

暂退法 (Dropout)

暂退法和 通过基于权重的L2范数来正则化 都是用于处理优化问题中的过拟合问题。

二者的主要区别在于：

- 1、正则化通过在对参数中增加一项惩罚项来控制模型的复杂度，防止过拟合。其中，L2范数正则化是一种常用的方法，它会使权重较小，从而减少模型的复杂度。
- 2、暂退法则是一种全局优化算法，通过逐步减小随机扰动的程度，逐步搜索解空间，从而寻找全局最优解。在搜索过程中，暂退法也会尽量避免陷入局部最优解，从而减少过拟合的风险。

暂退法在前向传播过程中，计算每一内部层的同时注入噪声，这已经成为训练神经网络的常用技术。

这种方法之所以被称为暂退法，因为我们从表面上看是在训练过程中丢弃（dropout）一些神经元。在整个训练过程的每一次迭代中，标准暂退法包括在计算下一层之前将当前层中的一些节点置零。

关键的挑战就是如何注入这种噪声：在标准暂退法正则化中，通过保留的节点的分数进行规范化来消除每一层的偏差。

换言之，每个“中间活性值*h*”以 暂退概率*p* 由 随机变量*h*’替换，如下所示：

$$h' = \begin{cases} 0 & \text{概率为 } p \\ \frac{h}{1-p} & \text{其他情况} \end{cases}$$

根据此模型的设计，其期望值保持不变。即 E(*h*)=*h*

实现单层的暂退法函数

dropout_layer(X, dropout)函数：

以dropout的概率丢弃张量输入X中的元素

```
In [1]:
import torch
from torch import nn
from d2l import torch as d2l

def dropout_layer(X, dropout): # 以dropout的概率丢弃张量X中的元素
    # 输入张量X，dropout概率的dropout
    assert 0 < dropout <= 1 # assert语句检查dropout概率是否落在0和1之间（断言语句，如果不在这个范围内，就会触发一个AssertionError）
    # 在本情况中，所有元素都被丢弃
    if dropout == 1:
        return torch.zeros_like(X) # 返回一个和输入张量X相同形状的张量
    # 在本情况中，所有元素都被保留
    if dropout == 0:
        return X # 返回输入张量X本身，不需要进行任何dropout操作
    mask = (torch.rand(X.shape) > dropout).float()
    # 对于每个情况，我们生成了一个与输入张量X相同形状的掩码mask
    # 掩码中的每个元素都是0或1（大于dropout概率mask=1，小于dropout概率mask=0）
    # 这个掩码代表了哪些元素需要被保留，哪些元素需要丢弃。
    return mask * X / (1.0 - dropout) # 最后，我们将输入张量X和掩码相乘，然后除以（1-dropout）以保持期望值不变。
```

测试dropout_layer函数

将输入X通过暂退法操作，暂退概率分别为0、0.5和1。

```
In [2]:
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.))

tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  0.,  4.,  0.,  0.,  0., 10.,  0., 14.],
        [ 0., 18.,  0.,  0.,  0., 24., 28., 30.]])
tensor([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

从以上代码运行的结果可以看出：

当dropout概率为0时，所有元素都被保留。

当dropout概率为0.5时，一半的元素被丢弃。

当dropout概率为1时，所有元素都被丢弃，返回了全0张量。

定义模型参数

num_inputs: 输入层的神经元个数，即MNIST数据集集中每个图像的大小（28*28）=784。

num_outputs: 输出层的神经元个数，即分类的类别数。MNIST数据集集中有10个数字，因此为10。

num_hiddens1: 第一个隐藏层的神经元个数，为256。

num_hiddens2: 第二个隐藏层的神经元个数，也为256。

```
In [3]: num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
```

定义模型

将模型应用用于每个隐藏层的输出（在激活函数之后）

可以为每一层分别设置暂退概率；常见的技巧是在靠近输入的地方设置较低的暂退概率。

下面的模型将第一个和第二个隐藏层的暂退概率分别设置为0.2和0.5，并且暂退法只在训练期间有效。

```
In [4]: dropout1, dropout2 = 0.2, 0.5 # 第一个和第二个隐藏层的暂退概率分别设置为0.2和0.5

class Net(nn.Module):
    # 定义了一个名为Net的类，该类继承自nn.Module。
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2, is_training = True):
        """
        定义了四个输入参数：
        num_inputs: 输入层的神经元个数
        num_outputs: 输出层的神经元个数
        num_hiddens1: 第一个隐藏层的神经元个数
        num_hiddens2: 第二个隐藏层的神经元个数
        此外，is_training的布尔型参数：是否处于训练模式。
        """
        super(Net, self).__init__() # 调用Super()初始化nn.Module类

        # 添加变量的初始化
        self.num_inputs = num_inputs
        self.training = is_training

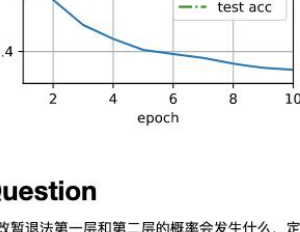
        # nn.Linear()定义三个全连接层
        self.lin1 = nn.Linear(num_inputs, num_hiddens1) # self.lin1连接输入层和第一个隐藏层
        self.lin2 = nn.Linear(num_hiddens1, num_hiddens2) # self.lin2连接第一个和第二个隐藏层
        self.lin3 = nn.Linear(num_hiddens2, num_outputs) # self.lin3连接第二个隐藏层和输出层
        self.relu = nn.ReLU() # 添加激活函数

