

2.预备知识

```
# 整数
torch.arange(12)

x.shape

x.reshape(3, 4)

torch.zeros((2, 3, 4)) # 2x3x4 0矩阵

torch.ones((2, 3, 4)) # 2x3x4 1矩阵

torch.tensor([1.0, 2, 4, 8])

torch.randn(3, 4) # 每个元素都从均值为0、标准差为1的标准高斯分布（正态分布）中随机采样 3x4矩阵

X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1) #dim0 0维度拼接 dim1 1维度
拼接

X == Y #匹配每一个位置
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])

X.sum() #对x里的所有元素求和 返回 1x1

A.sum(axis=0) #按列相加 一列中的每个元素加起来

A = X.numpy()
B = torch.tensor(A) #tensor 与numpy转换

# 矩阵对应元素相乘
x = torch.tensor([1,2,3])
y = torch.tensor([1,2,3])
print(x * y)
```

#求两个向量的点积

```
torch.dot(x, y)
```

#矩阵向量积 A为矩阵 x为向量

```
torch.mv(A, x)
```

两个矩阵相乘

```
torch.mm(A, B)
```

#更新梯度

```
x = torch.arange(4.0)
```

```
x.requires_grad_(True)    #设置梯度的存储空间
```

```
y = 2 * torch.dot(x, x)
```

```
y.backward()    #y反向传播
```

```
x.grad    #x更新梯度
```

```
x.grad.zero_()    #清除x的梯度
```

这个A更换了地址

```
A = A + B
```

这个A没有改变地址

```
A[:] = A + B
```

这个A没有改变地址

```
A += B
```

求tensor x中数据的个数

```
x.numel()
```

将张量变成一个标量

```
a.item()
```

通过分配新内存，将A的一个副本分配给B

```
B = A.clone()
```

对应位置元素相乘

```
A * B
```

3.线性回归

线性回归简单实现

#线性回归简单实现

```
import torch
from d2l import torch as d2l
from torch.utils import data
import numpy as np
from torch import nn

true_w = torch.tensor([2, -3.4])
true_b = 4.2

#通过权重和偏置生成数据集
features, labels = d2l.synthetic_data(true_w, true_b, 1000)

#把数据集加载到dataloader
def load_array(data_arrays, batch_size, is_train=True): #@save
    """构造一个PyTorch数据迭代器"""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)

net = nn.Sequential(nn.Linear(2, 1))

#初始化网络一开始的偏置和权重
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)

loss = nn.MSELoss()
# 定义优化器
trainer = torch.optim.SGD(net.parameters(), lr=0.03)

num_epochs = 3
for epoch in range(num_epochs):
    #遍历数据加载器
    for X, y in data_iter:
        #计算损失
        l = loss(net(X), y)
```

```
#反向传播之前 优化器清空梯度
trainer.zero_grad()
# 损失函数反向传播
l.backward()
#优化器更新梯度
trainer.step()
# 计算第一次迭代后 特征与标签的损失
l = loss(net(features), labels)
print(f'epoch {epoch + 1}, loss {l:f}')
```

训练必备东西

- 数据集
- 数据加载器
- 损失函数
- 优化器

训练过程

- 遍历数据与标签
- 计算损失
- 优化器清空原梯度
- 损失函数反向传播
- 优化器更新梯度

softmax简单实现

- 为了避免softmax的计算值溢出 把softmax的计算结合到 loss里面

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 256
#加载数据集
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)

#定义模型
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

#定义初始化权重函数
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

#网络初始化权重
net.apply(init_weights)

loss = nn.CrossEntropyLoss(reduction='none')

trainer = torch.optim.SGD(net.parameters(), lr=0.1)

num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

4.多层感知机

```
x.detach() #把x张量的数据值提取出来，不保留x的梯度信息等等
```

```
# 清除以前的梯度
```

```
x.grad.data.zero_()
```

```
#y对张量like x反向传播 这个方向传播只反映方向 retain_graph=True 保留这个计算图 后面可以继续调用
```

```
y.backward(torch.ones_like(x),retain_graph=True)
```

```
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

多层感知机的简单实现

```
import torch
```

```
from torch import nn
```

```
from d2l import torch as d2l
```

```
net = nn.Sequential(nn.Flatten(),  
                    nn.Linear(784, 256),  
                    nn.ReLU(),  
                    nn.Linear(256, 10))
```

```
#初始化权重函数
```

```
def init_weights(m):
```

```
    if type(m) == nn.Linear:
```

```
        nn.init.normal_(m.weight, std=0.01)
```

```
#初始化权重
```

```
net.apply(init_weights);
```

```
batch_size, lr, num_epochs = 256, 0.1, 10
```

```
loss = nn.CrossEntropyLoss(reduction='none')
```

```
#定义优化器
```

```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

```
#迭代数据
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

```
#训练模型
```

```
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

Dropout

```
# (torch.rand(X.shape) > 0.5) 生成概率大于0.5的 0, 1矩阵 该矩阵内元素只有0或1
mask = (torch.rand(X.shape) > 0.5).float()
```

```
import torch
from torch import nn
from d2l import torch as d2l
```

#定义dropout函数

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # 在本情况中, 所有元素都被丢弃
    if dropout == 1:
        return torch.zeros_like(X)
    # 在本情况中, 所有元素都被保留
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

```
X = torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

dropout训练

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    # 在第一个全连接层之后添加一个dropout层
                    nn.Dropout(dropout1),
                    nn.Linear(256, 256),
```

```

nn.ReLU(),
# 在第二个全连接层之后添加一个dropout层
nn.Dropout(dropout2),
nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)
net.apply(init_weights)

trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer

```

预测房价

```

import hashlib
import os
import tarfile
import zipfile
import requests
import pandas as pd

test_data = pd.read_csv('./house/test.csv')

train_data = pd.read_csv('./house/train.csv')

train_data.drop('Id',axis = 1)

all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:])) #从行的维度
拼接数据 不要ID列

# 返回不是字符串类型数据的索引。
# all_features.dtypes != 'object': 如果all_features每列数据类型不为 字符串(object)则返回index
下标
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index

#处理不为字符串的数据和缺失的数据

```



```

#将所有缺失的值替换为相应特征的平均值
#所有数据 apply lambda表达式
all_features[numeric_features] = all_features[numeric_features].apply(
lambda x: (x - x.mean()) / (x.std()))

# 在标准化数据之后，所有均值消失，因此我们可以将缺失值设置为0
all_features[numeric_features] = all_features[numeric_features].fillna(0)

# “Dummy_na=True”将“na”（缺失值）视为有效的特征值，并为其创建指示符特征
# 例如，“MSZoning”包含值“RL”和“Rm”。
# 我们将创建两个新的指示器特征“MSZoning_RL”和“MSZoning_RM”，其值为0或1。根据独热编码，如果
# “MSZoning”的原始值为“RL”，则：“MSZoning_RL”为1，“MSZoning_RM”为0
# dummy_na=True 创造新的数据列 来表示当独热编码数据列为空时的表示
all_features = pd.get_dummies(all_features, dummy_na=True)

# 获取训练数据的行数
n_train = train_data.shape[0]

import torch
import torch.nn as nn

#从拼接数据中获取训练的数据 转换成tensor
train_features = torch.tensor(all_features[:n_train].values, dtype=torch.float32)
#从拼接数据中获取测试的数据 转换成tensor
test_features = torch.tensor(all_features[n_train:].values, dtype=torch.float32)

#获取标签数据 该数据为 SalePrice列的值 然后reshape(-1, 1) 转化为 n行1列 广播机制
train_labels = torch.tensor(
train_data.SalePrice.values.reshape(-1, 1), dtype=torch.float32)

loss = nn.MSELoss()

#获取列数
in_features = train_features.shape[1]

in_features

def get_net():
    net = nn.Sequential(nn.Linear(in_features,1))
    return net

```

```

# 输出均分根误差
def log_rmse(net, features, labels):
    # 为了在取对数时进一步稳定该值, 将小于1的值设置为1
    # clamp函数限制 net的输出值 输出值小于1的值设置为1 最大值为float('inf') 无穷大
    clipped_preds = torch.clamp(net(features), 1, float('inf'))

    # torch.log(clipped_preds): 这是对神经网络的预测值 clipped_preds 进行自然对数变换,
    # 这种变换通常用于处理正数数据, 可能是为了使数据更接近正态分布
    # torch.sqrt 是 PyTorch 中的一个函数, 用于计算输入张量中每个元素的平方根
    rmse = torch.sqrt(loss(torch.log(clipped_preds), torch.log(labels)))
    # item() 函数用于从 PyTorch 张量中提取标量值
    return rmse.item()

from d2l import torch as d2l

def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []

    # 定义数据加载器
    train_iter = d2l.load_array((train_features, train_labels), batch_size)

    optimizer = torch.optim.Adam(net.parameters(), lr = learning_rate, weight_decay =
weight_decay)

    for epoch in range(num_epochs):
        for X, y in train_iter:
            optimizer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            optimizer.step()
        # 获取train集的均分根误差
        train_ls.append(log_rmse(net, train_features, train_labels))

        # 获取test集的均分根误差
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls

# 使用所有数据进行训练
def train_and_pred(train_features, test_features, train_labels, test_data,
num_epochs, lr, weight_decay, batch_size):

