活函数

Tanh 的公式如下:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

激活函数求导公式: $f'(x) = \left(\frac{1 - e^{-2x}}{1 + e^{-2x}}\right)' = 1 - f^2(x)$

Tanh 的函数图像、导数图像如下:



常见的激活函数 -tanh 激活函数

- Tanh 函数将输入映射到 $(-1, 1)$ 之间，图像以 0 为中心，在 0 点对称，当输入 大概 <-3 或者 >3 时将被映射为 -1 或者 1。其导数值范围 $(0, 1)$ ，当输入的值大概 <-3 或者 >3 时，其导数近似 0。
- 与 Sigmoid 相比，它是以 0 为中心的，且梯度相对于sigmoid大，使得其收敛速度要比 Sigmoid 快，减少迭代次数。然而，从图中可以看出，Tanh 两侧的导数也为 0，同样会造成梯度消失。
- 若使用时可在隐藏层使用tanh函数，在输出层使用sigmoid函数。

常见的激活函数 -tanh 激活函数

```
# 创建画布和坐标轴
_, axes = plt.subplots(1, 2)
# 函数图像
x = torch.linspace(-20, 20, 1000)
y = torch.tanh(x)
axes[0].plot(x, y)
axes[0].grid()
axes[0].set_title('Tanh 函数图像')
# 导数图像
x = torch.linspace(-20, 20, 1000, requires_grad=True)
torch.tanh(x).sum().backward()
axes[1].plot(x.detach(), x.grad)
axes[1].grid()
axes[1].set_title('Tanh 导数图像')
plt.show()
```

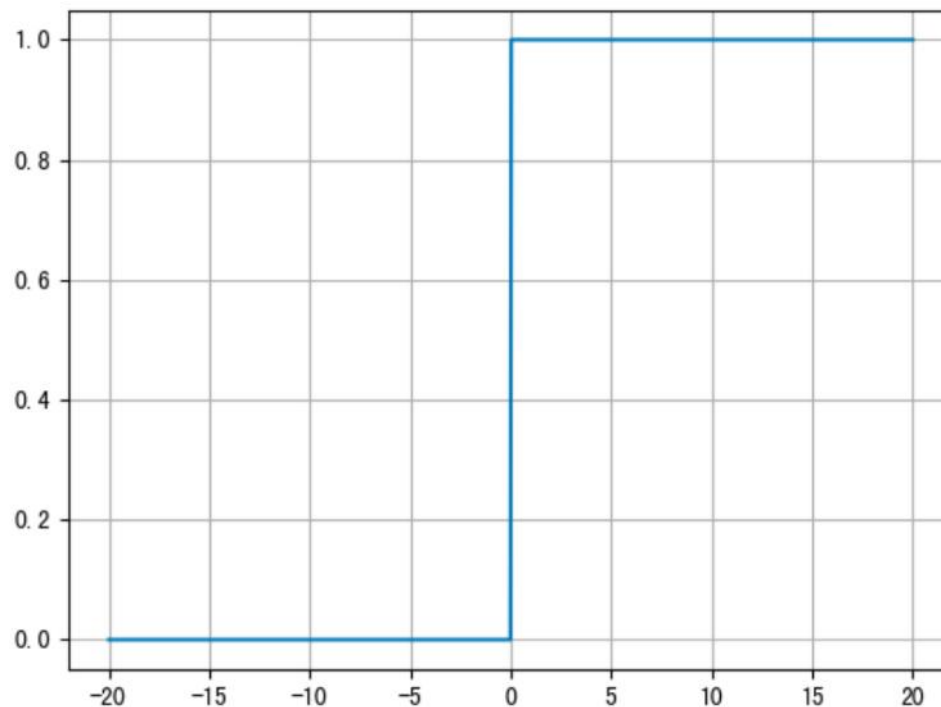
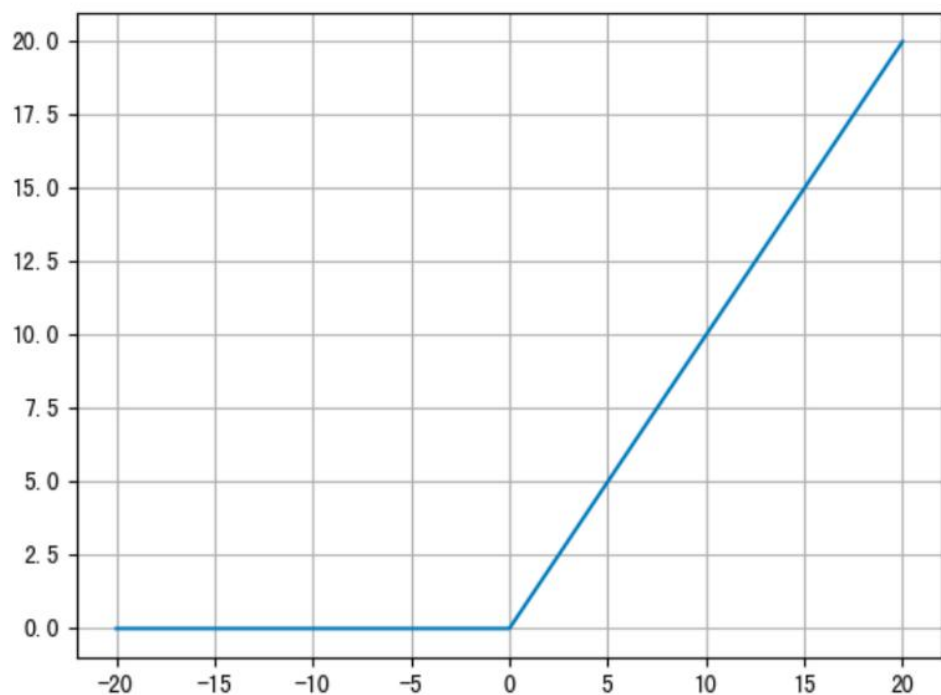
常见的激活函数-ReLU 激活函数

ReLU 公式如下: $f(x) = \max(0, x)$

激活函数求导公式:

$$f'(x) = 0 \text{ 或 } 1$$

ReLU 的函数图像如下:



常见的激活函数-ReLU 激活函数

- ReLU 激活函数将小于 0 的值映射为 0，而大于 0 的值则保持不变，它更加重视正信号，而忽略负信号，这种激活函数运算更为简单，能够提高模型的训练效率。
- 当 $x < 0$ 时，ReLU导数为0，而当 $x > 0$ 时，则不存在饱和问题。所以，ReLU 能够在 $x > 0$ 时保持梯度不衰减，从而缓解梯度消失问题。然而，随着训练的推进，部分输入会落入小于0区域，导致对应权重无法更新。这种现象被称为“神经元死亡”。
- ReLU是目前最常用的激活函数。与sigmoid相比，ReLU的优势是：

采用sigmoid函数，计算量大（指数运算），反向传播求误差梯度时，计算量相对大，而采用Relu激活函数，整个过程的计算量节省很多。 sigmoid函数反向传播时，很容易就会出现梯度消失的情况，从而无法完成深层网络的训练。 Relu会使一部分神经元的输出为0，这样就造成了网络的稀疏性，并且减少了参数的相互依存关系，缓解了过拟合问题的发生。

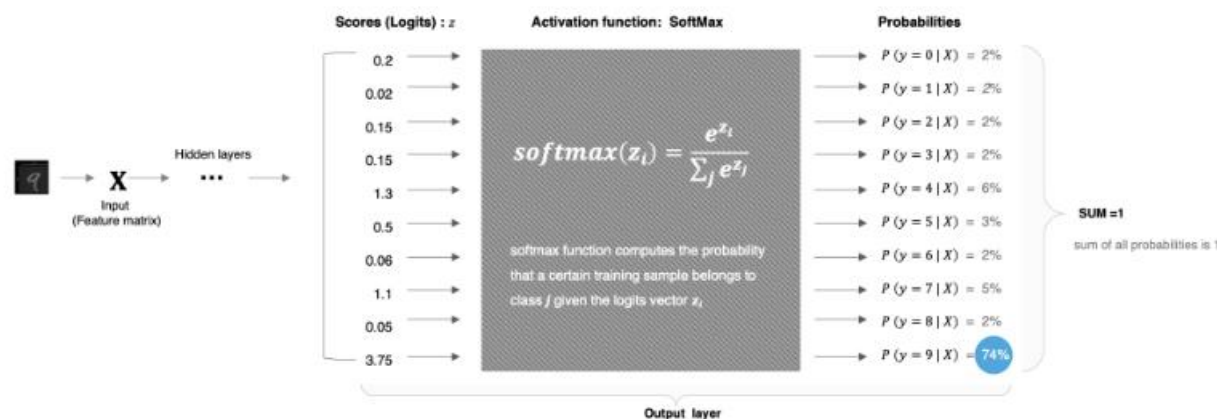
常见的激活函数-ReLU 激活函数

```
# 创建画布和坐标轴
_, axes = plt.subplots(1, 2)
# 函数图像
x = torch.linspace(-20, 20, 1000)
y = torch.relu(x)
axes[0].plot(x, y)
axes[0].grid()
axes[0].set_title('ReLU 函数图像')
# 导数图像
x = torch.linspace(-20, 20, 1000, requires_grad=True)
torch.relu(x).sum().backward()
axes[1].plot(x.detach(), x.grad)
axes[1].grid()
axes[1].set_title('ReLU 导数图像')
plt.show()
```

常见的激活函数-SoftMax 激活函数

softmax用于多分类过程中，它是二分类函数sigmoid在多分类上的推广，目的是将多分类的结果以概率的形式展现出来。计算方法如下图所示：

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$



常见的激活函数-SoftMax 激活函数

Softmax 就是将网络输出的 logits 通过 softmax 函数，就映射成为(0, 1)的值，而这些值的累和为1（满足概率的性质），那么我们将它理解成概率，选取概率最大（也就是值对应最大的）节点，作为我们的预测目标类别。


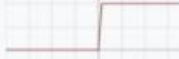







```
import torch

scores = torch.tensor([0.2, 0.02, 0.15, 0.15, 1.3, 0.5, 0.06, 1.1, 0.05, 3.75])
# dim = 0, 按行计算
probabilities = torch.softmax(scores, dim=0)
print(probabilities)
```

输出结果:

```
tensor([0.0212, 0.0177, 0.0202, 0.0202, 0.0638, 0.0287, 0.0185, 0.0522,
0.0183, 0.7392])
```


其他常见的激活函数

Name	Plot	Equation	Derivative (with respect to x)	Range
Identity function		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$	$\{0, 1\}$
Logistic		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified linear unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$[0, \infty)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Parameteric rectified linear unit (PReLU)		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Randomized leaky rectified linear unit (RRReLU)		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\infty, \infty)$
Exponential linear unit (ELU)		$f(\alpha, x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(\alpha, x) = \begin{cases} f(\alpha, x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$(-\alpha, \infty)$

激活函数的选择方法

对于隐藏层：

1. 优先选择ReLU激活函数
2. 如果ReLU效果不好，那么尝试其他激活，如Leaky ReLU等。
3. 如果你使用了ReLU， 需要注意一下Dead ReLU问题， 避免出现0梯度从而导致过多的神经元死亡。
4. 少用使用sigmoid激活函数，可以尝试使用tanh激活函数

对于输出层：

1. 二分类问题选择sigmoid激活函数
2. 多分类问题选择softmax激活函数
3. 回归问题选择identity激活函数



总结

1. 激活函数的作用?

向神经网络中添加非线性元素

2. 常见的激活函数及其特点?

Sigmoid, tanh, relu, softmax等

3. 激活函数的选择方法?

隐藏层: relu

输出层: 二分类: sigmoid, 多分类: softmax, 回归: 恒等

参数初始化

- 均匀分布初始化
 - 权重参数初始化从区间均匀随机取值，默认区间为 $(0, 1)$ 。可以设置为在 $(-1/\sqrt{d}, 1/\sqrt{d})$ 均匀分布中生成当前神经元的权重，其中 d 为神经元的输入数量
- 正态分布初始化
 - 随机初始化从均值为0，标准差是1的高斯分布中取样，使用一些很小的值对参数 W 进行初始化
- 全0初始化
 - 将神经网络中的所有权重参数初始化为 0
- 全1初始化
 - 将神经网络中的所有权重参数初始化为 1
- 固定值初始化
 - 将神经网络中的所有权重参数初始化为某个固定值

参数初始化

- kaiming 初始化，也叫做 **HE 初始化**
 - HE 初始化分为正态分布的 HE 初始化、均匀分布的 HE 初始化.
 - 正态分布的he初始化
 - ◆ 它是从 $[0, \text{std}]$ 中抽取样本的, $\text{std} = \sqrt{2 / \text{fan_in}}$
 - 均匀分布的he初始化
 - ◆ 它从 $[-\text{limit}, \text{limit}]$ 中的均匀分布中抽取样本, limit 是 $\sqrt{6 / \text{fan_in}}$
 - fan_in 输入层神经元的个数
 - xavier 初始化，也叫做 **Glorot初始化**
 - 该方法也有两种，一种是正态分布的 xavier 初始化、一种是均匀分布的 xavier 初始化.
 - 正态化的Xavier初始化
 - ◆ 它是从 $[0, \text{std}]$ 中抽取样本的, $\text{std} = \sqrt{2 / (\text{fan_in} + \text{fan_out})}$
 - 均匀分布的Xavier初始化
 - ◆ $[-\text{limit}, \text{limit}]$ 中的均匀分布中抽取样本, limit 是 $\sqrt{6 / (\text{fan_in} + \text{fan_out})}$
 - fan_in 是输入层神经元的个数, fan_out 是输出层神经元个数

参数初始化

```
import torch
import torch.nn.functional as F
import torch.nn as nn
# 1. 均匀分布随机初始化
def test01():
    linear = nn.Linear(5, 3)
    # 从0-1均匀分布产生参数
    nn.init.uniform_(linear.weight)
    print(linear.weight.data)
# 2. 固定初始化
def test02():
    linear = nn.Linear(5, 3)
    nn.init.constant_(linear.weight, 5)
    print(linear.weight.data)
```

参数初始化

```
# 3. 全0初始化
def test03():
    linear = nn.Linear(5, 3)
    nn.init.zeros_(linear.weight)
    print(linear.weight.data)

# 4. 全1初始化
def test04():
    linear = nn.Linear(5, 3)
    nn.init.ones_(linear.weight)
    print(linear.weight.data)

# 5. 正态分布随机初始化
def test05():
    linear = nn.Linear(5, 3)
    nn.init.normal_(linear.weight, mean=0, std=1)
    print(linear.weight.data)
```