```

```

net = get_net()
train_ls, _ = train(net, train_features, train_labels, None, None,
num_epochs, lr, weight_decay, batch_size)
d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
print(f'训练log rmse: {float(train_ls[-1]):f}')
# 将网络应用于测试集。

#将预测的结果转化为numpy格式
preds = net(test_features).detach().numpy()

# 将其重新格式化以导出到Kaggle
# preds.reshape(1, -1)[0]将数据转化为一行 取出第一行
# 将这一行结果赋值给 SalePrice列
test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
submission.to_csv('submission.csv', index=False)

train_and_pred(train_features, test_features, train_labels, test_data,
num_epochs, lr, weight_decay, batch_size)

```

5.深度学习计算

保存文件

```

import torch
from torch import nn
from torch.nn import functional as F
x = torch.arange(4)

#保存张量 把x张量存到 x-file
torch.save(x, 'x-file')

```

```

#加载张量
x2 = torch.load('x-file')

y = torch.zeros(4)
#存储张量列表
torch.save([x, y], 'x-files')
#读取张量列表
x2, y2 = torch.load('x-files')

#存储读取张量字典
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2

#存储 模型参数

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
torch.save(net.state_dict(), 'mlpparams.pth')

#恢复模型参数
clone = MLP()
clone.load_state_dict(torch.load('mlpparams.pth'))

# 把神经网络切换到评估模式
clone.eval()

```

配置GPU

```

torch.device('cpu')
torch.device('cuda')
torch.device('cuda:1')

#统计GPU个数

```

```

torch.cuda.device_count()

# 调用第i个GPU 如果不可用则调用CPU
def try_gpu(i=0):
    #@save
    """如果存在, 则返回gpu(i), 否则返回cpu()"""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

# 调用多个GPU 返回GPU列表 如果不可用则调用CPU
def try_all_gpus():
    #@save
    """返回所有可用的GPU, 如果没有GPU, 则返回[cpu(),]"""
    devices = [torch.device(f'cuda:{i}')]
    for i in range(torch.cuda.device_count()):
        return devices if devices else [torch.device('cpu')]

# 把张量存储在GPU上
X = torch.ones(2, 3, device=try_gpu())

# 把张量存放在1gpu上
Y = torch.rand(2, 3, device=try_gpu(1))

# 把x张量复制到 GPU1上
Z = X.cuda(1)

#张量相加的时候要把张量复制到同一张GPU上 不然会错
X+Z

# 把模型放到GPU上
#try_gpu() 是上面定义的函数

net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())

```

6.卷积神经网络

卷积示例

```
# 构造一个二维卷积层，它具有1个输出通道和形状为（1， 2）的卷积核
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)
# 这个二维卷积层使用四维输入和输出格式（批量大小、通道、高度、宽度），
# 其中批量大小和通道数都为1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2
# 学习率
for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    # l.sum().backward() 通常用于计算整体损失的梯度，而 l.backward() 用于计算每个样本的梯度
    l.sum().backward()
    # 迭代卷积核
    # 这行代码的作用是使用梯度下降法来更新卷积层的权重。具体地说，它从权重张量中减去学习率乘以梯度的
    # 值，以便在训练过程中逐渐减小损失函数
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i+1}, loss {l.sum():.3f}')
```

池化

```
pool2d = nn.MaxPool2d(3)    3x3的最大池化
```

LeNet

```
@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """用GPU训练模型（在第六章定义）"""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
```

```

        nn.init.xavier_uniform_(m.weight)
net.apply(init_weights)
print('training on', device)

net.to(device)
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['train loss', 'train acc', 'test acc'])
timer, num_batches = d2l.Timer(), len(train_iter)
for epoch in range(num_epochs):
    # 训练损失之和, 训练准确率之和, 样本数
    metric = d2l.Accumulator(3)
    net.train()
    for i, (X, y) in enumerate(train_iter):
        timer.start()
        optimizer.zero_grad()
        X, y = X.to(device), y.to(device)
        y_hat = net(X)
        l = loss(y_hat, y)
        l.backward()
        optimizer.step()
        with torch.no_grad():
            # 不更新梯度计算
            # 1 * X.shape[0] 计算总的损失    d2l.accuracy(y_hat, y)计算准确率
            metric.add(1 * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
        timer.stop()
    #计算平均损失
    train_l = metric[0] / metric[2]
    #计算平均准确率
    train_acc = metric[1] / metric[2]
    if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
        animator.add(epoch + (i + 1) / num_batches,
                      (train_l, train_acc, None))
    test_acc = evaluate_accuracy_gpu(net, test_iter)
    animator.add(epoch + 1, (None, None, test_acc))
print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
      f'test acc {test_acc:.3f}')
print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
      f'on {str(device)}')
```

7.现代卷积神经网络

VGG

```
##VGG
import torch
from torch import nn
from d2l import torch as d2l

import os
os.chdir('/home/jupyter-zhangwenkk/data/')

def VGG_block(num_conv,in_channel,out_channel):
    layers = []
    for _ in range(num_conv):
        layers.append(nn.Conv2d(in_channel,out_channel,kernel_size=3,padding=1))
        layers.append(nn.ReLU())
        in_channel = out_channel
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))

    return nn.Sequential(*layers)

conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))

def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    for (num_convs,out_channels) in conv_arch:
        conv_blks.append(VGG_block(num_convs,in_channels,out_channels))
        in_channels = out_channels

    return nn.Sequential(*conv_blks,nn.Flatten(),
                        # 全连接层部分
                        nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(),
nn.Dropout(0.5),
                        nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
                        nn.Linear(4096, 10))

net = vgg(conv_arch)

X = torch.randn(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.__class__.__name__, 'output shape:\t',X.shape)
```



```

ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)

lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())

```

BN

手写BN

```

import torch
from torch import nn
from d2l import torch as d2l

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # 通过is_grad_enabled来判断当前模式是训练模式还是预测模式
    if not torch.is_grad_enabled():
        # 如果是在预测模式下，直接使用传入的移动平均所得的均值和方差
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # 使用全连接层的情况，计算特征维上的均值和方差
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # 使用二维卷积层的情况，计算通道维上 (axis=1) 的均值和方差。
            # 这里我们需要保持x的形状以便后面可以做广播运算
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
            # 训练模式下，用当前的均值和方差做标准化
            X_hat = (X - mean) / torch.sqrt(var + eps)
            # 更新移动平均的均值和方差
            moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
            moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta
    # 缩放和移位
    return Y, moving_mean.data, moving_var.data

```

目标检测

RCNN

先提取图片的特征 输出坐标和特征，把特征送到SVM输出类别

要理解的概念(面试)

梯度下降

<https://zhuanlan.zhihu.com/p/261375491>

- 什么是梯度，方程求偏导得到的式子就是梯度，把对应的x值放进去就能得到x点的斜率，这样就可以知道要减少值时前进的方向
- 使用所有样本来更新梯度

随机梯度下降

SGD的核心思想是在每次迭代中随机选择一个样本（或一小批样本）来估计梯度，而不是使用整个数据集。这样做的优点是计算效率高，尤其是当数据集很大时。SGD也能够逃离局部最小值，因为随机性引入了一定的噪声，有助于模型探索更多的参数空间。

这个随机的梯度如何选择

- 使用一个样本来更新梯度

二、随机梯度下降算法^Q

随机梯度下降 (Stochastic Gradient Descent, SGD) 是梯度下降算法的一种变种。与批量梯度下降不同的是, SGD 不是在每一次迭代中都使用全部训练样本来更新参数, 而是随机地选择一个样本来计算梯度并更新参数。其更新规则如下:

$$\theta_{t+1} = \theta_t - \alpha \nabla J(\theta_t; x_i, y_i)$$

其中, (x_i, y_i) 是随机选择的一个训练样本, $\nabla J(\theta_t; x_i, y_i)$ 是损失函数 $J(\theta)$ 在参数 θ_t 处关于样本 (x_i, y_i) 的梯度。

小批量随机梯度下降

<https://zhuanlan.zhihu.com/p/72929546>

- 使用一些样本来更新梯度

令 $f(\mathbf{x}; \theta)$ 表示一个深度神经网络, θ 为网络参数, 在使用小批量梯度下降进行优化时, 每次选取 K 个训练样本 $\mathcal{S}_t = \{(\mathbf{x}^{(k)}, \mathbf{y}^{(k)})\}_{k=1}^K$. 第 t 次迭代 (Iteration)

<https://nndl.github.io/>

7.2 优化算法

2020 年 1 月 3 日

时损失函数关于参数 θ 的偏导数为

$$\mathbf{g}_t(\theta) = \frac{1}{K} \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{S}_t} \frac{\partial \mathcal{L}(\mathbf{y}, f(\mathbf{x}; \theta))}{\partial \theta}, \quad (7.1)$$

其中 $\mathcal{L}(\cdot)$ 为可微分的损失函数, K 称为**批量大小** (Batch Size).

第 t 次更新的梯度 \mathbf{g}_t 定义为

$$\mathbf{g}_t \triangleq \mathbf{g}_t(\theta_{t-1}). \quad (7.2)$$

使用梯度下降来更新参数,

$$\theta_t \leftarrow \theta_{t-1} - \alpha \mathbf{g}_t, \quad (7.3)$$

这里的损失了正则化项. 则化的损失第 7.7.1 节.

批量大小的选择

批量大小不影响随机梯度的期望，但是会影响随机梯度的方差. 批量大小越大，随机梯度的方差越小，引入的噪声也越小，训练也越稳定，因此可以设置较大的学习率. 而批量大小较小时，需要设置较小的学习率，否则模型会不收敛. 学习率通常要随着批量大小的增大而相应地增大. 一个简单有效的方法是线性缩放规则（Linear Scaling Rule）

<https://cloud.baidu.com/article/3252849>

首先，学习率直接影响模型的收敛速度。如果学习率设置得过大，模型可能会在最优解附近反复震荡，无法收敛；如果学习率设置得过小，模型的训练速度可能会非常慢。因此，我们需要根据模型的训练情况和学习任务的特点，动态地调整学习率。例如，在模型训练的初期，我们可以使用较大的学习率，以加快模型的收敛速度；在模型训练的后期，我们可以逐渐减小学习率，以确保模型能够收敛到最优解。

其次，批量大小会影响模型的泛化能力。一般来说，批量大小越大，模型的训练稳定性越好，但泛化能力可能会下降。这是因为大的批量大小减少了模型在训练过程中遇到的随机误差，使得模型更容易过拟合训练数据。相反，小的批量大小可以引入更多的随机误差，提高模型的泛化能力。但这也可能导致模型的训练过程不稳定，需要更多的训练时间和更精细的调参。

为什么batchsize越大，随机误差越小（梯度的方差越小）？

- 在小批次（mini-batch）梯度下降法中，每次迭代都会计算一个批次样本的平均梯度。批次大小越大，这个平均梯度就越接近于整个训练集的真实梯度。 - 较大的批次包含更多的样本，这使得计算出的梯度能够更好地代表数据集的总体趋势，从而减少了随机误差

学习率衰减算法

学习率是神经网络优化时的重要超参数. 在梯度下降法中，学习率 α 的取值非常关键，如果过大就不会收敛，如果过小则收敛速度太慢. 常用的学习率调整方法包括学习率衰减、学习率预热、周期性学习率调整以及一些自适应调整学习率的方法

学习率一开始要大，保证收敛速度

后期学习率要变小，避免来回震荡

学习率衰减的常见方法

- 分段常数衰减
- 逆时衰减
- 指数衰减
- 自然指数衰减
- 余弦衰减
- 学习率衰减往往与衰减率，epoch，初始学习率有关

学习率预热

- 由于参数是随机初始化的，梯度往往也比较大，再加上比较大的初始学习率，会使得训练不稳定
- 为了提高训练稳定性，我们可以在最初几轮迭代时，采用比较小的学习率，等梯度下降到一定程度后再恢复到初始的学习率，这种方法称为学习率预热（Learning Rate Warmup）

周期性学习率调整

- 为了使得梯度下降法能够逃离局部最小值或鞍点，一种经验性的方式是在训练过程中周期性地增大学习率。虽然增大学习率可能短期内有损网络的收敛稳定性，但从长期来看有助于找到更好的局部最优解

循环学习率

- 让学习率在一个区间内周期性地增大和缩小

自适应学习率算法

AdaGrad

- 通过计算梯度平方的累计值来调整学习率

RMSprop 算法

- 计算每次迭代梯度 g_t 平方的指数衰减移动平均，

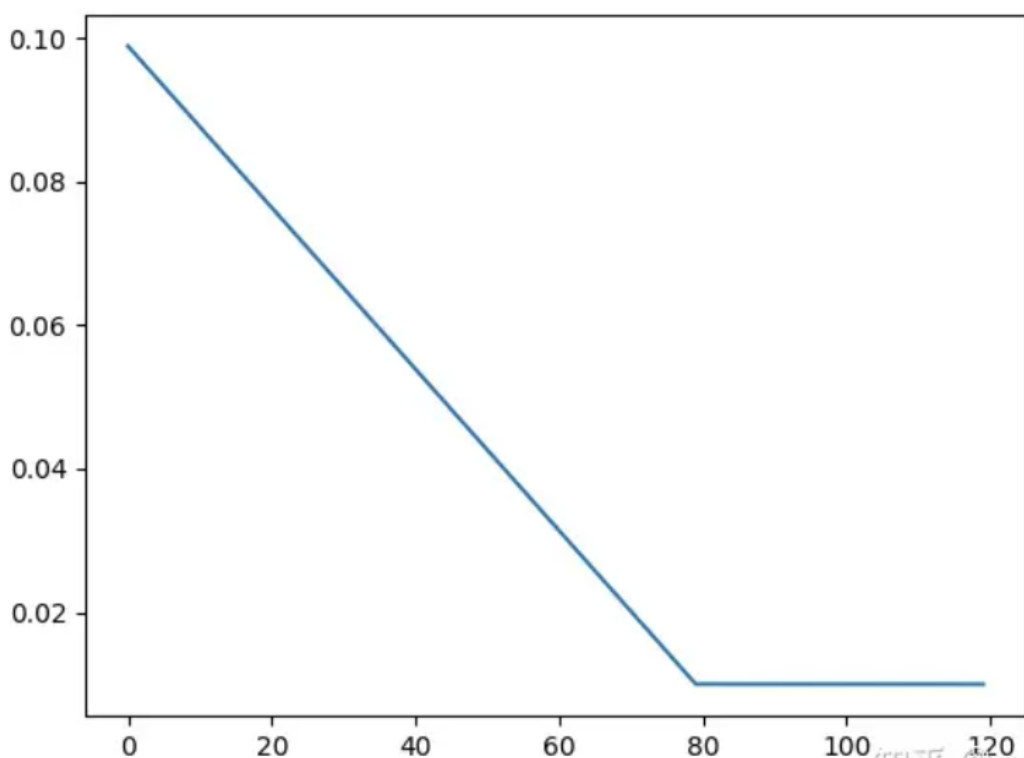
学习率调整策略

- linearLR
 - 线性学习率
 - 预先定义起始学习率，结束学习率，迭代次数。从起始学习率开始经过定义的迭代次数达到结束学习率

4. LinearLR

LinearLR是线性学习率，给定起始factor和最终的factor，LinearLR会在中间阶段做线性插值，比如学习率为0.1，起始factor为1，最终的factor为0.1，那么第0次迭代，学习率将为0.1，最终轮学习率为0.01。下面设置的总轮数total_iters为80,所以超过80时，学习率恒为0.01。

```
scheduler=lr_scheduler.LinearLR(optimizer,start_factor=1,end_factor=0.1,total_iters=80)
```