参数初始化

```
# 6. kaiming 初始化
def test06():
    # kaiming 正态分布初始化
    linear = nn.Linear(5, 3)
    nn.init.kaiming_normal_(linear.weight)
    print(linear.weight.data)

    # kaiming 均匀分布初始化
    linear = nn.Linear(5, 3)
    nn.init.kaiming_uniform_(linear.weight)
    print(linear.weight.data)
```


参数初始化

```
# 7. xavier 初始化
def test07():
    # xavier 正态分布初始化
    linear = nn.Linear(5, 3)
    nn.init.xavier_normal_(linear.weight)
    print(linear.weight.data)

    # xavier 均匀分布初始化
    linear = nn.Linear(5, 3)
    nn.init.xavier_uniform_(linear.weight)
    print(linear.weight.data)
```

参数初始化选择

激活函数的选择：根据激活函数的类型选择对应的初始化方法

Sigmoid/Tanh: Xavier 初始化

ReLU/Leaky ReLU: kaiming 初始化

网络的深度

浅层网络：随机初始化即可

深层网络：需要考虑方差平衡，如 Xavier 或 kaiming 初始化



总结

1. 常见的初始化方式

均匀分布初始化，正态分布初始化，全0全1初始化，固定值初始化，kaiming的初始化，xavier初始化

2. 初始化方法的选择

一般我们在使用 PyTorch 构建网络模型时，每个网络层的参数都有默认的初始化方法，优先选择kaiming的初始化，xavier初始化方式。

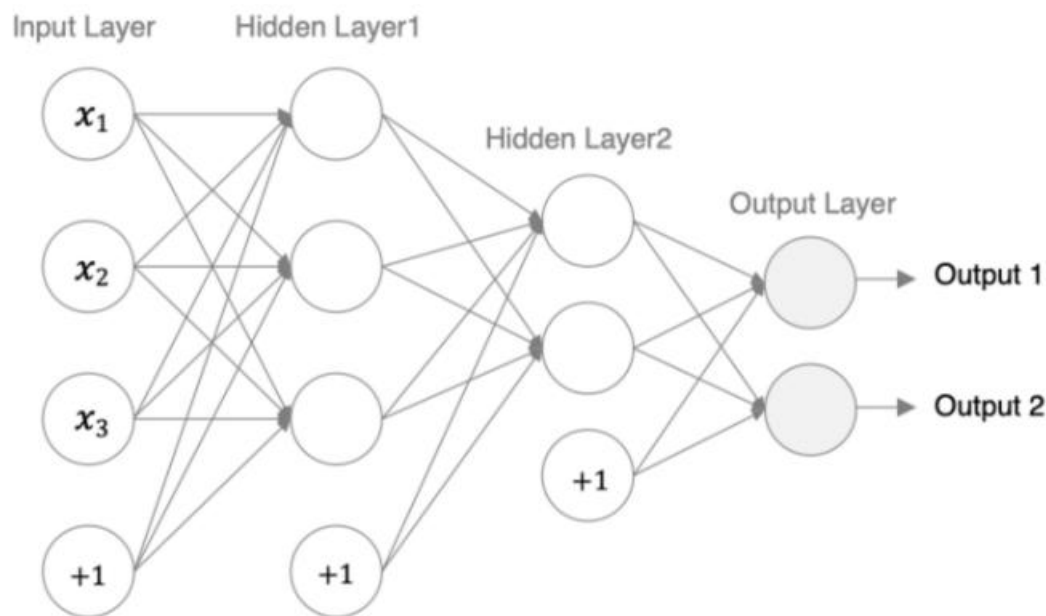
神经网络搭建和参数计算

在pytorch中定义深度神经网络其实就是层堆叠的过程，继承自nn.Module，实现两个方法：

- `__init__`方法中定义网络中的层结构，主要是全连接层，并进行初始化
- `forward`方法，在实例化模型的时候，底层会自动调用该函数。该函数中为初始化定义的layer传入数据，进行前向传播等。

神经网络搭建和参数计算

我们来构建如下图所示的神经网络模型：



编码设计如下：

1. 第1个隐藏层：权重初始化采用标准化的xavier初始化 激活函数使用sigmoid
2. 第2个隐藏层：权重初始化采用标准化的He初始化 激活函数采用relu
3. out输出层线性层 假若多分类，采用softmax做数据归一化

神经网络搭建和参数计算

```
import torch
import torch.nn as nn
from torchsummary import summary # 计算模型参数, 查看模型结构, pip install torchsummary -i
https://mirrors.aliyun.com/pypi/simple/
# 创建神经网络模型类
class Model(nn.Module):
    # 初始化属性值
    def __init__(self):
        super(Model, self).__init__() # 调用父类的初始化属性值
        self.linear1 = nn.Linear(3, 3) # 创建第一个隐藏层模型, 3个输入特征, 3个输出特征
        nn.init.xavier_normal_(self.linear1.weight) # 初始化权重
        nn.init.zeros_(self.linear1.bias)
        # 创建第二个隐藏层模型, 3个输入特征(上一层的输出特征), 2个输出特征
        self.linear2 = nn.Linear(3, 2)
        # 初始化权重
        nn.init.kaiming_normal_(self.linear2.weight, nonlinearity='relu')
        nn.init.zeros_(self.linear2.bias)
        # 创建输出层模型
```

神经网络搭建和参数计算

```
# 创建前向传播方法, 自动执行forward() 方法
def forward(self, x):
    # 数据经过第一个线性层
    x = self.linear1(x)
    # 使用sigmoid激活函数
    x = torch.sigmoid(x)
    # 数据经过第二个线性层
    x = self.linear2(x)
    # 使用relu激活函数
    x = torch.relu(x)
    # 数据经过输出层
    x = self.out(x)
    # 使用softmax激活函数
    # dim=-1:每一维度行数据相加为1
    x = torch.softmax(x, dim=-1)
    return x
```

神经网络搭建和参数计算

```
if __name__ == "__main__":  
    # 实例化model对象  
    my_model = Model()  
    # 随机产生数据  
    my_data = torch.randn(5, 3)  
    print("mydata shape", my_data.shape)  
    # 数据经过神经网络模型训练  
    output = my_model(my_data)  
    print("output shape-->", output.shape)  
    # 计算模型参数  
    # 计算每层每个神经元的w和b个数总和  
    summary(my_model, input_size=(3,), batch_size=5)  
    # 查看模型参数  
    print("=====查看模型参数w和b=====")  
    for name, parameter in my_model.named_parameters():  
        print(name, parameter)
```


神经网络搭建和参数计算

- 神经网络的输入数据是为[batch_size, in_features]的张量经过网络处理后获取了[batch_size, out_features]的输出张量。
- 在上述例子中，batchsize=5, infeatures=3, out_features=2, 结果如下所示：

```
mydata.shape---> torch.Size([5, 3])
```

```
output.shape---> torch.Size([5, 2])
```

神经网络搭建和参数计算

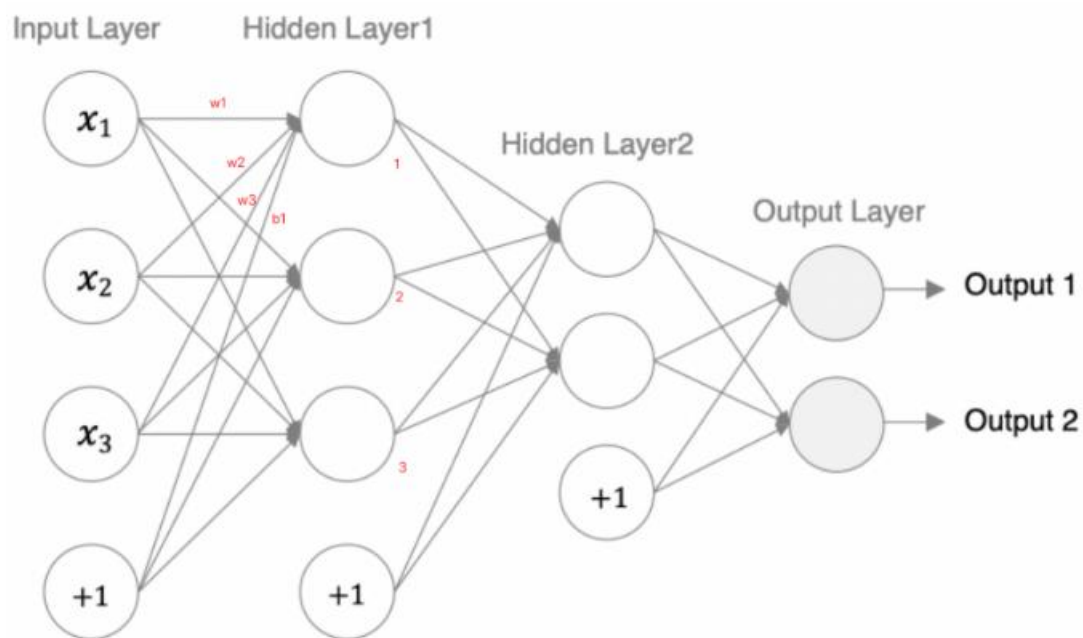
输出结果:

Layer (type)	Output Shape	Param #
Linear-1	[5, 3]	12
Linear-2	[5, 2]	8
Linear-3	[5, 2]	6
Total params: 26		
Trainable params: 26		
Non-trainable params: 0		
Input size (MB): 0.00		
Forward/backward pass size (MB): 0.00		
Params size (MB): 0.00		
Estimated Total Size (MB): 0.00		

神经网络搭建和参数计算

模型参数的计算：

1. 以第一个隐层为例：该隐层有3个神经元，每个神经元的参数为：4个 (w_1, w_2, w_3, b_1)，所以一共用 $3 \times 4 = 12$ 个参数。
2. 输入数据和网络权重是两个不同的事儿！对于初学者理解这一点十分重要，要分得清。





总结

1. 神经网络的搭建方法

- 定义继承自`nn.Module`的模型类
- 在`__init__`方法中定义网络中的层结构
- 在`forward`方法中定义数据传输方式

2. 网络参数量的统计方法

- 统计每一层中的权重`w`和偏置`b`的数量

神经网络的优缺点

1. 优点

- 精度高，性能优于其他的机器学习方法，甚至在某些领域超过了人类
- 可以近似任意的非线性函数
- 近年来在学界和业界受到了热捧，有大量的框架和库可供调。

2. 缺点

- 黑箱，很难解释模型是怎么工作的
- 训练时间长，需要大量的计算力
- 网络结构复杂，需要调整超参数
- 小数据集上表现不佳，容易发生过拟合



目录

Contents



- ◆ 神经网络
- ◆ 损失函数
- ◆ 网络优化方法
- ◆ 正则化方法
- ◆ 案例-价格分类案例

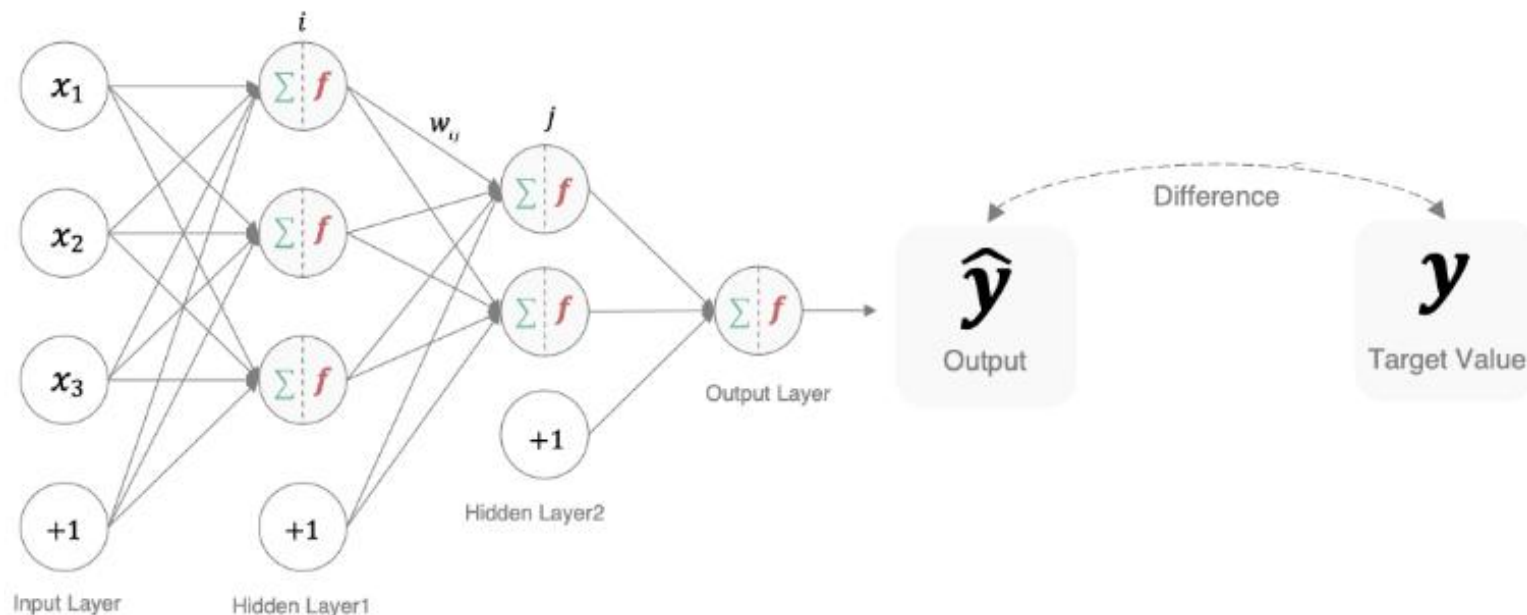
学习目标

Learning Objectives

1. 知道分类任务的损失函数
2. 知道回归任务的损失函数

什么是损失函数

在深度学习中，损失函数是用来衡量模型参数的质量的函数，衡量的方式是比较网络输出和真实输出的差异：



什么是损失函数

损失函数在不同的文献中名称是不一样的，主要有以下几种命名方式：



多分类任务损失函数

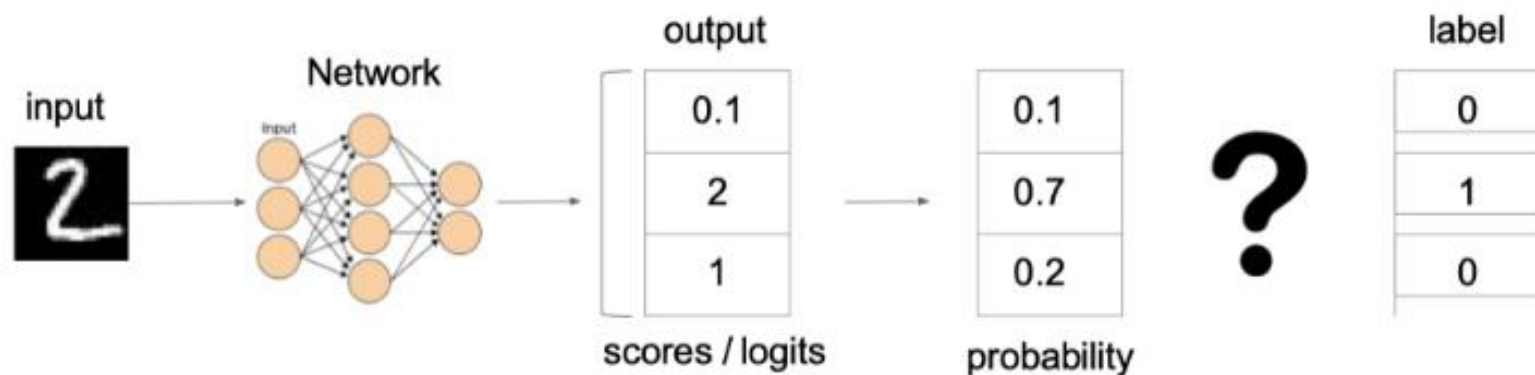
在多分类任务通常使用softmax将logits转换为概率的形式，所以多分类的交叉熵损失也叫做**softmax损失**，它的计算方法是：

$$\mathcal{L} = - \sum_{i=1}^n \overset{\text{labels (one-hot)}}{y_i} \log(\overset{\text{Softmax}}{S(f_{\theta}(\mathbf{x}_i)))}$$