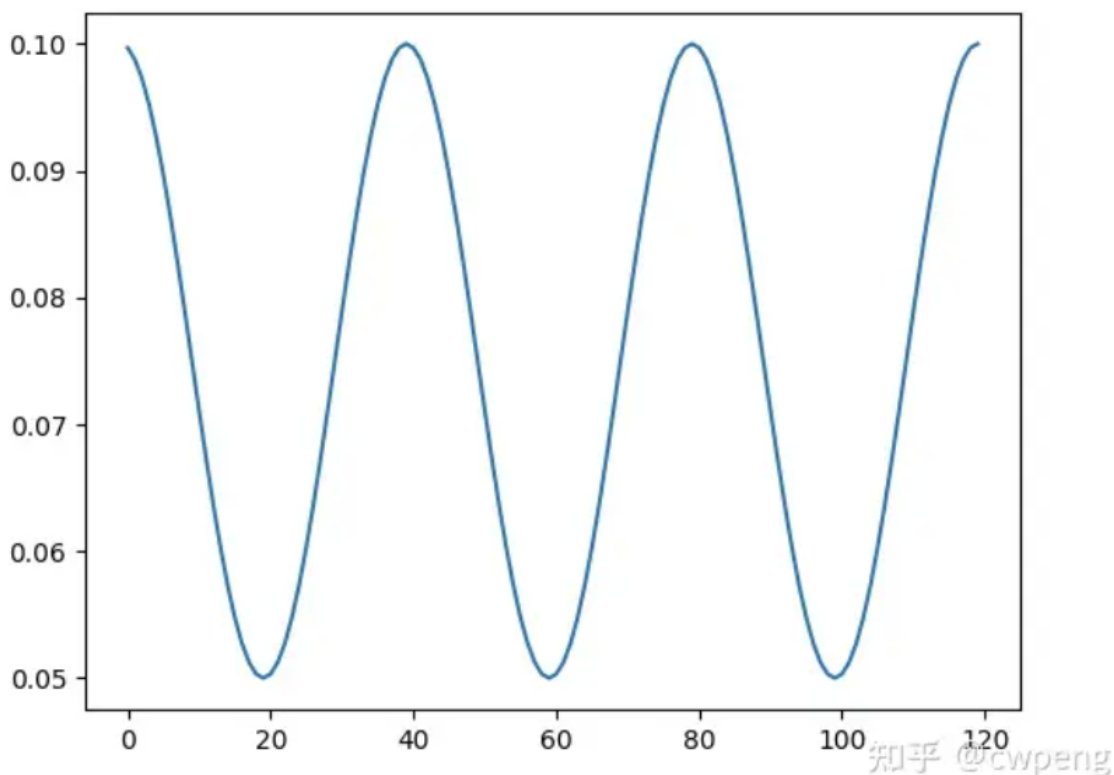
知乎 @cwpeng

- CosineAnnealingLR
 - 余弦退火学习率
 - 定义初始学习率，结束学习率，定义学习率周期。在一个后期内下降，然后一个周期内上升

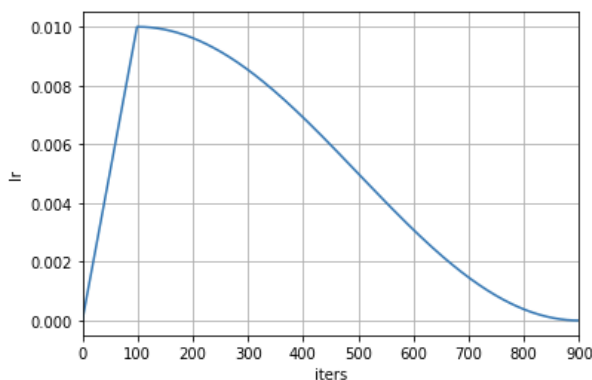
7. CosineAnnealingLR

CosineAnnealingLR是余弦退火学习率， T_{max} 是周期的一半，最大学习率在optimizer中指定，最小学习率为 η_{min} 。这里同样能够帮助逃离鞍点。值得注意的是最大学习率不宜太大，否则loss可能出现和学习率相似周期的上下剧烈波动。

```
scheduler=lr_scheduler.CosineAnnealingLR(optimizer,T_max=20,eta_min=0.05)
```



- 组合两个学习率，达到学习率预热的目的



上述例子表示在训练的前 100 次迭代时使用线性的学习率预热，然后在第 100 到第 900 次迭代时使用周期为 800 的余弦退火学习率调度器使学习率按照余弦函数逐渐下降为 0。

我们可以组合任意多个调度器，既可以使用 MMEngine 中已经支持的调度器，也可以实现自定义的调度器。如果相邻两个调度器的生效区间没有紧邻，而是有一段区间没有被覆盖，那么这段区间的学习率维持不变。而如果两个调度器的生效区间发生了重叠，则对多组调度器叠加使用，学习率的调整会按照调度器配置文件中的顺序触发（行为与 PyTorch 中 `ChainedScheduler` 一致）。在一般情况下，我们推荐用户在训练的不同阶段使用不同的学习率调度策略来避免调度器的生效区间发生重叠。如果确实需要将两个调度器叠加使用，则需要十分小心，避免学习率的调整与预期不符。

动量算法

Adam

- 计算梯度平方的指数加权平均，计算梯度 的指数加权平均

梯度截断

- 梯度的模小于或大于这个区间时就进行截断.避免梯度消失(爆炸)

参数初始化

在第一遍前向计算时，如果参数都为 0，所有的隐层神经元的激活值都相同. 这样会导致深层神经元没有区分性. 这种现象也称为对称权重现象.

我们要避免这个现象，对每个参数进行随机初始化

- 如果初始化参数太小，会导致网络梯度消失，如果过大会导致梯度爆炸（根据使用了哪一个激活函数来定义，不同的激活函数造成的后果可能不一样）

常用的初始化方法

- 高斯分布初始化

- 参数从一个固定均值（比如 0）和固定方差（比如 0.01）的高斯分布进行随机初始化。
- 均匀分布初始化
 - 在一个给定的区间 $[-r, r]$ 内采用均匀分布来初始化参数. 超参数 r 的设置可以按神经元的连接数量进行自适应的调整
- Xavier 初始化
 - 根据每层的神经元数量来自动计算初始化参数的方差.

数据预处理

- 标准归一化处理
- 白化处理
 - 用来降低输入数据特征之间的冗余性. 输入数据经过白化处理后，特征之间相关性较低，并且所有特征具有相同的方差.
 - 白化的一个主要实现方式是使用主成分分析（Principal Component Analysis, PCA）方法去除掉各个成分之间的相关性

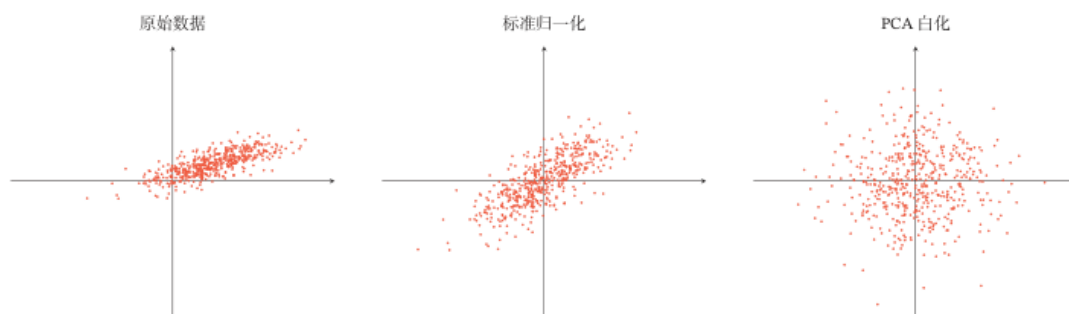


图 7.10 标准归一化和 PCA 白化

超参数优化

- 网格搜索
 - 通过尝试所有超参数的组合来寻址合适一组超参数配置的方法
- 随机搜索
 - 一种在实践中比较有效的改进方法是对超参数进行随机组合，然后选取一个性能最好的配置

softmax回归

下面给出Softmax函数的定义（以第*i*个节点输出为例）：

$Softmax(z_i) = \frac{e^{z_i}}{\sum_{c=1}^C e^{z_c}}$ ，其中 z_i 为第*i*个节点的输出值， C 为输出节点的个数，即分类的类别个数。通过Softmax函数就可以将多分类的输出值转换为范围在[0, 1]和为1的概率分布。



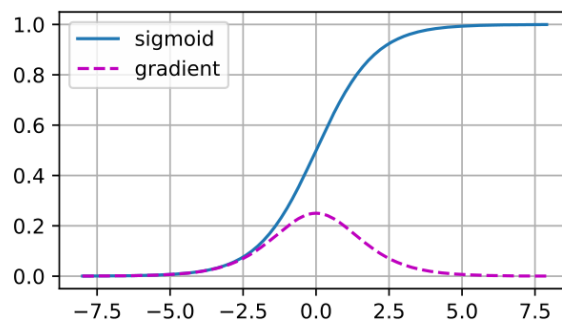
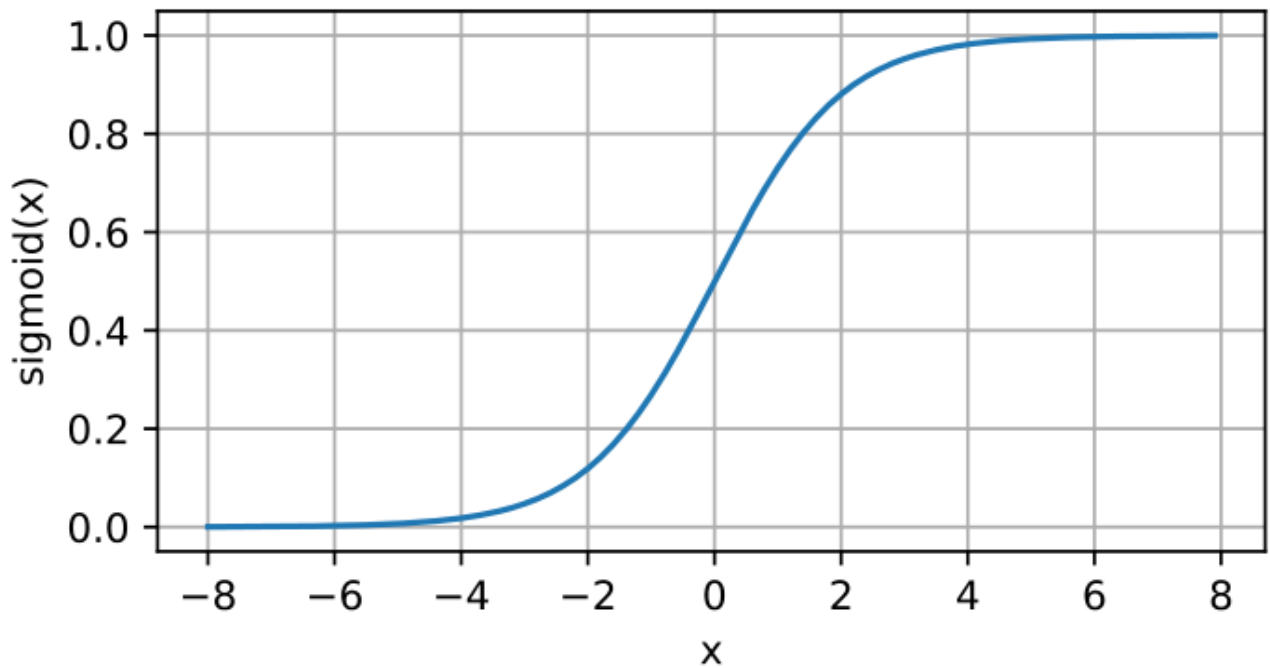
softmax要配合交叉熵损失函数同时实现