其中：

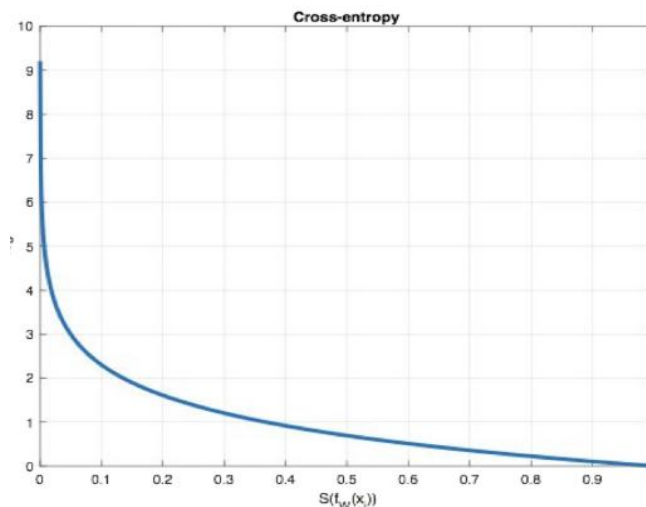
1. y 是样本 x 属于某一个类别的真实概率
2. 而 $f(x)$ 是样本属于某一类别的预测分数
3. S 是softmax激活函数,将属于某一类别的预测分数转换成概率
4. L 用来衡量真实值 y 和预测值 $f(x)$ 之间差异性的损失结果

多分类任务损失函数



上图中的交叉熵损失为: $-(0\log(0.10) + 1\log(0.7) + 0\log(0.2)) = -\log 0.7$

从概率角度理解，我们的目的是最小化正确类别所对应的预测概率的对数的负值(损失值最小)，如下图所示：



多分类任务损失函数

在pytorch中使用nn.CrossEntropyLoss()实现，如下所示：

```
# 分类损失函数：交叉熵损失使用nn.CrossEntropyLoss()实现。nn.CrossEntropyLoss()=softmax + 损失计算
def test01():
    # 设置真实值：可以是热编码后的结果也可以不进行热编码
    # y_true = torch.tensor([[0, 1, 0], [0, 0, 1]], dtype=torch.float32)
    # 注意的类型必须是64位整型数据
    y_true = torch.tensor([1, 2], dtype=torch.int64)
    y_pred = torch.tensor([[0.2, 0.6, 0.2], [0.1, 0.8, 0.1]], requires_grad=True,
dtype=torch.float32)
    # 实例化交叉熵损失
    loss = nn.CrossEntropyLoss()
    # 计算损失结果
    my_loss = loss(y_pred, y_true).detach().numpy()
    print('loss:', my_loss)
```

二分类任务损失函数

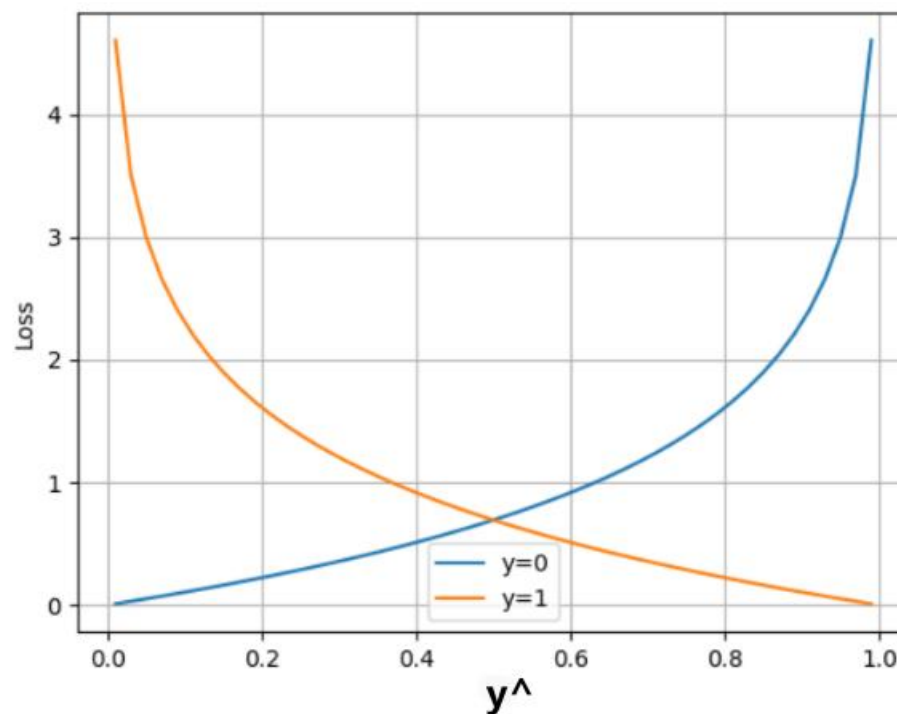
在处理二分类任务时，我们不再使用softmax激活函数，而是使用sigmoid激活函数，那损失函数也相应的进行调整，使用二分类的交叉熵损失函数：

$$L = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

其中：

1. y 是样本 x 属于某一个类别的真实概率
2. 而 \hat{y} 是样本属于某一类别的预测概率
3. L 用来衡量真实值 y 与预测值 \hat{y} 之间差异性的损失结果。

在pytorch中实现时使用`nn.BCELoss()`，如下所示：



二分类任务损失函数

```
def test02():  
    # 1 设置真实值和预测值  
    # 预测值是sigmoid输出的结果  
    y_pred = torch.tensor([0.6901, 0.5459, 0.2469],  
requires_grad=True)  
    y_true = torch.tensor([0, 1, 0], dtype=torch.float32)  
    # 2 实例化二分类交叉熵损失  
    loss = nn.BCELoss()  
    # 3 计算损失  
    my_loss = loss(y_pred, y_true).detach().numpy()  
    print('loss: ', my_loss)
```

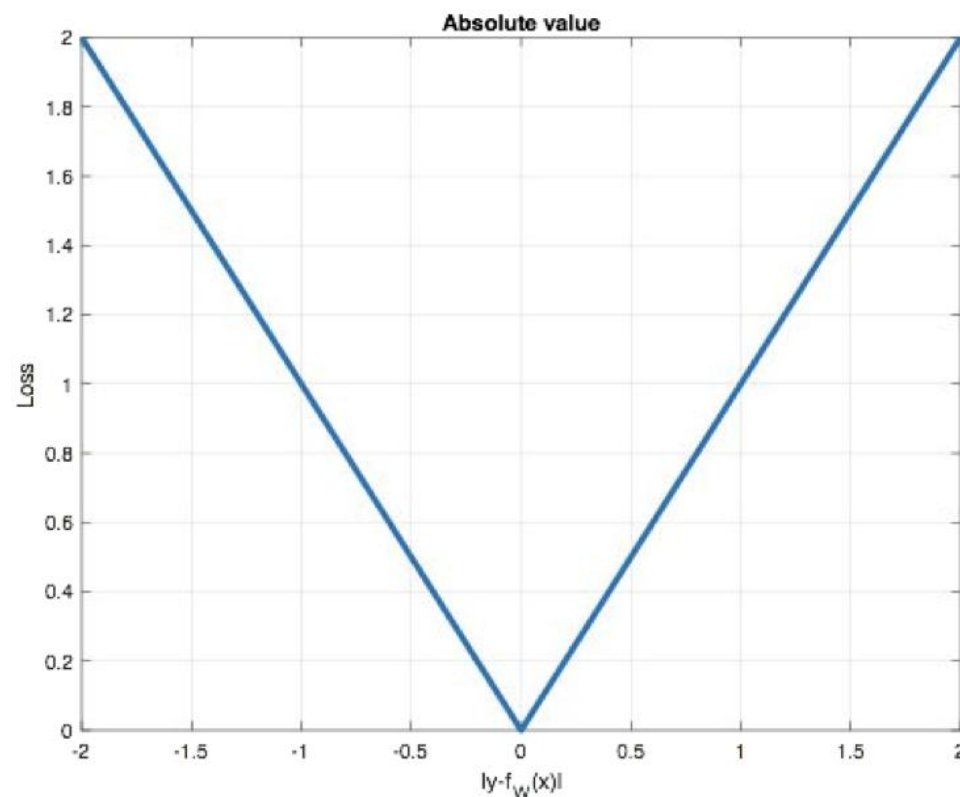
回归任务损失函数-MAE损失函数

Mean absolute loss (MAE) 也被称为L1 Loss，是以绝对误差作为距离。损失函数公式：

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n |y_i - f_{\theta}(x_i)|$$

特点是：

1. 由于L1 loss具有稀疏性，为了惩罚较大的值，因此常常将其作为正则项添加到其他loss中作为约束。
2. L1 loss的最大问题是梯度在零点不平滑，导致会跳过极小值。



回归任务损失函数- MAE损失函数

在pytorch中使用nn.L1Loss()实现，如下所示：

```
# 计算inputs与target之差的绝对值
def test03():
    # 1 设置真实值和预测值
    y_pred = torch.tensor([1.0, 1.0, 1.9],
requires_grad=True)
    y_true = torch.tensor([2.0, 2.0, 2.0],
dtype=torch.float32)
    # 2 实例MAE损失对象
    loss = nn.L1Loss()
    # 3 计算损失
    my_loss = loss(y_pred, y_true).detach().numpy()
    print('loss:', my_loss)
```


回归任务损失函数-MSE损失函数

Mean Squared Loss/ Quadratic Loss (MSE loss) 也被称为L2 loss，或欧氏距离，它以误差的平方和的均值作为距离

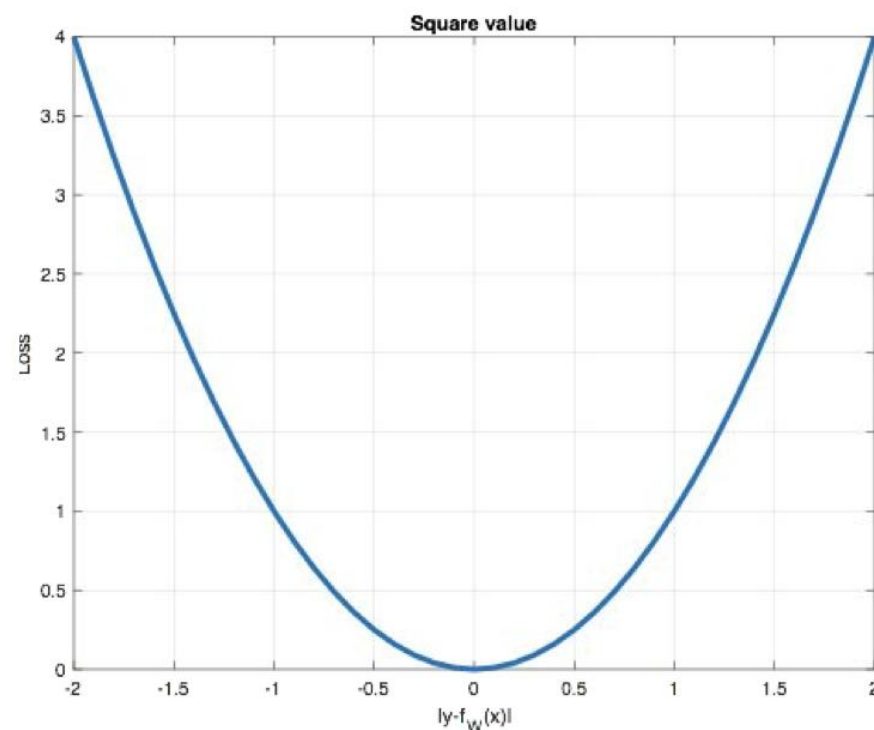
损失函数公式：

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^n (y_i - f_{\theta}(x_i))^2$$

曲线如下图所示：

特点是：

1. L2 loss也常常作为正则项。
2. 当预测值与目标值相差很大时，梯度容易爆炸。



回归任务损失函数-MSE损失函数

在`pytorch`中使用`nn.MSELoss()`实现，如下所示：

```
def test04():  
    # 1 设置真实值和预测值  
    y_pred = torch.tensor([1.0, 1.0, 1.9], requires_grad=True)  
    y_true = torch.tensor([2.0, 2.0, 2.0], dtype=torch.float32)  
    # 2 实例MSE损失对象  
    loss = nn.MSELoss()  
    # 3 计算损失  
    my_loss = loss(y_pred, y_true).detach().numpy()  
    print('myloss:', my_loss)
```