- 这里需要注意一下，当使用Softmax函数作为输出节点的激活函数的时候，一般使用交叉熵作为损失函数。由于Softmax函数的数值计算过程中，很容易因为输出节点的输出值比较大而发生数值溢出的现象，在计算交叉熵的时候也可能会出现数值溢出的问题。

激活函数

sigmoid

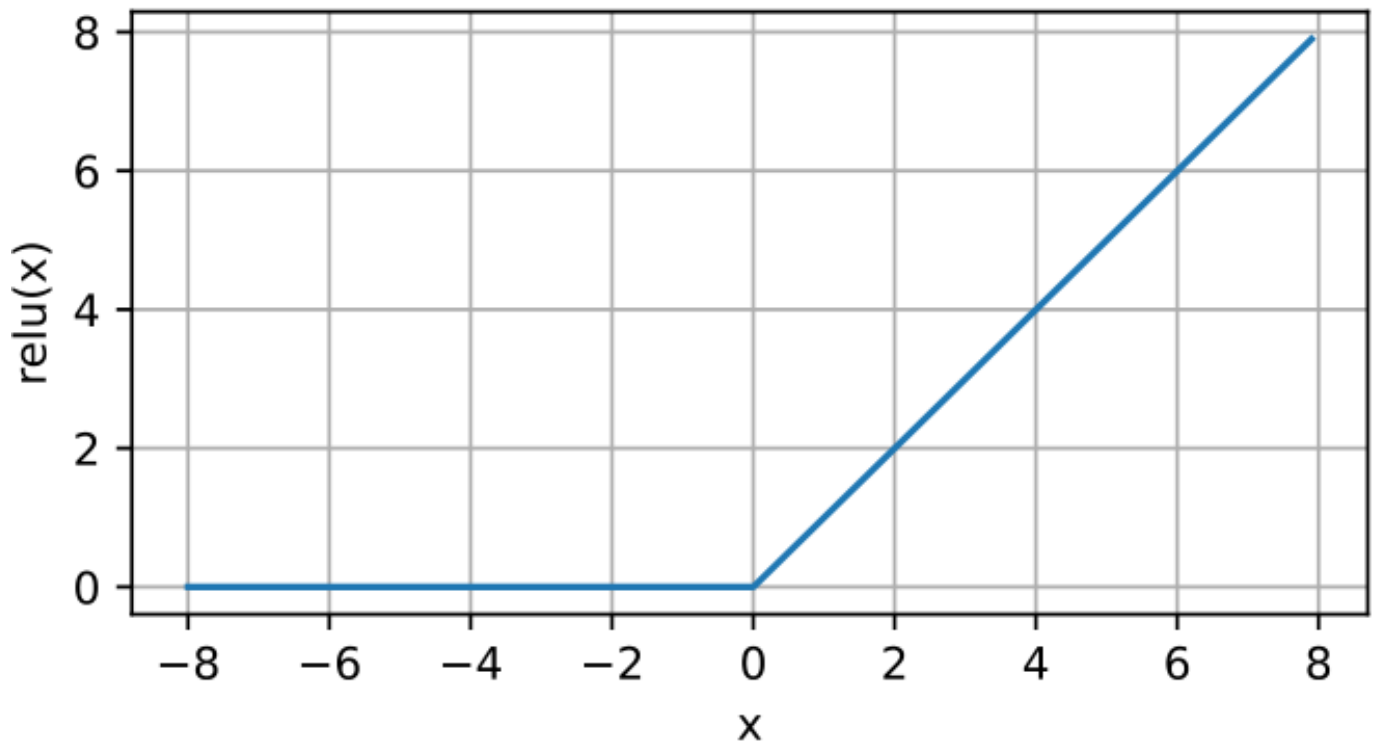
- sigmoid通常称为挤压函数（squashing function）：它将范围 $(-\infty, \infty)$ 中的任意输入压缩到区间 $(0, 1)$ 中的某个值
- $sigmoid(x) = \frac{1}{1+e^{-x}}$
-



正如图，当sigmoid函数的输入很大或是很小时，它的梯度都会消失。此外，当反向传播通过许多层时，除非我们在刚刚好的地方，这些地方sigmoid函数的输入接近于零，否则整个乘积的梯度可能会消失。当我们的网络有很多层时，除非我们很小心，否则在某一层可能会切断梯度。事实上，这个问题曾经困扰着深度网络的训练。因此，更稳定的ReLU系列函数已经成为从业者的默认选择（虽然在神经科学的角度看起来不太合理）。

sigmoid函数的缺点，取值被映射到0-1范围内，所以会出现梯度消失的问题

- ReLU
 - $\text{ReLU}(x) = \max(x, 0)$
 - ReLU激活函数缓解了梯度消失问题，这样可以加速收敛



ReLU的好处

<https://zhuanlan.zhihu.com/p/390655512>

- 能够避免反向传播过程中的梯度消失、屏蔽负值、防止梯度饱和

ReLU的缺点

- 当参数取值为负数时，ReLU激活函数会输出0，
-

$$w' = w - \eta \Delta w$$

上式是神经网络权重更新的公式，其中 η 表示学习率， Δw 表示通过求导得到的当前参数的梯度（一般为正值）当学习率过大时，会导致 $\eta \Delta w$ 这一项很大，当 $\eta \Delta w$ 大于 w 时，更新后的 w' 就会变为负值；

当权重参数变为负值时，输入网络的正值会和权重相乘后也会变为负值，负值通过relu后就会输出0；如果 w 在后期有机会被更新为正值也不会出现大问题，但是当relu函数输出值为0时，relu的导数也为0，因此会导致后边 Δw 一直为0，进而导致 w 一直不会被更新，因此会导致这个神经元永久性死亡（一直输出0）

优点 采用 ReLU 的神经元只需要进行加、乘和比较的操作，计算上更加高效。ReLU 函数被认为有生物上的解释性，比如**单侧抑制**、**宽兴奋边界**（即兴奋程度也可以非常高）。在生物神经网络中，同时处于兴奋状态的神经元非常稀疏。人脑中在同一时刻大概只有 1% ~ 4% 的神经元处于活跃状态。Sigmoid 型激活函数会导致一个非稀疏的神经网络，而 ReLU 却具有很好的稀疏性，大约 50% 的神经元会处于激活状态。

在优化方面，相比于 Sigmoid 型函数的两端饱和，ReLU 函数为左饱和函数，且在 $x > 0$ 时导数为 1，在一定程度上缓解了神经网络的梯度消失问题，加速梯度下降的收敛速度。

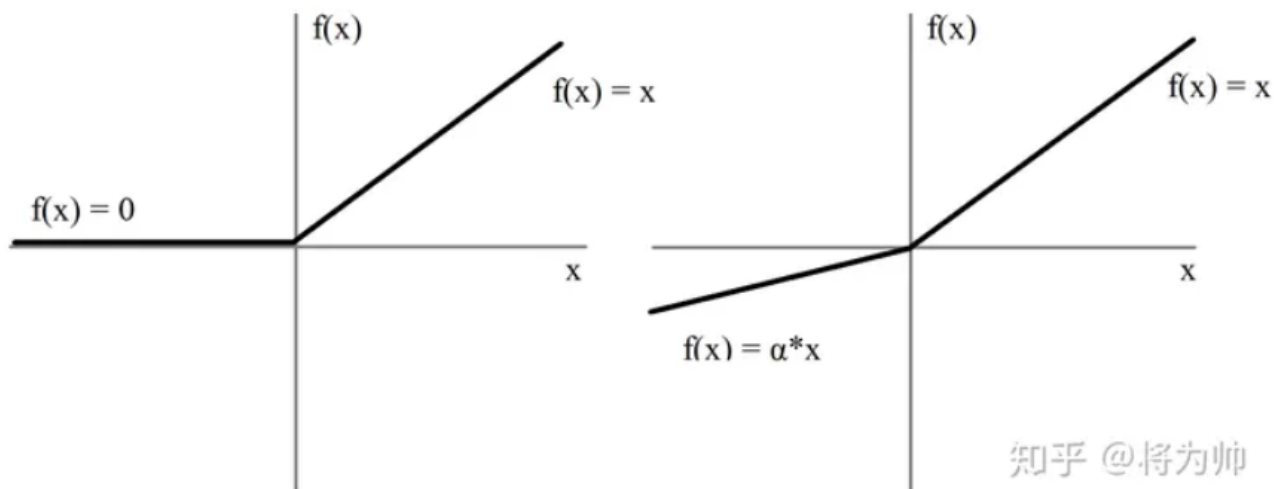
缺点 ReLU 函数的输出是非零中心化的，给后一层的神经网络引入**偏置偏移**，会影响梯度下降的效率。此外，ReLU 神经元在训练时比较容易“死亡”。在训练时，如果参数在一次不恰当的更新后，第一个隐藏层中的某个 ReLU 神经元在所有的训练数据上都不能被激活，那么这个神经元自身参数的梯度永远都会是 0，在以后的训练过程中永远不能被激活。这种现象称为**死亡 ReLU 问题**（Dying ReLU Problem），并且也有可能发生在其他隐藏层。