回归任务损失函数-Smooth L1损失函数

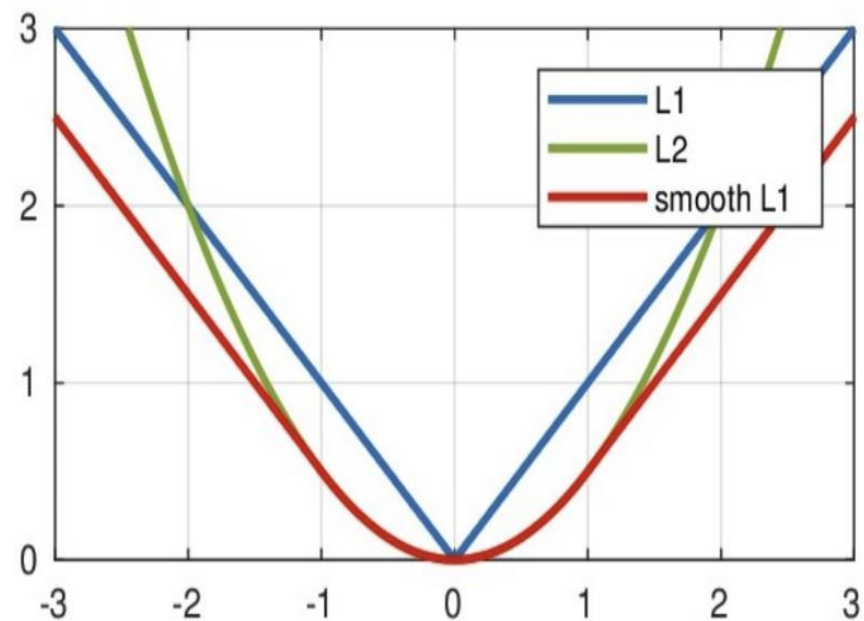
Smooth L1说的是光滑之后的L1。损失函数公式：

$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise} \end{cases}$$

其中： $x = f(x) - y$ 为真实值和预测值的差值。

从右图中可以看出，该函数实际上就是一个分段函数

1. 在 $[-1, 1]$ 之间实际上就是L2损失，这样解决了L1的不光滑问题
2. 在 $[-1, 1]$ 区间外，实际上就是L1损失，这样就解决了离群点梯度爆炸的问题



回归任务损失函数-Smooth L1损失函数

在pytorch中使用nn.SmoothL1Loss()实现，如下所示：

```
def test05():  
    # 1 设置真实值和预测值  
    y_true = torch.tensor([0, 3])  
    y_pred = torch.tensor([0.6, 0.4], requires_grad=True)  
    # 2 实例smoothL1损失对象  
    loss = nn.SmoothL1Loss()  
    # 3 计算损失  
    my_loss = loss(y_pred, y_true).detach().numpy()  
    print('loss:', my_loss)
```



总结

1. 分类任务的损失函数
 - 多分类的交叉熵损失函数
 - 二分类的交叉熵损失函数
2. 回归任务的损失函数
 - MAE, MSE, smooth L1损失函数



目录

Contents



- ◆ 神经网络
- ◆ 损失函数
- ◆ 网络优化方法
- ◆ 正则化方法
- ◆ 案例-价格分类案例

学习目标

Learning Objectives

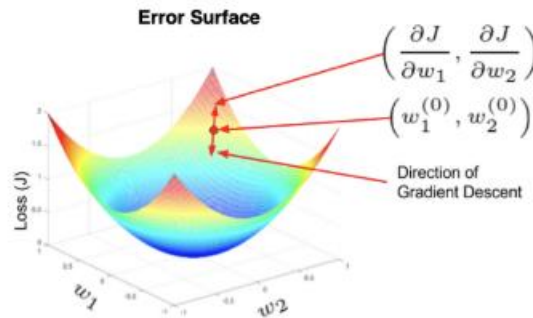
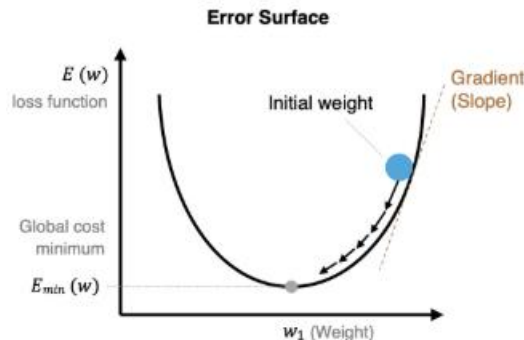
1. 知道梯度下降算法
2. 知道网络训练过程中的epoch, batch, iter
3. 了解反向传播算法过程
4. 知道梯度下降的优化方法
5. 知道学习率优化策略

梯度下降算法回顾

梯度下降法是一种寻找使损失函数最小化的方法。从数学角度来看，梯度的方向是函数增长速度最快的方向，那么梯度的反方向就是函数减少最快的方向，所以有：

$$w_{ij}^{new} = w_{ij}^{old} - \eta \frac{\partial E}{\partial w_{ij}}$$

其中， η 是学习率，如果学习率太小，那么每次训练之后得到的效果都太小，增大训练的时间成本。如果，学习率太大，那就有可能直接跳过最优解，进入无限的训练中。解决的方法就是，学习率也需要随着训练的进行而变化。



梯度下降算法回顾

在进行模型训练时，有三个基础的概念：

1. **Epoch**: 使用全部数据对模型进行以此完整训练，训练轮次
2. **Batch_size**: 使用训练集中的小部分样本对模型权重进行以此反向传播的参数更新，每次训练每批次样本数量
3. **Iteration**: 使用一个 Batch 数据对模型进行一次参数更新的过程

假设数据集有 50000 个训练样本，现在选择 Batch Size = 256 对模型进行训练。

每个 Epoch 要训练的图片数量：50000

训练集具有的 Batch 个数： $50000/256+1=196$

每个 Epoch 具有的 Iteration 个数：196

10个 Epoch 具有的 Iteration 个数：1960

梯度下降算法回顾

在深度学习中，梯度下降的几种方式的根本区别就在于 Batch Size不同,如下表所示:

梯度下降方式	Training Set Size	Batch Size	Number of Batches
BGD	N	N	1
SGD	N	1	N
Mini-Batch	N	B	$N/B + 1$

注：上表中 Mini-Batch 的 Batch 个数为 $N / B + 1$ 是针对未整除的情况。整除则是 N / B 。



总结

1、梯度下降算法的思想

网络优化，使损失函数最小的方法

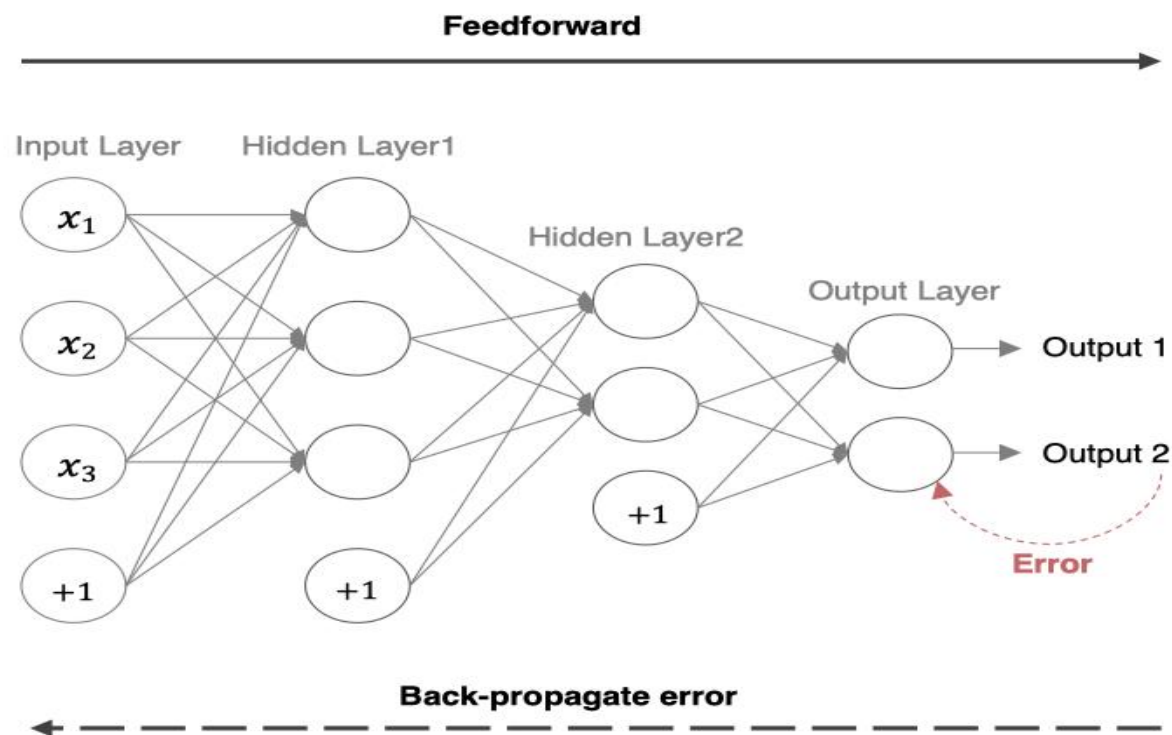
2、网络训练中的三个概念

Epoch, batch, iter

反向传播(BP算法)[了解]

前向传播：指的是数据输入到神经网络中，逐层向前传输，一直运算到输出层为止。

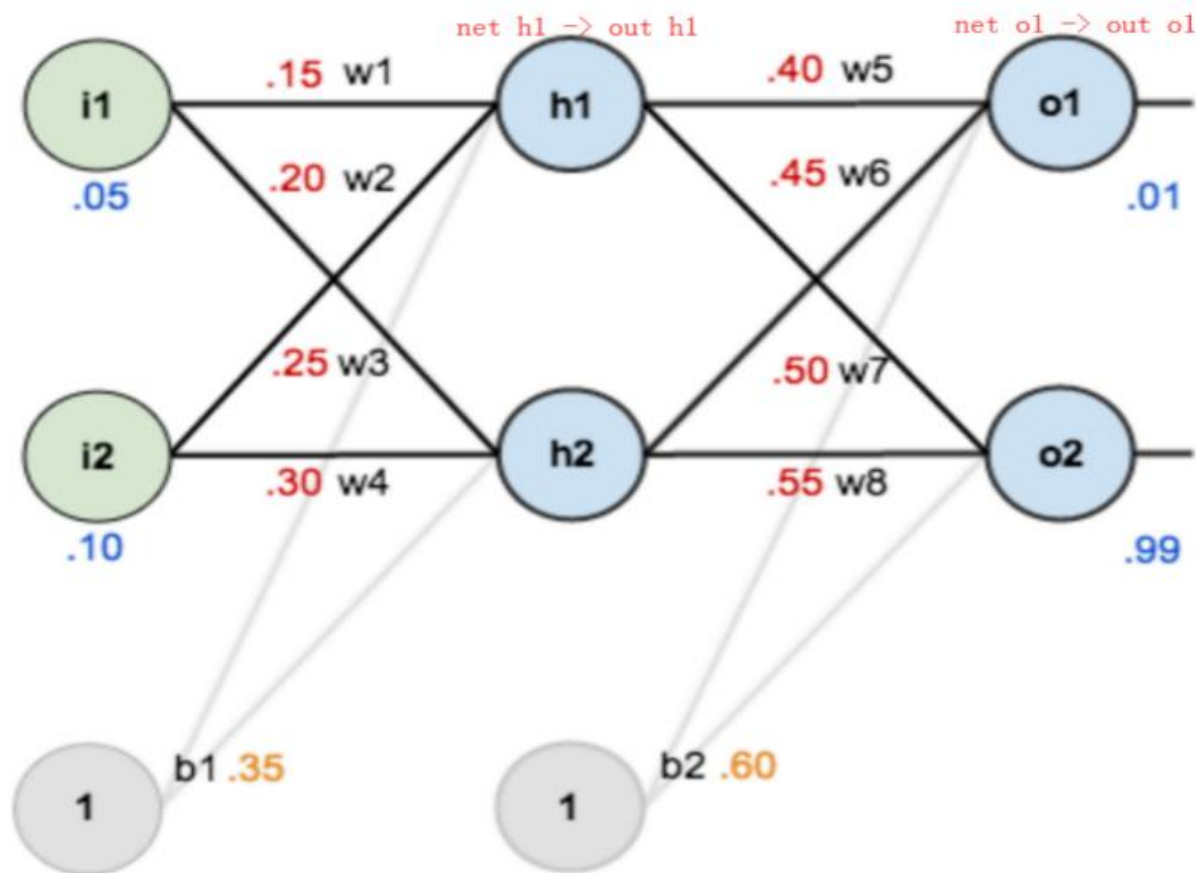
反向传播（Back Propagation）：利用损失函数 ERROR值，从后往前，结合梯度下降算法，依次求各个参数的偏导，并进行参数更新



反向传播(BP算法)[了解]

反向传播算法利用链式法则对神经网络中的各个节点的权重进行更新。

如下图是一个简单的神经网络用来举例：激活函数为sigmoid



反向传播(BP算法)[了解]

前向传播运算过程:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

$$out_{h2} = 0.596884378$$



$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

$$out_{o2} = 0.772928465$$

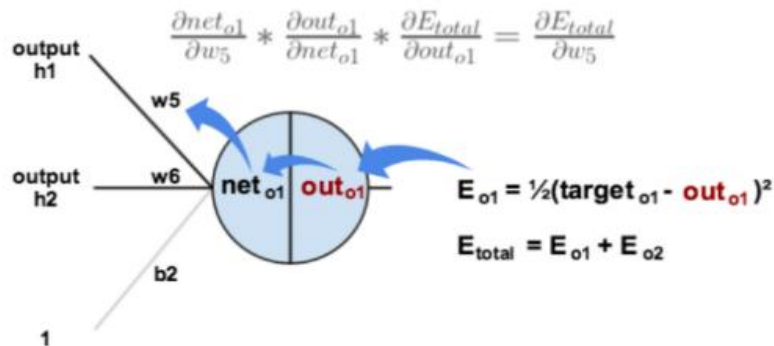


$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

反向传播(BP算法)[了解]

接下来是反向传播，我们先来求最简单的，求误差E对w5的导数。要求误差E对w5的导数，需要先求误差E对out o1的导数，再求out o1对net o1的导数，最后再求net o1对w5的导数，经过这个处理，我们就可以求出误差E对w5的导数（偏导），如下图所示：



$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$$out_{o1} = \frac{1}{1 + e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

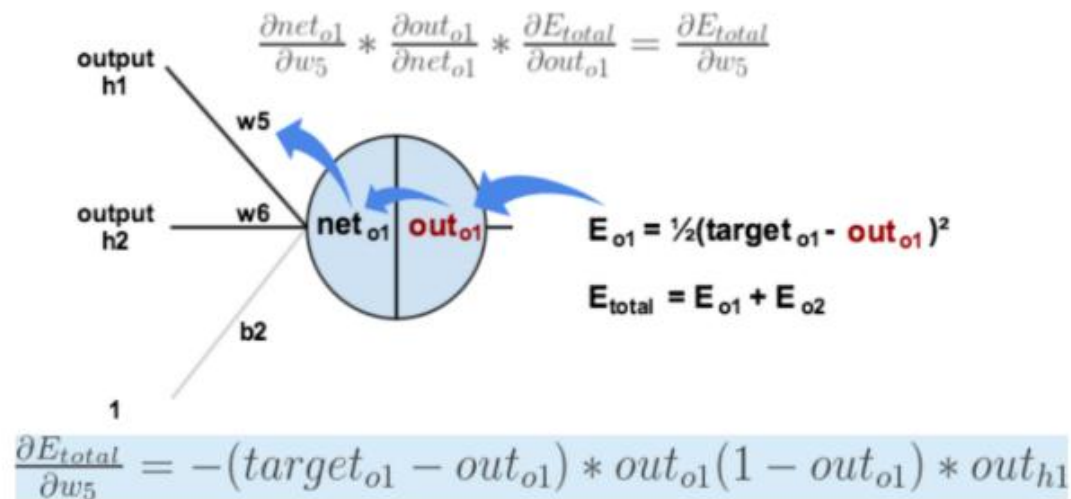
$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

反向传播(BP算法)[了解]

导数（梯度）已经计算出来了，下面就是反向传播与参数更新过程：



$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

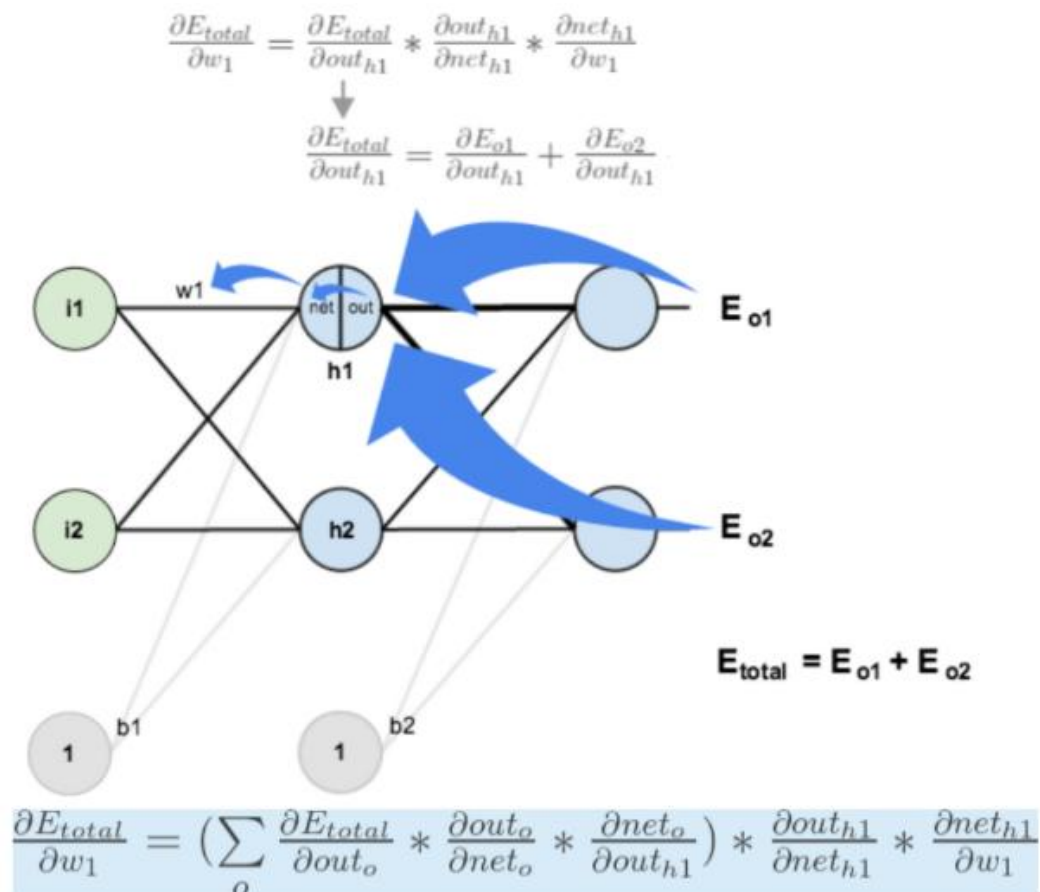
$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

反向传播(BP算法)[了解]

如果要想求误差E对w1的导数，误差E对w1的求导路径不止一条，这会稍微复杂一点，但换汤不换药，计算过程如下所示：



$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$



总结

1. 前向传播和反向传播是设么？

前向传播：指的是数据输入的神经网络中，逐层向前传输，一直到运算到输出层为止。

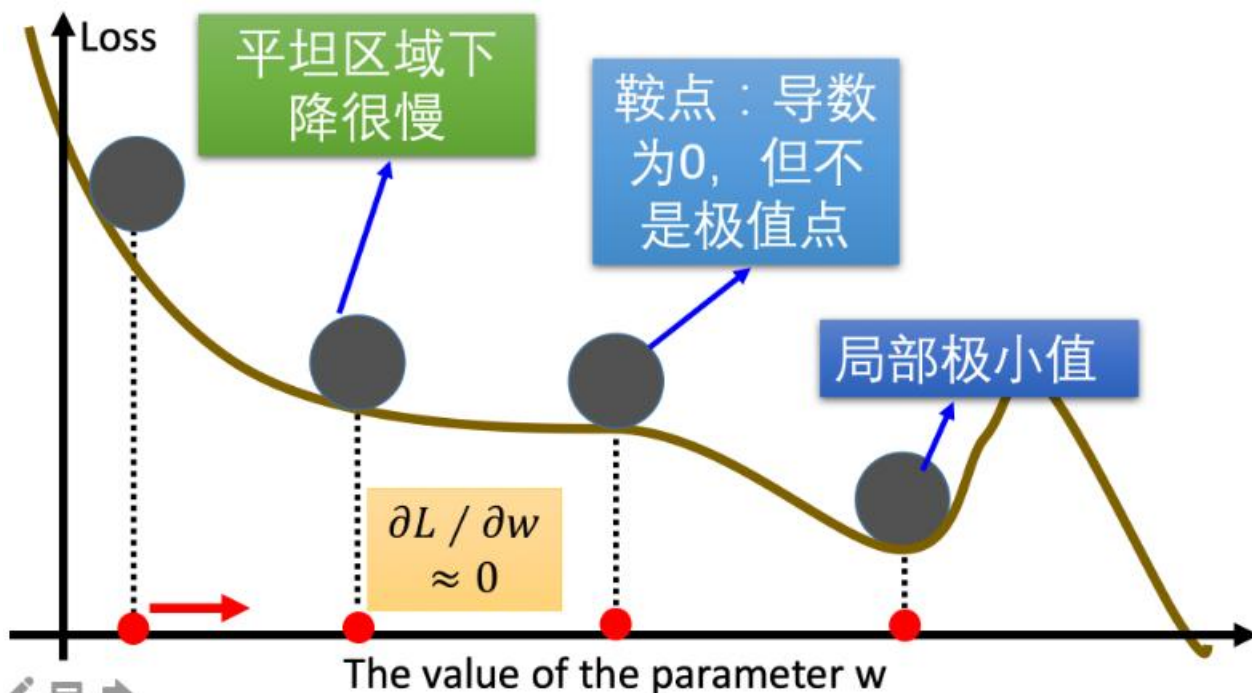
反向传播（Back Propagation）：利用损失函数 ERROR，从后往前，结合梯度下降算法，依次求各个参数的偏导，并进行参数更新

梯度下降的优化方法

梯度下降优化算法中，可能会碰到以下情况：

1. 碰到平缓区域，梯度值较小，参数优化变慢
2. 碰到“鞍点”，梯度为 0，参数无法优化
3. 碰到局部最小值，参数不是最优

对于这些问题，出现了一些对梯度下降算法的优化方法，例如：Momentum、AdaGrad、RMSprop、Adam 等



梯度下降的优化方法-指数加权平均

指数移动加权平均则是参考各数值，并且各数值的权重都不同，距离越远的数字对平均数计算的贡献就越小（权重较小），距离越近则对平均数的计算贡献就越大（权重越大）。

比如：明天气温怎么样，和昨天气温有很大关系，而和一个月前的气温关系就小一些。

计算公式可以用下面的式子来表示：

$$S_t = \begin{cases} Y_1, & t = 0 \\ \beta * S_{t-1} + (1 - \beta) * Y_t, & t > 0 \end{cases}$$

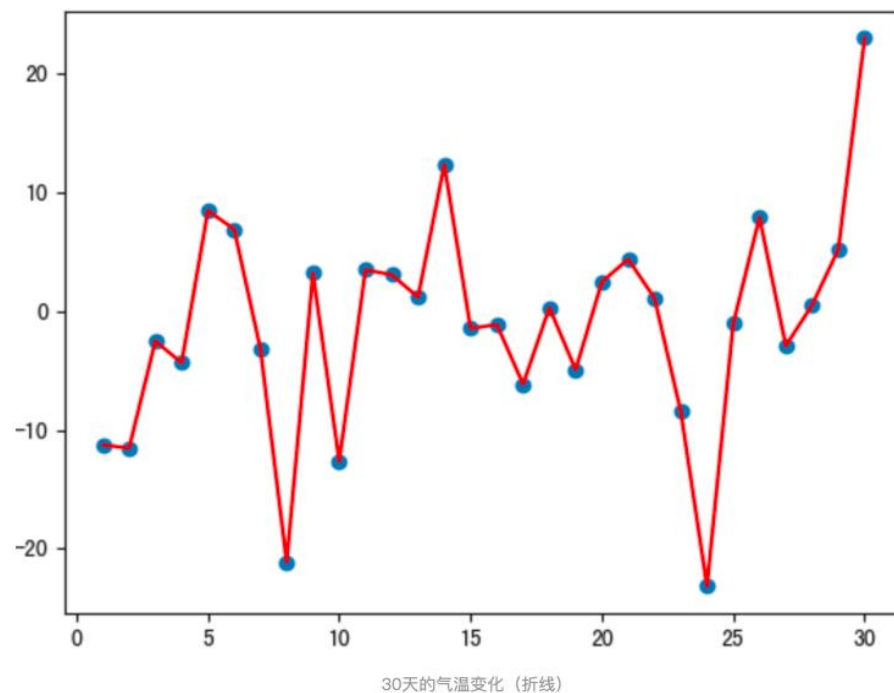
- S_t 表示指数加权平均值；
- Y_t 表示 t 时刻的值；
- β 调节权重系数，该值越大平均数越平缓。

下面通过代码来看结果，随机产生 30 天的气温数据：

梯度下降的优化方法-指数加权平均

```
import torch
import matplotlib.pyplot as plt

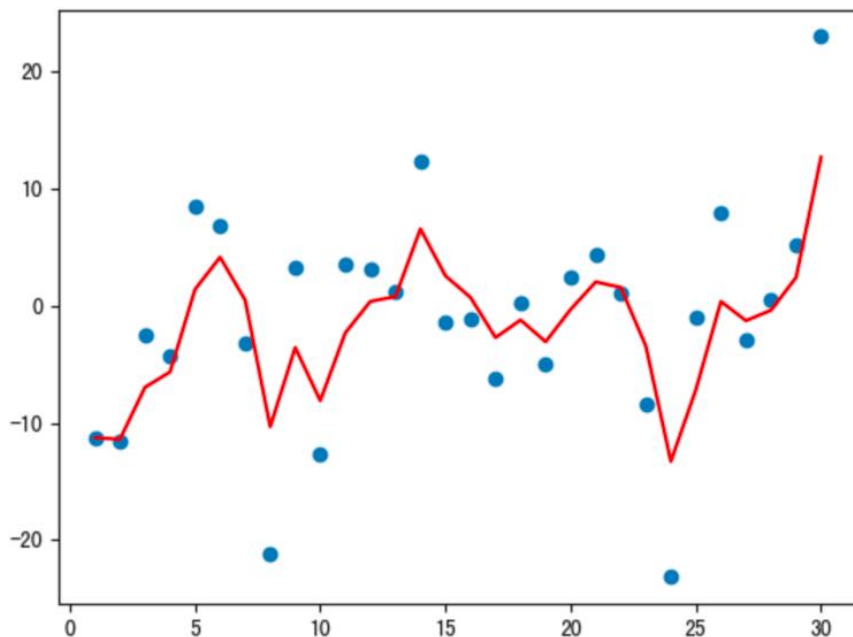
ELEMENT_NUMBER = 30
# 1. 实际平均温度
def test01():
    # 固定随机数种子
    torch.manual_seed(0)
    # 产生30天的随机温度
    temperature = torch.randn(size=[ELEMENT_NUMBER,]) * 10
    print(temperature)
    # 绘制平均温度
    days = torch.arange(1, ELEMENT_NUMBER + 1, 1)
    plt.plot(days, temperature, color='r')
    plt.scatter(days, temperature)
    plt.show()
```