- LeakyReLU

- 在小于 0 的部分不像 ReLU 是 0，而是一个负数的随机梯度，a 的值一般为 0.01

3.1 LeakyReLU可以解决神经元“死亡”问题

LeakyReLU的提出就是为了解决神经元“死亡”问题，LeakyReLU与ReLU很相似，仅在输入小于0的部分有差别，ReLU输入小于0的部分值都为0，而LeakyReLU输入小于0的部分，值为负，且有微小的梯度。函数图像如下图：



实际中，LeakyReLU的 α 取值一般为0.01。使用LeakyReLU的好处就是：在反向传播过程中，对于LeakyReLU激活函数输入小于零的部分，也可以计算得到梯度(而不是像ReLU一样值为0)，这样就避免了上述梯度方向锯齿问题。

- tanh函数

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

-

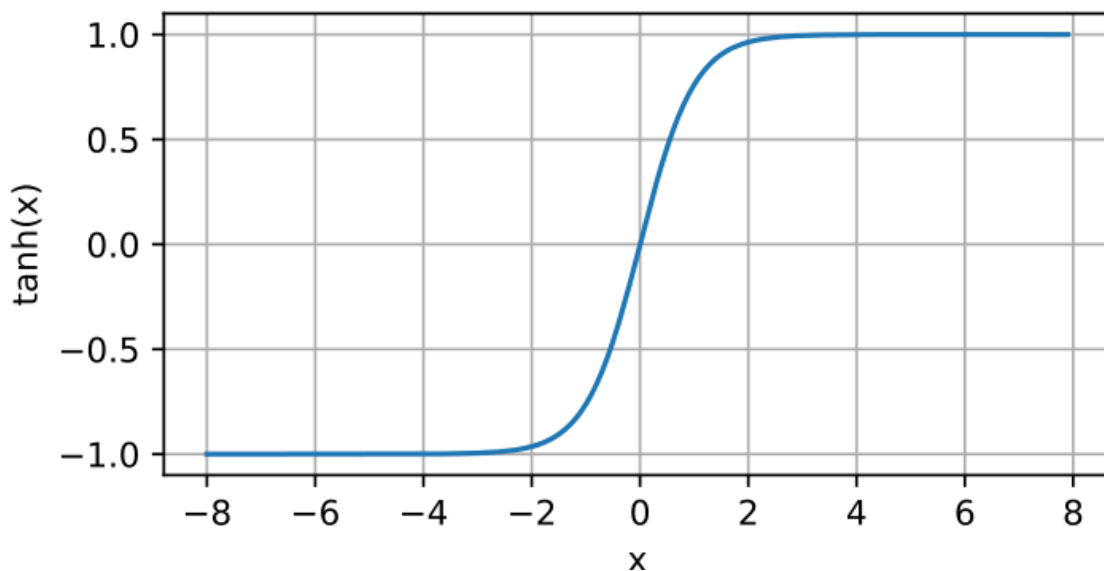
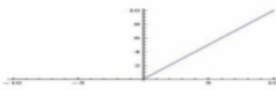



表5-2 激活函数各种属性

名称	表达式	导数	图形
sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	
tanh	$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$	$f'(x) = 1 - (f(x))^2$	

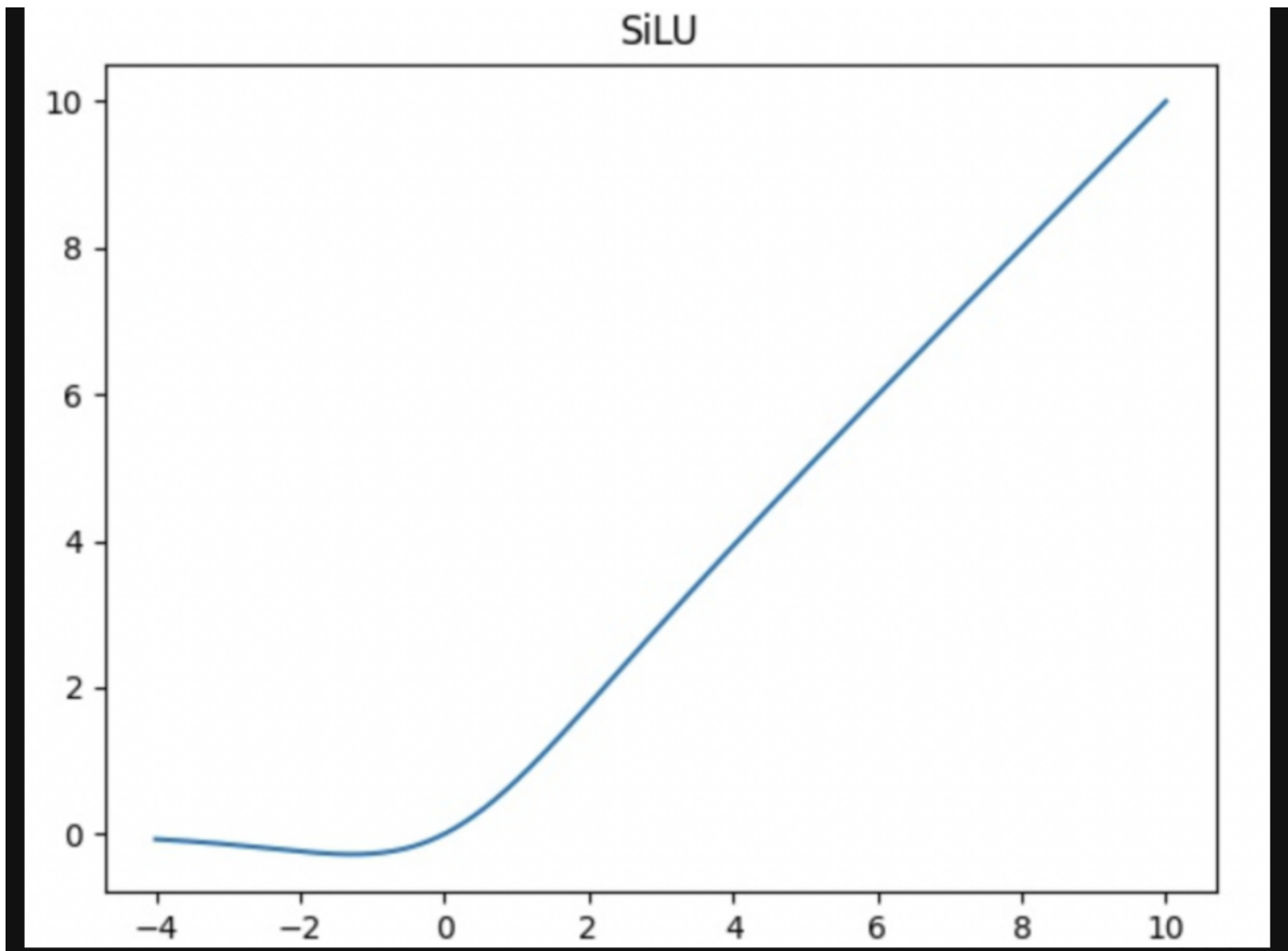
(续)

名称	表达式	导数	图形
relu	$f(x) = \max(0, x)$	$f'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$	
LeakyReLU	$f(x) = \max(ax, x)$	$f'(x) = \begin{cases} 1 & x \geq 0 \\ \alpha & x < 0 \end{cases}$	
softmax	$\sigma_i(z) = \frac{e^{z_i}}{\sum_{j=1}^m e^{z_j}}$		

SiLu(swish)激活函数

$f(x) = x * \text{sigmoid}(x)$ (yolov5)

它既有 ReLU（Rectified Linear Unit）激活函数的一些优点（例如，能够缓解梯度消失问题），又能解决 ReLU 函数的一些缺点（例如，ReLU 函数不是零中心的，且在负数部分的梯度为零）



激活函数的选择

如果搭建的神经网络层数不多，选择sigmoid、tanh、relu、softmax都可以；而如果搭建的网络层次较多，那就需要小心，选择不当就可导致梯度消失问题。此时一般不宜选择**sigmoid**、**tanh**激活函数，因它们的导数都小于1，所以，搭建比较深的神经网络时，一般使用relu激活函数