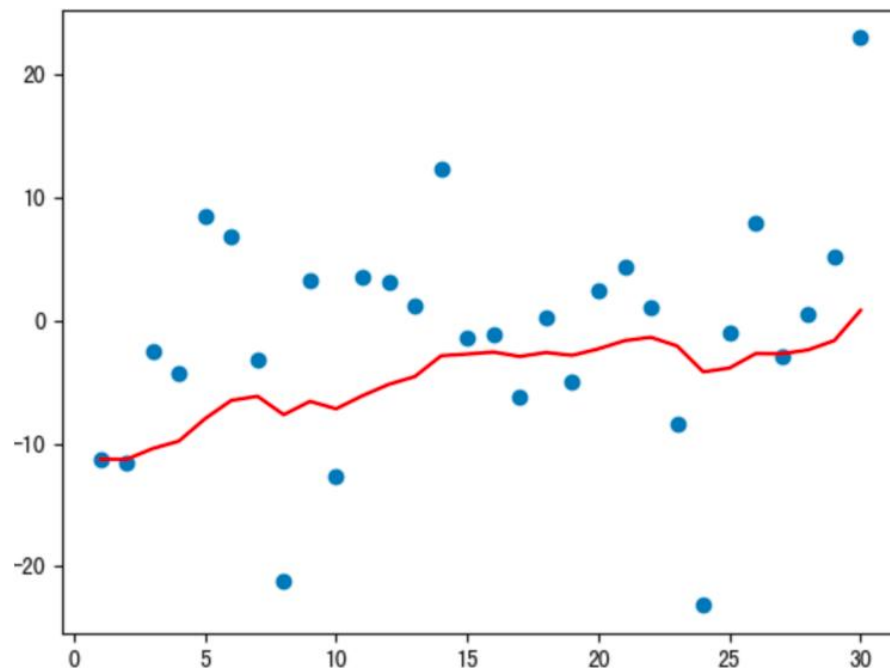
梯度下降的优化方法-指数加权平均

```
# 2. 指数加权平均温度
def test02(beta=0.9):
    torch.manual_seed(0) # 固定随机数种子
    temperature = torch.randn(size=[ELEMENT_NUMBER,]) * 10 # 产生30天的随机温度
    exp_weight_avg = []
    for idx, temp in enumerate(temperature, 1): # 从下标1开始
        # 第一个元素的 EWA 值等于自身
        if idx == 1:
            exp_weight_avg.append(temp)
            continue
        # 第二个元素的 EWA 值等于上一个 EWA 乘以  $\beta$  + 当前气温乘以  $(1-\beta)$ 
        new_temp = exp_weight_avg[idx - 2] * beta + (1 - beta) * temp
        exp_weight_avg.append(new_temp)
    days = torch.arange(1, ELEMENT_NUMBER + 1, 1)
    plt.plot(days, exp_weight_avg, color='r')
    plt.scatter(days, temperature)
    plt.show()
```

梯度下降的优化方法-指数加权平均



$\beta=0.5$ 的气温变化 (折线)



$\beta=0.9$ 的气温变化 (折线)

上图是 β 为0.5和0.9时的结果，从中可以看出：

- 指数加权平均绘制出的气温变化曲线更加平缓，
- β 的值越大，则绘制出的折线越加平缓，波动越小。 $(1-\beta)$ 越小, t 时刻的 S_t 越不依赖 Y_t 的值)
- β 值一般默认都是 0.9

梯度下降的优化方法-动量算法Momentum

梯度计算公式:

$$s_t = \beta s_{t-1} + (1 - \beta) g_t$$

参数更新公式:

$$w_t = w_{t-1} - \eta s_t$$

s_t 是当前时刻指数加权平均梯度值

s_{t-1} 是历史指数加权平均梯度值

g_t 是当前时刻的梯度值

β 是调节权重系数，通常取 0.9 或 0.99

η 是学习率

w_t 是当前时刻模型权重参数

梯度下降的优化方法-动量算法Momentum

咱们举个例子，假设：权重 β 为 0.9，例如：

第一次梯度值： $s1 = g1 = w1$

第二次梯度值： $s2 = 0.9*s1 + g2*0.1$

第三次梯度值： $s3 = 0.9*s2 + g3*0.1$

第四次梯度值： $s4 = 0.9*s3 + g4*0.1$

1. w 表示初始梯度
2. g 表示当前轮数计算出的梯度值
3. s 表示历史梯度移动加权平均值

梯度下降公式中梯度的计算，就不再是当前时刻 t 的梯度值，而是历史梯度值的指数移动加权平均值。

公式修改为：

$$W_t = W_{t-1} - \eta * S_t$$

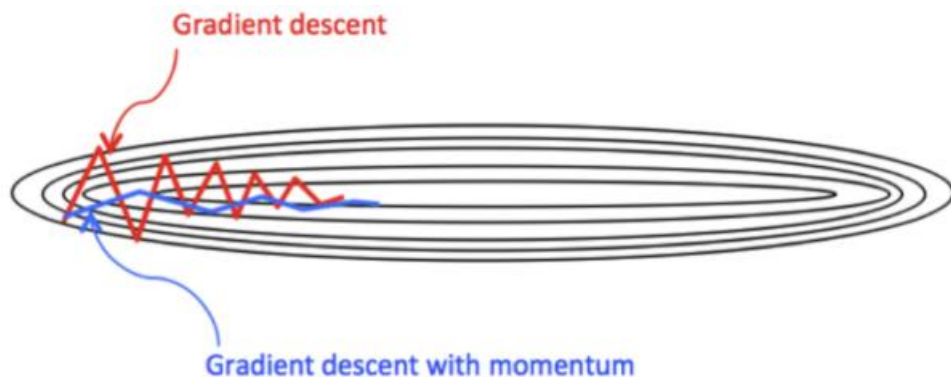
W_t ：当前时刻模型权重参数

S_t ：当前时刻指数加权平均梯度值

η ：学习率

梯度下降的优化方法-动量算法Momentum

Monmomentum 优化方法是如何一定程度上克服 “平缓”、”鞍点” 的问题呢？



- 当处于鞍点位置时，由于当前的梯度为 0，参数无法更新。但是 Momentum 动量梯度下降算法已经在先前积累了一些梯度值，很有可能使得跨过鞍点。
- 由于 mini-batch 普通的梯度下降算法，每次选取少数的样本梯度确定前进方向，可能会出现震荡，使得训练时间变长。Momentum 使用移动加权平均，平滑了梯度的变化，使得前进方向更加平缓，有利于加快训练过程。

梯度下降的优化方法-动量算法Momentum

```
def test01():  
    # 1 初始化权重参数  
    w = torch.tensor([1.0], requires_grad=True, dtype=torch.float32)  
    loss = ((w ** 2) / 2.0).sum()  
  
    # 2 实例化优化方法: SGD 指定参数beta=0.9  
    optimizer = torch.optim.SGD([w], lr=0.01, momentum=0.9)  
  
    # 3 第1次更新 计算梯度, 并对参数进行更新  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
  
    print('第1次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))  
    # 4 第2次更新 计算梯度, 并对参数进行更新  
    # 使用更新后的参数机选输出结果  
    loss = ((w ** 2) / 2.0).sum()  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
    print('第2次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))
```

输出结果:

第1次: 梯度w.grad: 1.0000000000, 更新后的权重:0.99000000095
第2次: 梯度w.grad: 0.99000000095, 更新后的权重:0.9711000323

梯度下降的优化方法-adaGrad

AdaGrad 通过对不同的参数分量使用不同的学习率，AdaGrad 的学习率总体会逐渐减小。

其计算步骤如下：

1. 初始化学习率 η 、初始化参数 w 、小常数 $\sigma = 1e-10$
2. 初始化梯度累计变量 $s = 0$
3. 从训练集中采样 m 个样本的小批量，计算梯度 g_t
4. 累积平方梯度： $s_t = s_{t-1} + g_t \odot g_t$ ， \odot 表示各个分量相乘
5. 学习率 η 的计算公式如下：

$$\eta = \frac{\eta}{\sqrt{s_t} + \sigma}$$

6. 权重参数更新公式如下：

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t} + \sigma} * g_t$$

7. 重复 3-7 步骤

AdaGrad 缺点是可能会使得学习率过早、过量的降低，导致模型训练后期学习率太小，较难找到最优解。

梯度下降的优化方法-AdaGrad

```
def test02():  
    # 1 初始化权重参数  
    w = torch.tensor([1.0], requires_grad=True, dtype=torch.float32)  
    loss = ((w ** 2) / 2.0).sum()  
  
    # 2 实例化优化方法: adagrad 优化方法  
    optimizer = torch.optim.Adagrad ([w], lr=0.01)  
  
    # 3 第1次更新 计算梯度, 并对参数进行更新  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
  
    print('第1次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))  
    # 4 第2次更新 计算梯度, 并对参数进行更新  
    # 使用更新后的参数机选输出结果  
    loss = ((w ** 2) / 2.0).sum()  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
    print('第2次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))
```

输出结果:

第1次: 梯度w.grad: 1.000000,
更新后的权重:0.990000 第2次:
梯度w.grad: 0.990000, 更新后
的权重:0.982965

梯度下降的优化方法RMSProp

RMSProp 优化算法是对 AdaGrad 的优化。最主要的不同是，其使用指数加权平均梯度替换历史梯度的平方和。其计算过程如下：

1. 初始化学率 η 、初始化权重参数 w 、小常数 $\sigma = 1e-10$
2. 初始化梯度累计变量 $s = 0$
3. 从训练集中采样 m 个样本的小批量，计算梯度 g_t
4. 使用指数加权平均累计历史梯度， \odot 表示各个分量相乘，公式如下：

$$s_t = \beta s_{t-1} + (1-\beta) g_t \odot g_t$$

5. 学习率 η 的计算公式如下：

$$\eta = \frac{\eta}{\sqrt{s_t} + \sigma}$$

6. 权重参数更新公式如下：

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_t} + \sigma} * g_t$$

7. 重复 3-7 步骤

梯度下降的优化方法

```
def test03():  
    # 1 初始化权重参数  
    w = torch.tensor([1.0], requires_grad=True, dtype=torch.float32)  
    loss = ((w ** 2) / 2.0).sum()  
  
    # 2 实例化优化方法: RMSprop算法, 其中alpha对应这beta  
    optimizer = torch.optim.RMSprop([w], lr=0.01,alpha=0.9)  
  
    # 3 第1次更新 计算梯度, 并对参数进行更新  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
  
    print('第1次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))  
    # 4 第2次更新 计算梯度, 并对参数进行更新  
    # 使用更新后的参数机选输出结果  
    loss = ((w ** 2) / 2.0).sum()  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
    print('第2次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))
```

输出结果:

第1次: 梯度w.grad: 1.000000, 更新后的权重:0.968377 第2次: 梯度w.grad: 0.968377, 更新后的权重:0.945788

梯度下降的优化方法-Adam

- Momentum 使用指数加权平均计算当前的梯度值
- AdaGrad、RMSProp 使用自适应的学习率
- Adam优化算法（Adaptive Moment Estimation，自适应矩估计）将 Momentum 和 RMSProp 算法结合在一起
 - 修正梯度：使用梯度的指数加权平均
 - 修正学习率：使用梯度平方的指数加权平均
- 原理：Adam 是结合了 **Momentum** 和 **RMSProp** 优化算法的优点的自适应学习率算法。它计算了梯度的一阶矩（平均值）和二阶矩（梯度的方差）的自适应估计，从而动态调整学习率。
- 梯度计算公式：
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$s_t = \beta_2 s_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{s}_t = \frac{s_t}{1 - \beta_2^t}$$
- 权重参数更新公式：
$$w_t = w_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \hat{m}_t$$

其中， m_t 是梯度的一阶矩估计， s_t 是梯度的二阶矩估计， \hat{m}_t 和 \hat{s}_t 是偏差校正后的估计。

梯度下降的优化方法-Adam

```
def test04():  
    # 1 初始化权重参数  
    w = torch.tensor([1.0], requires_grad=True)  
    loss = ((w ** 2) / 2.0).sum()  
  
    # 2 实例化优化方法: Adam算法, 其中betas是指数加权的系数  
    optimizer = torch.optim.Adam([w], lr=0.01, betas=[0.9, 0.99])  
  
    # 3 第1次更新 计算梯度, 并对参数进行更新  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
  
    print('第1次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))  
    # 4 第2次更新 计算梯度, 并对参数进行更新  
    # 使用更新后的参数机选输出结果  
    loss = ((w ** 2) / 2.0).sum()  
    optimizer.zero_grad()  
    loss.backward()  
    optimizer.step()  
    print('第2次: 梯度w.grad: %f, 更新后的权重:%f' % (w.grad.numpy(), w.detach().numpy()))
```

输出结果:

第1次: 梯度w.grad: 1.000000,
更新后的权重:0.990000
第2次: 梯度w.grad: 0.990000,
更新后的权重:0.980003



总结

1. 梯度下降算法优化的目的?

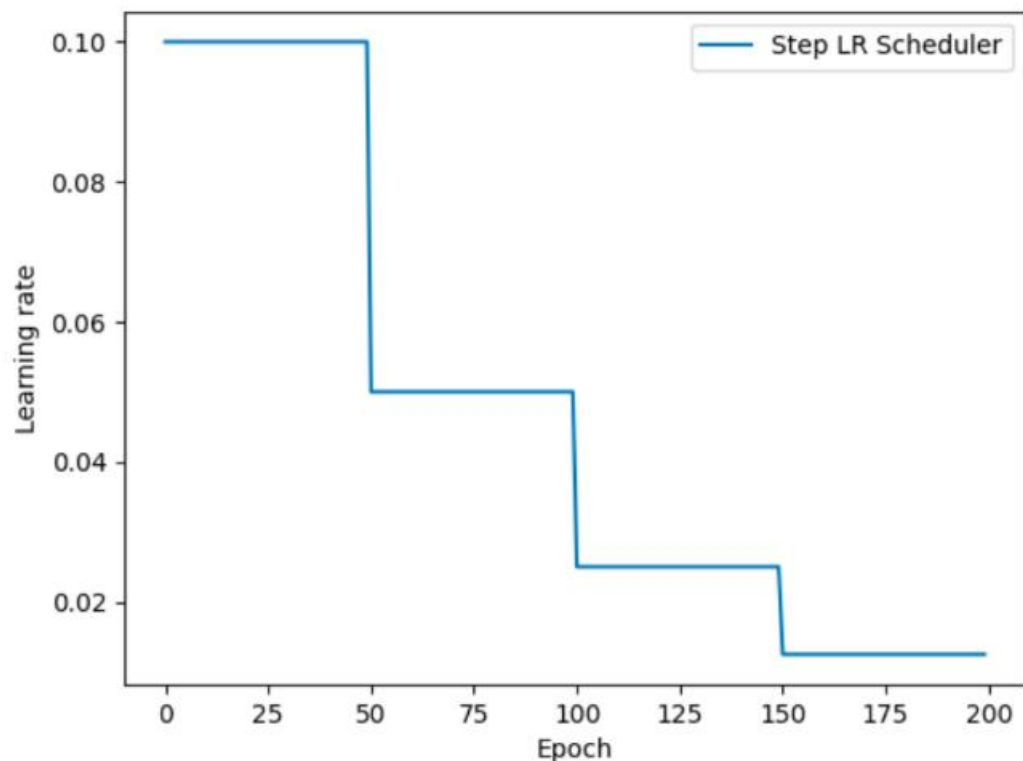
梯度下降优化算法中，可能会碰到平缓区域，“鞍点”等问题

2. 梯度下降算法的优化有哪些?

动量法, Adagrad, RMSProp, Adam

学习率衰减方法-等间隔学习率衰减

等间隔学习率衰减方式如下所示：



```
# step_size: 调整间隔数=50
# gamma: 调整系数=0.5
# 调整方式:  $lr = lr * gamma$ 
lr_scheduler.StepLR(optimizer, step_size, gamma=0.1)
```

学习率衰减方法-等间隔学习率衰减

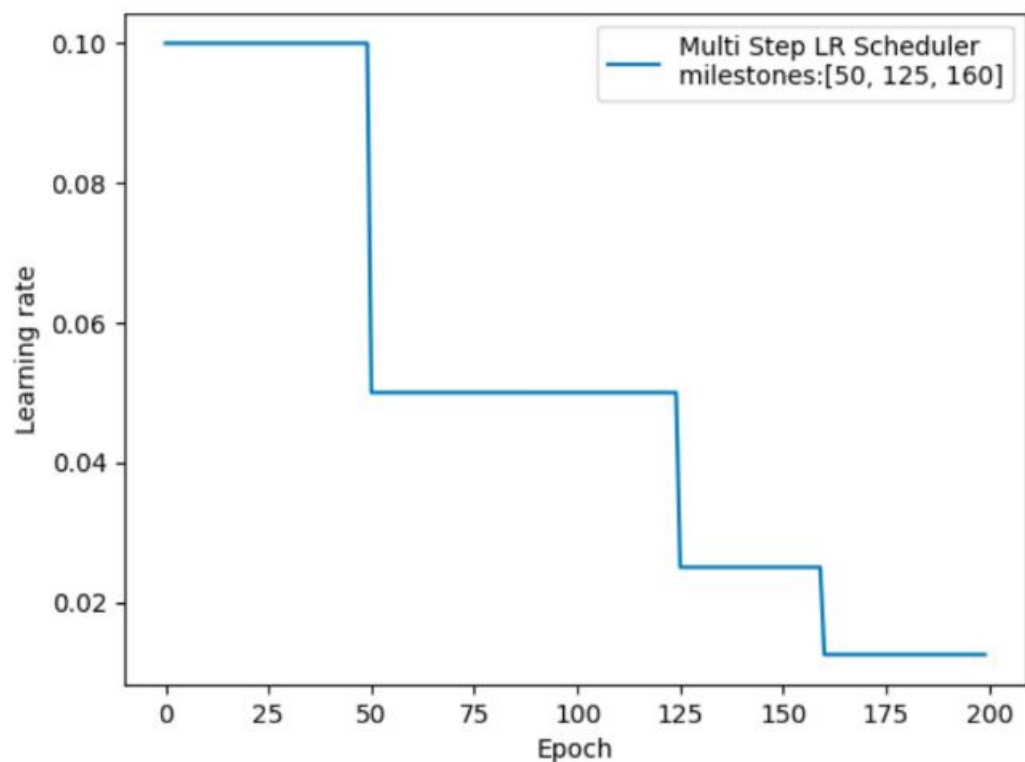
```
def test_StepLR():  
    # 0. 参数初始化  
    LR = 0.1 # 设置学习率初始化值为0.1  
    iteration = 10  
    max_epoch = 200  
    # 1 初始化参数  
    y_true = torch.tensor([0])  
    x = torch.tensor([1.0])  
    w = torch.tensor([1.0], requires_grad=True)  
    # 2. 优化器  
    optimizer = optim.SGD([w], lr=LR, momentum=0.9)  
    # 3. 设置学习率下降策略  
    scheduler_lr = optim.lr_scheduler.StepLR(optimizer, step_size=50, gamma=0.5)  
    # 4. 获取学习率的值和当前的epoch  
    lr_list, epoch_list = list(), list()
```

学习率衰减方法-等间隔学习率衰减方法

```
for epoch in range(max_epoch):  
    lr_list.append(scheduler_lr.get_last_lr()) # 获取当前lr  
    epoch_list.append(epoch) # 获取当前的epoch  
    for i in range(iteration): # 遍历每一个batch数据  
        loss = ((w*x-y_true)**2)/2.0 # 目标函数  
        optimizer.zero_grad()  
        # 反向传播  
        loss.backward()  
        optimizer.step()  
        # 更新下一个epoch的学习率  
        scheduler_lr.step()  
# 5. 绘制学习率变化的曲线  
plt.plot(epoch_list, lr_list, label="Step LR Scheduler")  
plt.xlabel("Epoch")  
plt.ylabel("Learning rate")  
plt.legend()  
plt.show()
```

学习率优化方法-指定间隔学习率衰减

指定间隔学习率衰减的效果如下：



```
# milestones: 设定调整轮次:[50, 125, 160]
# gamma: 调整系数
# 调整方式:  $lr = lr * gamma$ 
optim.lr_scheduler.MultiStepLR(optimizer, milestones,
gamma=0.1)
```

学习率优化方法-指定间隔学习率衰减

```
def test_MultiStepLR():  
    torch.manual_seed(1)  
    LR = 0.1  
    iteration = 10  
    max_epoch = 200  
    weights = torch.randn((1), requires_grad=True)  
    target = torch.zeros((1))  
    print('weights--->', weights, 'target--->', target)  
    optimizer = optim.SGD([weights], lr=LR, momentum=0.9)  
    # 设定调整时刻数  
    milestones = [50, 125, 160]  
    # 设置学习率下降策略  
    scheduler_lr = optim.lr_scheduler.MultiStepLR(optimizer,  
milestones=milestones, gamma=0.5)  
    lr_list, epoch_list = list(), list()
```

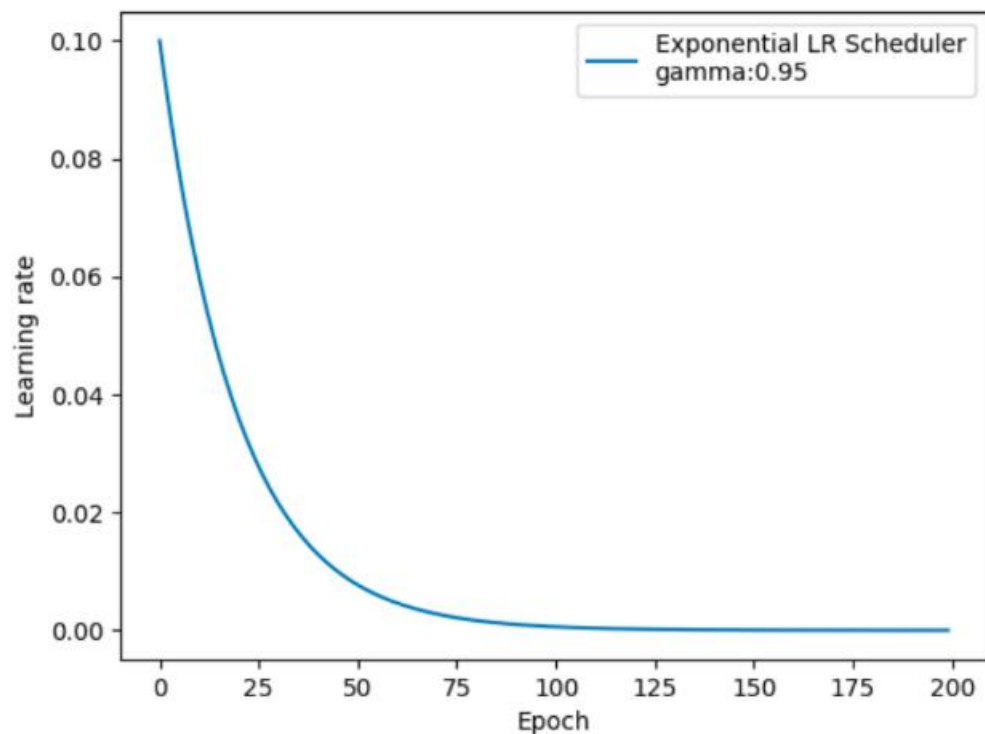
学习率优化方法-指定间隔学习率衰减

```
for epoch in range(max_epoch):
    lr_list.append(scheduler_lr.get_last_lr())
    epoch_list.append(epoch)
    for i in range(iteration):
        loss = torch.pow((weights - target), 2)
        optimizer.zero_grad()
        # 反向传播
        loss.backward()
        # 参数更新
        optimizer.step()
    # 更新下一个epoch的学习率
    scheduler_lr.step()

plt.plot(epoch_list, lr_list, label="Multi Step LR Scheduler\nmilestones: {}".format(milestones))
plt.xlabel("Epoch")
plt.ylabel("Learning rate")
plt.legend()
plt.show()
```


学习率优化方法-按指数学习率衰减

按指数学习率衰减的效果如下：



```
# gamma: 指数的底  
# 调整方式  
#  $lr = lr * \gamma^{\text{epoch}}$   
optim.lr_scheduler.ExponentialLR(optimizer, gamma)
```

学习率优化方法-按指数学习率衰减

```
def test_ExponentialLR():  
    # 0. 参数初始化  
    LR = 0.1 # 设置学习率初始化值为0.1  
    iteration = 10  
    max_epoch = 200  
    # 1 初始化参数  
    y_true = torch.tensor([0])  
    x = torch.tensor([1.0])  
    w = torch.tensor([1.0], requires_grad=True)  
    # 2. 优化器  
    optimizer = optim.SGD([w], lr=LR, momentum=0.9)  
    # 3. 设置学习率下降策略  
    gamma = 0.95  
    scheduler_lr = optim.lr_scheduler.ExponentialLR(optimizer,  
gamma=gamma)  
    # 4. 获取学习率的值和当前的epoch  
    lr_list, epoch_list = list(), list()
```

学习率优化方法-按指数学习率衰减

```
for epoch in range(max_epoch):
    lr_list.append(scheduler_lr.get_last_lr())
    epoch_list.append(epoch)
    for i in range(iteration): # 遍历每一个batch数据
        loss = ((w*x-y_true)**2)/2.0
        optimizer.zero_grad()
        # 反向传播
        loss.backward()
        optimizer.step()
    # 更新下一个epoch的学习率
    scheduler_lr.step()
# 5. 绘制学习率变化的曲线
plt.plot(epoch_list, lr_list, label="Multi Step LR Scheduler")
plt.xlabel("Epoch")
plt.ylabel("Learning rate")
plt.legend()
plt.show()
```



总结

1.学习率调整的策略有哪些?

- 等间隔衰减
- 指定间隔衰减
- 指数衰减



目录

Contents

◆ 神经网络

◆ 损失函数

◆ 网络优化方法



◆ 正则化方法

◆ 案例-价格分类案例

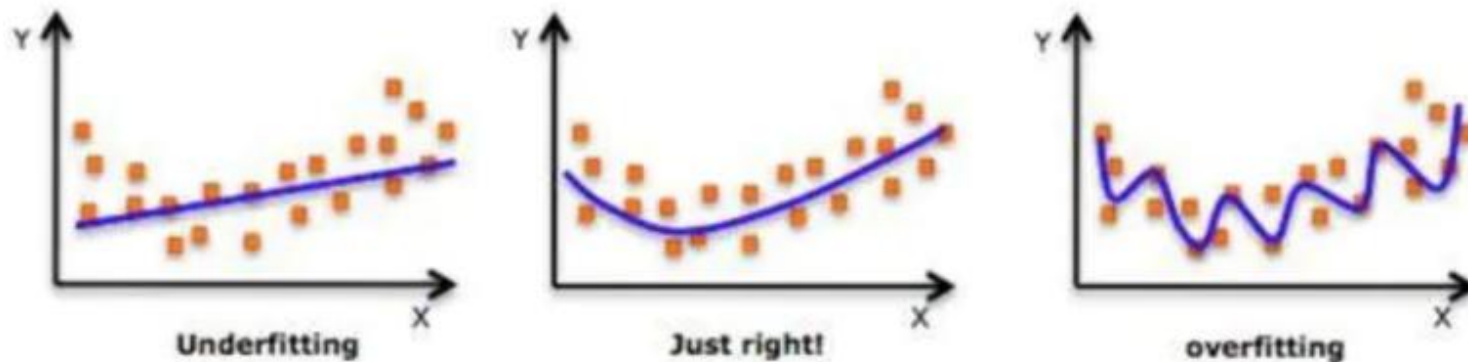


学习目标

Learning Objectives

1. 知道正则化的作用
2. 掌握随机失活DropOut策略
3. 知道BN层的作用

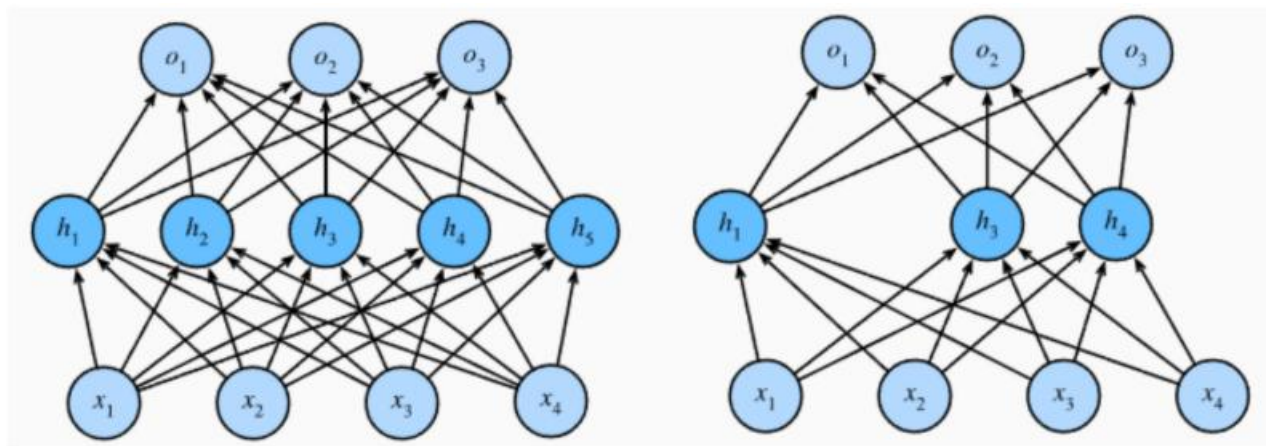
正则化



- 在设计机器学习算法时希望在新样本上的泛化能力强。许多机器学习算法都采用相关的策略来减小测试误差，这些策略被统称为正则化。
- 神经网络强大的表示能力经常遇到过拟合，所以需要使用不同形式的正则化策略。
- 目前在深度学习中使用较多的策略有范数惩罚，DropOut，特殊的网络层等，接下来我们对其进行详细的介绍。