损失函数的选择

什么是损失函数？

损失函数（Loss Function）在机器学习中非常重要，因为训练模型的过程实际就是优化损失函数的过程。损失函数用来衡量模型的好坏，损失函数越小说明模型和参数越符合训练样本。

常用的损失函数有两种

- 交叉熵(Cross Entropy) softmax激活函数

- https://www.zhihu.com/tardis/zm/art/35709485?source_id=1005

- 分类问题 (交叉熵反应的两个概率分布的距离)
 - 多分类
 - softmax激活函数+交叉熵损失函数
 - 二分类 (逻辑回归损失 (Logistic Loss))

1.3 Cross Entropy Loss Function (交叉熵损失函数)

1.3.1 表达式

(1) 二分类

在二分的情况下，模型最后需要预测的结果只有两种情况，对于每个类别我们的预测得到的概率为 p 和 $1 - p$ ，此时表达式为 (\log 的底数是 e)：

$$L = \frac{1}{N} \sum_i L_i = \frac{1}{N} \sum_i -[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)]$$

其中：

- y_i —— 表示样本 i 的label, 正类为 1，负类为 0
- p_i —— 表示样本 i 预测为正类的概率

(2) 多分类

多分类的情况实际上就是对二分类的扩展：

$$L = \frac{1}{N} \sum_i L_i = -\frac{1}{N} \sum_i \sum_{c=1}^M y_{ic} \log(p_{ic})$$

为什么交叉熵可以计算类别的损失

<https://zhuanlan.zhihu.com/p/124309304>

- 交叉熵 (Cross Entropy) 用于衡量一个概率分布与另一个概率分布之间的距离。交叉熵是机器学习和深度学习中常常用到的一个概念。在分类问题中，我们通常有一个真实的概率分布 (通常是专家或训练数据的分布)，以及一个模型生成的概率分布，交叉熵可以衡量这两个分布之间的距离。模型训练时，通过最小化交叉熵损失函数，我们可以使模型预测值的概率分布逐步接近真实的概率分布。
- 神经网络的类别输出往往是输出类别的概率值，这时候要计算输出概率值与标签概率之间的误差。这个要追溯到信息论，香浓熵。当标签的概率分布为 P ，模型预测的概率为 Q ，我们要减小 P ， Q 之间的误差，就要对 PQ 的交叉熵做最小化处理，这时候就要最小化 PQ 的交叉熵，让他们的概率分布更接近

- 交叉熵

通过上面的介绍，相信聪明的同学已经发现可以通过最小化相对熵来用分布Q逼近分布P(目标概率分布)。首先我们对相对熵公式进行变形：

$$\begin{aligned}D_{KL}(P||Q) &= \sum_{i=1}^n P(x_i) \log \frac{P(x_i)}{Q(x_i)} \\&= \sum_{i=1}^n P(x_i) \log P(x_i) - \sum_{i=1}^n P(x_i) \log Q(x_i) \\&= -H(P) + H(P, Q)\end{aligned}$$

这里的 $H(P, Q)$ 就是交叉熵(cross-entropy)，它的表达式为：

$$H(P, Q) = - \sum_{i=1}^n P(x_i) \log Q(x_i)$$

对于确定的概率分布P，它的香农熵 $H(P)$ 是一个常数，所以要对相对熵 $D_{KL}(P||Q)$ 进行最小化，只需对交叉熵 $H(P, Q)$ 做最小化处理即可。

通过最小化交叉熵，就可以得到分布P的近似分布，这也是为什么可以用交叉熵作为网络的损失函数。

- 均方误差 (Mean squared error, MSE)
- https://blog.csdn.net/Xiaobai_rabbit0/article/details/111032136

- 回归问题（回归问题预测的不是类别，而是一个任意实数）

1.1、均方误差MSE

均方误差 (Mean Square Error, MSE) 是回归损失函数中最常用的误差，它是预测值 $f(x)$ 与目标值 y 之间差值平方和的均值，其公式如下所示：

$$MSE = \frac{\sum_{i=1}^n (f(x) - y)^2}{n}$$

优化器

5.6.1 传统梯度优化的不足

传统梯度更新算法为最常见、最简单的一种参数更新策略。其基本思想是：先设定一个学习率 λ ，参数沿梯度的反方向移动。假设需更新的参数为 θ ，梯度为 g ，则其更新策略可表示为：

$$\theta \leftarrow \theta - \lambda g \quad (5-12)$$

这种梯度更新算法简洁，当学习率取值恰当时，可以收敛到全面最优点（凸函数）或局部最优点（非凸函数）。

- 传统梯度下降
- 动量算法 + 梯度下降
 - NAG

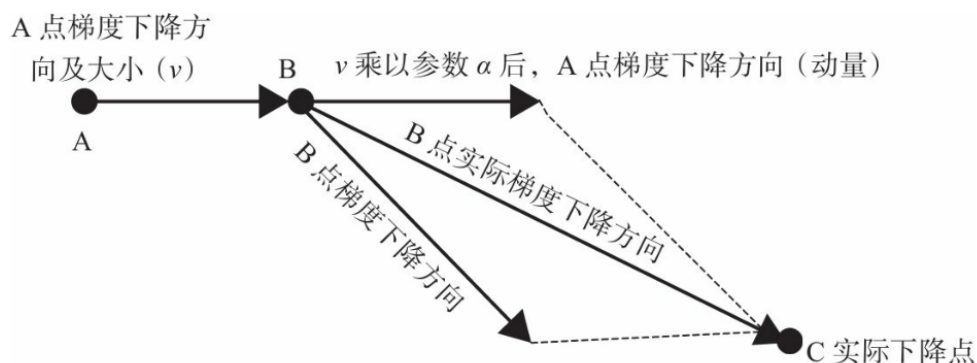


图5-15 动量算法示意图

由图5-15可知，动量算法每下降一步都是由前面下降方向的一个累积和当前点的梯度方向组合而成。含动量的随机梯度下降法，其算法伪代码如下：

- 学习率算法
 - AdaGrad（自动更新学习率）
 - 为了更好地驾驭这个超参数，人们想出来多种自适应优化算法，使用自适应优化算法，学习率不再是一个固定不变值，它会根据不同情况自动调整来适应相应的情况
 - RMSProp算法
 - 针对梯度平方和累计越来越大的问题，RMSProp指数加权的移动平均代替梯度平方和
 - Adam
 - Adam（Adaptive Moment Estimation）本质上是带有动量项的RMSprop，它利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率
 - Adam 通过计算梯度的一阶矩估计和二阶矩估计的移动平均值，调整每个参数的学习率

算法的综合使用

- 有时可以考虑综合使用这些优化算法，如采用先使用Adam，然后使用SGD的优化方法，这个想法，实际上是由于在训练的早期阶段SGD对参数调整和初始化非常敏感。因此，我们可以通过先使用Adam优化算法来进行训练，这将大大地节省训练时间，且不必担心初始化和参数调整，一旦用Adam训练获得较好的参数后，就可以切换到SGD+动量优化，以达到最佳性能

adamW

- 优化器是用来优化梯度的，通过距离范数来优化下一次下降的梯度值

2.7 AdamW

在AdamW提出之前，Adam算法已经被广泛应用于深度学习模型训练中。但是人们发现，理论上更优的Adam算法，有时表现并不如SGD momentum好，尤其是在模型泛化性上。

我们知道，L2范数（也叫权重衰减weight decay）有助于提高模型的泛化性能。

但是AdamW的作者证明，Adam算法弱化了L2范数的作用，所以导致了用Adam算法训练出来的模型泛化能力较弱。具体来说，在Adam中（上图的算法），权重衰减的梯度是直接加在 g_t 上的，这就导致权重衰减的梯度也会随着 g_t 去除以分母。当梯度的平方和累积过大时，权重衰减的作用就会被大大削弱（红框部分）。

$$x_t \leftarrow x_{t-1} - \alpha \frac{\beta_1 m_{t-1} + (1 - \beta_1)(\nabla f_t + wx_{t-1})}{\sqrt{v_t} + \epsilon}$$

欠拟合（模型太简单，无法很好学习到数据集中的特征，）

过拟合（模型太复杂，学习数据集中的特征学过头了，数据集太简单）



ChatGPT

过拟合 (Overfitting) 和欠拟合 (Underfitting) 是深度学习中两种常见的模型训练问题，它们都可能导致模型在实际应用中表现不佳。

1.

过拟合 (Overfitting) :

- 定义：过拟合指的是模型在训练数据上表现很好，但在测试数据（即未见过的数据）上表现较差的现象。
- 原因：通常是因为模型过于复杂，以至于学习了训练数据的噪声和细节，而忽略了数据中的一般趋势和规律。
- 表现：在训练数据上的误差很低，但在测试数据上的误差较高。
- 解决方法：
 - 增加训练数据量，以减少模型对特定数据的依赖。
 - 使用正则化技术，如 L1、L2 正则化，以惩罚模型复杂度。
 - 采用 Dropout、Batch Normalization 等技术，减少模型的复杂度和提高泛化能力。
 - 使用交叉验证等技术，优化模型参数。
 - 减小模型的复杂度，如减少神经网络的层数、节点数等。

2.