Dropout正则化

在练神经网络中模型参数较多，在数据量不足的情况下，很容易过拟合。Dropout（随机失活）是一个简单有效的正则化方法。



- 在训练过程中，Dropout的实现是让神经元以超参数 p 的概率停止工作或者激活被置为0, 未被置为0的进行缩放，缩放比例为 $1/(1-p)$ 。训练过程可以认为是对完整的神经网络的一些子集进行训练，每次基于输入数据只更新子网络的参数。
- 在测试过程中，随机失活不起作用。

Dropout正则化

```
import torch
import torch.nn as nn

def test():
    # 初始化随机失活层
    dropout = nn.Dropout(p=0.4)
    # 初始化输入数据:表示某一层的weight信息
    inputs = torch.randint(0, 10, size=[1, 4]).float()
    layer = nn.Linear(4, 5)
    y = layer(inputs)
    y = torch.relu(y)
    print("未失活FC层的输出结果: \n", y)

    y = dropout(y)
    print("失活后FC层的输出结果: \n", y)
```

输出结果:

未失活FC层的输出结果:

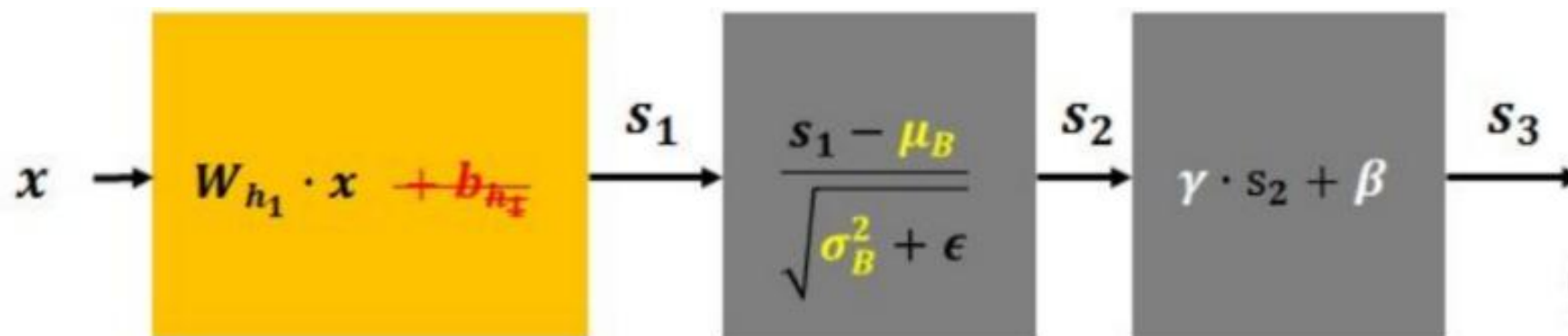
```
tensor([[0.0000, 1.8033, 1.4608, 4.5189, 6.9116]],
grad_fn=<ReluBackward0>)
```

失活后FC层的输出结果:

```
tensor([[ 0.0000, 3.0055, 2.4346, 7.5315, 11.5193]],
grad_fn=<MulBackward0>)
```

上述代码将 Dropout 层的概率 p 设置为 0.4，此时经过 Dropout 层计算的张量中就出现了很多 0，未变为0的按照 $(1/(1-0.4))$ 进行处理。

批量归一化(Batch Normalization)



先对数据标准化，再对数据重构（缩放+平移），如下所示：

$$f(x) = \lambda \cdot \frac{x - E(x)}{\sqrt{\text{Var}(x) + \epsilon}} + \beta$$

1. λ 和 β 是可学习的参数，它相当于对标准化后的值做了一个线性变换， λ 为系数， β 为偏置；
2. ϵ 通常指为 $1e-5$ ，避免分母为 0；
3. $E(x)$ 表示变量的均值；
4. $\text{Var}(x)$ 表示变量的方差；

批量归一化层在计算机视觉领域使用较多。

批量归一化(Batch Normalization)

```
import torch
import torch.nn as nn

m = nn.BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True) #
# affine参数设为True表示weight和bias将被使用

input = torch.randn(1, 2, 3, 4)
print("input-->", input)

output = m(input)
print("output-->", output)
print(output.size())

print(m.weight)
print(m.bias)
```

输出结果:

```
input--> tensor([[[[-0.2751, -1.2183, -0.5106, -0.1540],
  [-0.4585, -0.5989, -0.6063,  0.5986],
  [-0.4745,  0.1496, -1.1266, -1.2377]],

  [[ 0.2580,  1.2065,  1.4598,  0.8387],
   [-0.4586,  0.8938, -0.3328,  0.1192],
   [-0.3265, -0.6263,  0.0419, -1.2231]]]])

output--> tensor([[[[ 0.4164, -1.3889, -0.0343,  0.6484],
  [ 0.0655, -0.2032, -0.2175,  2.0889],
  [ 0.0349,  1.2294, -1.2134, -1.4262]],

  [[ 0.1340,  1.3582,  1.6853,  0.8835],
   [-0.7910,  0.9546, -0.6287, -0.0452],
   [-0.6205, -1.0075, -0.1449, -1.7779]]]])
grad_fn=<NativeBatchNormBackward0>)
torch.Size([1, 2, 3, 4])
Parameter containing:
tensor([1., 1.], requires_grad=True)
Parameter containing:
tensor([0., 0.], requires_grad=True)
```



总结

1、正则化的作用

缓解过拟合

2、随机失活DropOut策略

让神经元以超参数 p 的概率停止工作或者激活被置为0

3、BN层是什么?

对数据标准化，再对数据重构（缩放+平移）



目录

Contents

- ◆ 神经网络
- ◆ 损失函数
- ◆ 网络优化方法
- ◆ 正则化方法



- ◆ 案例-价格分类案例



学习目标

Learning Objectives

1. 掌握构建分类模型流程
2. 动手实践整个过程

需求分析

小明创办了一家手机公司，他不知道如何估算手机产品的价格。为了解决这个问题，他收集了多家公司的手机销售数据。该数据为二手手机的各个性能的数据，最后根据这些性能得到4个价格区间，作为这些二手手机售出的价格区间。

主要包括：

```
battery_power: 电池一次可储存的总能量，单位为毫安时
blue : 是否有蓝牙
clock_speed: 微处理器执行指令的速度
dual_sim: 是否支持双卡
fc: 前置摄像头百万像素
four_g: 是否有4G
int_memory: 内存 (GB)
m_dep: 移动深度 (cm)
mobile_wt: 手机重量
n_cores: 处理器内核数
pc: 主摄像头百万像素
px_height: 像素分辨率高度
px_width: 像素分辨率宽度
ram: 随机存取存储器 (兆字节)
sc_h: 手机屏幕高度 (cm)
sc_w: 手机屏幕宽度 (cm)
talk_time: 一次电池充电持续时间最长的时间
three_g: 是否有3G
touch_screen: 是否有触控屏
wifi: 是否能连wifi
price_range: 价格区间 (0, 1, 2, 3)
```

需求分析

我们需要帮助小明找出手机的功能（例如：RAM等）与其售价之间的某种关系。我们可以使用机器学习的方法来解决这个问题，也可以构建一个全连接的网络。

需要注意的是：在这个问题中，我们不需要预测实际价格，而是一个价格范围，它的范围使用 0、1、2、3 来表示，所以该问题也是一个**分类问题**。接下来我们还是按照四个步骤来完成这个任务：

- 准备训练集数据
- 构建要使用的模型
- 模型训练
- 模型预测评估

导入所需的工具包

```
# 导入相关模块

import torch
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader
import torch.nn as nn
import torch.optim as optim
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import time
```

构建数据集

数据共有 2000 条，其中 1600 条数据作为训练集，400 条数据用作测试集。我们使用 sklearn 的数据集划分工作来完成。并使用 PyTorch 的 TensorDataset 来将数据集构建为 Dataset 对象，方便构造数据集加载对象。

```
# 构建数据集
def create_dataset():
    # 使用pandas读取数据
    data = pd.read_csv('data/手机价格预测.csv')
    # 特征值和目标值
    x, y = data.iloc[:, :-1], data.iloc[:, -1]
    # 类型转换: 特征值, 目标值
    x = x.astype(np.float32)
    y = y.astype(np.int64)
    # 数据集划分
    x_train, x_valid, y_train, y_valid = train_test_split(x, y, train_size=0.8, random_state=88)
    # 构建数据集, 转换为pytorch的形式
    train_dataset = TensorDataset(torch.from_numpy(x_train.values), torch.tensor(y_train.values))
    valid_dataset = TensorDataset(torch.from_numpy(x_valid.values), torch.tensor(y_valid.values))
    # 返回结果
    return train_dataset, valid_dataset, x_train.shape[1], len(np.unique(y))
```

获取数据的结果

```
if __name__ == '__main__':  
    # 获取数据  
    train_dataset, valid_dataset, input_dim, class_num = create_dataset()  
    print("输入特征数: ", input_dim)  
    print("分类个数: ", class_num)
```

输出结果为:

输入特征数: 20

分类个数: 4

构建分类网络模型

构建全连接神经网络来进行手机价格分类，该网络主要由三个线性层来构建，使用relu激活函数。

网络共有 3 个全连接层，具体信息如下：

1. 第一层：输入为维度为 20，输出维度为：128
2. 第二层：输入为维度为 128，输出维度为：256
3. 第三层：输入为维度为 256，输出维度为：4

构建分类网络模型

```
# 构建网络模型
class PhonePriceModel(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(PhonePriceModel, self).__init__()
        # 1. 第一层: 输入为维度为 20, 输出维度为: 128
        self.linear1 = nn.Linear(input_dim, 128)
        # 2. 第二层: 输入为维度为 128, 输出维度为: 256
        self.linear2 = nn.Linear(128, 256)
        # 3. 第三层: 输入为维度为 256, 输出维度为: 4
        self.linear3 = nn.Linear(256, output_dim)

    def forward(self, x):
        # 前向传播过程
        x = torch.relu(self.linear1(x))
        x = torch.relu(self.linear2(x))
        output = self.linear3(x)
        # 获取数据结果
        return output
```

构建分类网络模型

模型实例化:

```
if __name__ == '__main__':  
    # 模型实例化  
    model = PhonePriceModel(input_dim, class_num)  
    summary(model, input_size=(input_dim,), batch_size=16)
```

```
-----  
Layer (type)           Output Shape          Param #  
=====
```

Linear-1	[16, 128]	2,688
Linear-2	[16, 256]	33,024
Linear-3	[16, 4]	1,028

```
=====
```

Total params:	36,740
Trainable params:	36,740
Non-trainable params:	0

```
-----
```

Input size (MB):	0.00
Forward/backward pass size (MB):	0.05
Params size (MB):	0.14
Estimated Total Size (MB):	0.19

```
-----
```

模型训练

网络编写完成之后，我们需要编写训练函数。所谓的训练函数，指的是输入数据读取、送入网络、计算损失、更新参数的流程，该流程较为固定。我们使用的是多分类交叉熵损失函数、使用 SGD 优化方法。最终，将训练好的模型持久化到磁盘中。

```
# 模型训练过程
def train(train_dataset,input_dim,class_num,):
    # 固定随机数种子
    torch.manual_seed(0)

    # 初始化数据加载器
    dataloader = DataLoader(train_dataset, shuffle=True, batch_size=8)

    # 初始化模型
    model = PhonePriceModel(input_dim, class_num)

    # 损失函数
    criterion = nn.CrossEntropyLoss()

    # 优化方法
    optimizer = optim.SGD(model.parameters(), lr=1e-3)

    # 训练轮数
    num_epoch = 50
```

编写训练函数

```
# 遍历每个轮次的数据
for epoch_idx in range(num_epoch):
    # 训练时间
    start = time.time()
    # 计算损失
    total_loss = 0.0
    total_num = 0
    # 遍历每个batch数据进行处理
    for x, y in dataloader:
        # 将数据送入网络中进行预测
        output = model(x)
        # 计算损失
        loss = criterion(output, y)
        # 梯度清零
        optimizer.zero_grad()
        # 反向传播
        loss.backward()
        # 参数更新
        optimizer.step()
        # 损失计算
        total_num += 1
        total_loss += loss.item()
    # 打印损失变换结果
    print('epoch: %4s loss: %.2f, time: %.2fs' % (epoch_idx + 1, total_loss / total_num, time.time() - start))
# 模型保存
torch.save(model.state_dict(), 'model/phone.pth')
```


模型训练

调用训练函数：

```
if __name__ == '__main__':  
    # 获取数据  
    train_dataset, valid_dataset, input_dim, class_num = create_dataset()  
    # 模型训练过程  
    train(train_dataset, input_dim, class_num)
```

训练结果如下所示：

```
epoch:    1 loss: 13.10, time: 0.11s  
epoch:    2 loss: 0.97, time: 0.09s  
epoch:    3 loss: 0.92, time: 0.10s  
epoch:    4 loss: 0.89, time: 0.10s  
  
...  
epoch:   48 loss: 0.69, time: 0.10s  
epoch:   49 loss: 0.71, time: 0.09s  
epoch:   50 loss: 0.71, time: 0.10s
```

编写评估函数

使用训练好的模型，对未知的样本的进行预测的过程。我们这里使用前面单独划分出来的验证集来进行评估。

```
def test(valid_dataset,input_dim,class_num):  
  
    # 加载模型和训练好的网络参数  
    model = PhonePriceModel(input_dim, class_num)  
    model.load_state_dict(torch.load('model/phone.pth'))  
  
    # 构建加载器  
    dataloader = DataLoader(valid_dataset, batch_size=8, shuffle=False)  
  
    # 评估测试集  
    correct = 0  
    # 遍历测试集中的数据  
    for x, y in dataloader:  
        # 将其送入网络中  
        output = model(x)  
        # 获取类别结果  
        y_pred = torch.argmax(output, dim=1)  
        # 获取预测正确的个数  
        correct += (y_pred == y).sum()  
    # 求预测精度  
    print('Acc: %.5f' % (correct.item() / len(valid_dataset)))
```

```
if __name__ == '__main__':  
    # 获取数据  
    train_dataset, valid_dataset, input_dim, class_num = create_dataset()  
    # 模型预测结果  
    test(valid_dataset,input_dim,class_num)
```

输出结果:

Acc: 0.64250



总结

1. 案例的整体流程是:

- 数据集构建
- 模型构建
- 模型训练
- 模型预测



今日作业

我们前面的网络模型在测试集的准确率为：0.64250，

我们可以通过以下方面进行调优：

1. 优化方法由 SGD 调整为 Adam
2. 学习率由 $1e-3$ 调整为 $1e-4$
3. 对数据进行标准化
4. 增加网络深度，即：增加网络参数量
5. 调整训练轮次
6.



传智教育旗下高端IT教育品牌