欠拟合 (Underfitting) :

- 定义：欠拟合指的是模型在训练数据和测试数据上都表现不佳的现象。
- 原因：通常是因为模型过于简单，无法捕捉数据中的复杂模式和趋势。
- 表现：在训练数据和测试数据上的误差都较高。
- 解决方法：
 - 增加模型复杂度，如增加神经网络的层数、节点数等。
 - 选择更复杂的模型结构，如使用更深的神经网络、更复杂的模型架构。
 - 添加更多的特征或进行特征工程，以提高模型的表现能力。
 - 增加训练时间，以使模型更好地学习数据中的模式和规律。

在实践中，解决过拟合和欠拟合问题往往需要结合经验和实践，针对具体问题采取适当的策略和技术来调整模型。

- 数据集太小容易过拟合
- 数据集太大容易欠拟合(模型无法很好学习数据集中的规律)

解决过拟合问题

- 权重衰减
 - 最常用方法是将权重向量的范数作为惩罚项加到最小化损失的问题中。权重衰减通过对模型的权重参数进行惩罚，使得模型倾向于学习简单的权重分布，而不是复杂的分布，从而让模型变得简单。
- dropout（暂退法）（丢弃法）
 - $f(x) = wx + b$ 当 $w=0$ 时，相当于这个神经元不作为输入了，训练过程中丢弃一些神经元，所以模型有变简单了
- 提前停止
 - 当验证集上的错误率不再下降，就停止迭代

梯度消失

- 参数更新过小，在每次更新时权重几乎不会移动，导致模型无法学习
- 用了不合适的激活函数

梯度爆炸

- 参数更新过大，破坏了模型的稳定收敛；
- 梯度的初始值太大，降低初始值，设置梯度阈值，强制在这个范围内
- 通过权重正则项惩罚来限制权重大小
- batch normalization 对每一层的输出规范为均值和方差一致的方法，消除权重参数放大缩小带来的影响，从而解决梯度消失和梯度爆炸（BN将输出从饱和区拉到非饱和区）

模型轻量化

- DW卷积的使用
- 点卷积代替全连接层

卷积的特性

- 平移不变性
 - 卷积过程中卷积核是平移的，但是输入特征的位置对应着输出特征的位置
- 局部性
 - 卷积核 探索输入图像中的局部区域，不在意图像中相隔较远的区域关系

输出HW的计算公式

输出高度:
$$\text{output_height} = \frac{\text{input_height} - \text{kernel_height} + 2 \times \text{padding}}{\text{stride}} + 1$$

汇聚层

- 池化层

批量归一化(batch normalization)

让数据变成均值为0，方差为1的分布，让数据变成独立同分布

这样做有什么好处

- 让模型更好的训练
 - 让特征的数值映射到一个区间分布。如果一个特征的分布趋向于两边，使用sigmoid激活函数激活时，可能要么是0 要么是1
- 减少训练和测试的误差（让训练和测试时，数据的分布一样）
 - 训练数据与测试数据的分布是相同的，这减少了模型的偏差风险。这样，模型在训练数据上的表现和在实际应用中的表现更加一致。

凸函数与非凸优化



ChatGPT

在深度学习中，几乎所有优化问题都是非凸的，这是因为深度学习模型通常具有大量的参数，导致优化目标函数具有非凸性质。下面我将解释这句话的含义：

1. 凸优化问题 vs. 非凸优化问题：

- 在凸优化问题中，目标函数是凸函数，这意味着函数图像上的任意两点之间的线段都在或者位于函数图像的上方。
- 而在非凸优化问题中，目标函数可能是非凸函数，即函数图像上存在凹陷或者山谷。

2. 深度学习中的优化问题：

- 在深度学习中，我们通常通过优化算法来调整神经网络的参数，使得模型在给定数据集上表现良好。
- 这个优化过程可以看作是一个优化目标函数的过程，而这个目标函数通常被称为损失函数或者代价函数。
- 深度学习模型往往拥有大量的参数，例如神经网络的权重和偏置，因此优化目标函数是一个高维函数。

3. 高维函数的非凸性：

- 高维函数的性质往往更加复杂，而在深度学习中，目标函数通常是一个高维函数，因为它依赖于许多参数。
- 当函数的维度增加时，非凸性变得更加普遍，这意味着函数图像上可能存在许多局部最小值或者鞍点，这使得优化问题变得更加困难。

4. 影响和应对：

- 非凸性使得优化问题变得更加复杂，可能导致优化算法陷入局部最小值或者鞍点，从而影响模型的训练效果。
- 为了应对非凸性，深度学习中通常使用的优化算法，例如随机梯度下降（SGD）的变体，具有一定的鲁棒性，能够在一定程度上避免陷入局部最小值。
- 此外，一些高级的优化算法，如Adam、Adagrad等，也被设计用于更有效地处理非凸优化问题，提高模型的训练效率和性能。

因此，深度学习中几乎所有的优化问题都是非凸的，这使得优化过程更具挑战性，需要使用专门设计的优化算法来应对这种复杂性。



特征表示形式

局部表示

分布式表示

分析为什么平方损失函数不适用于分类问题

平方损失函数是计算两个连续值之间的距离，而分类问题输出的是一个离散值（概率），标签也是一个离散值，没有连续的概念。如果使用平方损失函数计算预测值与标签之间的距离没有实际意义，预测值和标签两个向量之间的平方差这个值不能反应分类这个问题的优化程度。

多层感知机（MLP）

多层感知机MLP层往往通过Linear降为 -》激活函数-》Linear升维

为什么要降维

- 在原始的高维空间中，包含冗余信息和噪声信息，会在实际应用中引入误差，影响准确率；
- 而降维可以提取数据内部的本质结构，减少冗余信息和噪声信息造成的误差，提高应用中的精度。



将数据映射到低维度有助于降低噪声，主要原因有以下几点：

1. **去除冗余信息：**高维数据中可能包含许多冗余特征，这些特征对数据的主要结构贡献较小。在降维过程中，算法会识别和保留那些对数据主要结构有贡献的特征，去除对主要结构贡献小的特征，从而减少了冗余信息和噪声。
2. **平滑效应：**降维可以看作是一种数据平滑过程，高频噪声成分通常集中在某些特征或在高维空间中分散，降维过程中这些高频噪声成分可能会被削弱或去除，只保留了数据的低频成分，也就是数据的主要模式和结构。
3. **噪声累积效应：**在高维空间中，每一个特征的噪声可能会累积，导致总的噪声水平增高。而降维后，剩余的特征数减少，噪声的累积效应也随之减少。特征减少意味着每个特征中噪声的影响也被稀释了。
4. **降维方法的选择：**许多降维算法（如主成分分析（PCA）和t-SNE）是设计用来捕捉数据中重要的结构模式，这些模式往往是对数据的主要变化有贡献的部分。噪声通常是随机的，对这些主要变化的贡献很小，所以这些算法在降维时自然会倾向于忽略噪声。

为什么要激活函数

- 不使用激活函数，每一层输出都是上层输入的线性函数，无论神经网络有多少层，输出都是输入的线性组合。
- 使用激活函数，能够给神经元引入非线性因素，使得神经网络可以任意逼近任何非线性函数，这样神经网络就可以利用到更多的非线性模型中。

常用库

- torchvision.transforms
- torchvision.datasets
- torch.utils.data.DataLoader
- torch.nn as nn (特征提取模块 损失函数)
- torch.nn.functional as F (激活函数)
- torch.optim as optim

```
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5,
0.5), (0.5, 0.5, 0.5))])
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
download=False, transform=transform)

trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=False, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=4,
shuffle=False, num_workers=2)

dataiter = iter(trainloader)
images, labels = dataiter.next()


import torch.nn as nn
import torch.nn.functional as F
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
class CNNNet(nn.Module):
    def __init__(self):
        super(CNNNet, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=16, kernel_size=5, stride=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=36, kernel_size=3, stride=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(1296, 128)
```

```

        self.fc2 = nn.Linear(128,10)
    def forward(self,x):
        x=self.pool1(F.relu(self.conv1(x)))
        x=self.pool2(F.relu(self.conv2(x)))
        #print(x.shape)
        x=x.view(-1,36*6*6)
        x=F.relu(self.fc2(F.relu(self.fc1(x))))
        return x
net = CNNNet()
net=net.to(device)

import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# 训练模型
for epoch in range(10):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 获取训练数据
        inputs, labels = data
        inputs, labels = inputs.to(device), labels.to(device)
        # 权重参数梯度清零
        optimizer.zero_grad()
        # 正向及反向传播
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        # 显示损失值
        running_loss += loss.item()
    if i % 2000 == 1999: # print every 2000 mini-batches
        print('[%d, %5d] loss: %.3f' %(epoch + 1, i + 1, running_loss / 2000))
        running_loss = 0.0
        print('Finished Training')

# 测试模型
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images, labels = images.to(device), labels.to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

```

```
print('Accuracy of the network on the 10000 test images: %d %%' % (  
    100 * correct / total))
```