    def forward(self, X):
        # forward函数是Net类中的前向传播函数，给定输入X，它返回模型的输出。
        H1 = self.relu(self.lin1(X.reshape((-1, self.num_inputs))))
        # 先将输入进行形状转换，使其变为2D张量（reshape第一维是batch size，第二维是num_inputs）
        # 再reshape后的张量作为输入传给self.lin1，进行第一次全连接计算，将结果作为self.relu的输入，进行ReLU激活函数操作
        # 得到第一个全连接层 H1
        if self.training == True: # 只有在训练模式时才使用dropout
            H1 = dropout_layer(H1, dropout1) # 在第一个全连接层之后添加一个dropout层
        H2 = self.relu(self.lin2(H1)) # 将 H1 作为self.lin2的输入，结果作为self.relu的输入进行ReLU激活函数操作
        # 得到第二个全连接层 H2
        if self.training == True:
            H2 = dropout_layer(H2, dropout2) # 在第二个全连接层之后添加一个dropout层
        out = self.lin3(H2) # 输出层
        return out # 返回模型的输出

# 创建一个Net实例
net = Net(num_inputs, num_outputs, num_hiddens1, num_hiddens2)
```

```
In [5]: num_epochs, lr, batch_size = 10, 0.5, 256

num_epochs: 训练的轮数，即对整个数据集进行几次迭代训练。
lr: 学习率，在训练时更新网络权重。
batch_size: 每次迭代训练时，从数据集中读取的样本数量。
loss = nn.CrossEntropyLoss(reduction='none') # 使用交叉熵损失函数
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size) # 训练集和测试集的数据迭代器（用于遍历数据集集中的批次数据，每次遍历返回一个批次数据）
trainer = torch.optim.SGD(net.parameters(), lr=lr) # 模型优化器，使用随机梯度下降算法。
# net.parameters() 返回一个包含模型所有参数的迭代器，可以用于更新模型的参数。
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer) # 训练模型
```



函数 train_ch3(net, train_iter, test_iter, loss, num_epochs, updater) 见ch3.6

简洁实现

对于深度学习框架的高级API，我们只需在每个全连接层之后添加一个Dropout层，将暂退概率作为唯一的参数传递给它构造函数。

在训练时，Dropout层将根据指定的暂退概率随机丢弃上一层的输出（相当于下一层的输入）。

在测试时，Dropout层仅连接神经网络。

```
In [6]: # 定义一个三层全连接神经网络
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256), # 将输入的图像展平成向量
                    nn.ReLU(), # 第一个全连接层
                    # 在第一个全连接层之后添加一个dropout层
                    nn.Dropout(dropout1),
                    nn.Linear(256, 256), # 第二个全连接层
                    nn.ReLU(), # 第二个全连接层
                    # 在第二个全连接层之后添加一个dropout层
                    nn.Dropout(dropout2),
                    nn.Linear(256, 10)) # 输出层

# 定义一个权重初始化函数，对模型的权重进行初始化
def init_weights(m):
    if type(m) == nn.Linear: # 判断 m 是否是 nn.Linear 类型的参数
        nn.init.normal_(m.weight, std=0.01) # nn.init.normal_() 对该参数的权重进行标准正态分布的随机初始化：均值为0，标准差为1

# 对模型进行权重初始化
net.apply(init_weights);
```

```
In [7]: trainer = torch.optim.SGD(net.parameters(), lr=lr) # 创建一个随机梯度下降sgd优化器，优化net中所有可训练参数（即权重和偏置）
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer) # 训练模型
```



Question

更改暂退法第一层和第二层的概率会发生什么，定量描述结果，总结定性的结论

提高dropout1, dropout2; dropout1> ropout2

第一层暂退法概率0.5 第二层暂退法概率0.8

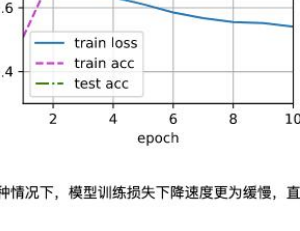
```
In [8]: dropout1, dropout2 = 0.5, 0.8

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)
```



此种情况下，模型训练损失下降速度较慢，测试准确性也出现波动的现象。

最终训练损失在epoch=10时下降到0.45，低于dropout1, dropout2 = 0.2, 0.5时的训练损失0.35

模型训练效果不如dropout1, dropout2 = 0.2, 0.5

提高dropout1, dropout2; dropout1< ropout2

第一层暂退法概率0.8 第二层暂退法概率0.3

```
In [9]: dropout1, dropout2 = 0.8, 0.3

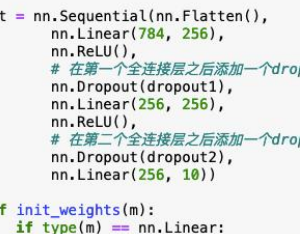
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    # 在第一个全连接层之后添加一个dropout层
                    nn.Dropout(dropout1),
                    nn.Linear(256, 256),
                    nn.ReLU(),
                    # 在第二个全连接层之后添加一个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)
```

```
AssertionError: Traceback (most recent call last)
Cell [9], line 21
18 net.apply(init_weights);
20 trainer = torch.optim.SGD(net.parameters(), lr=lr)
--> 21 d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
File ~/local/lib/python3.9/site-packages/d2l/torch.py:340, in train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
339 animator.add(epoch + 1, train_metrics + (test_acc,))
338 train_loss, train_acc = train_metrics
--> 340 assert train_loss < 0.5, train_loss
341 assert train_acc > 0.7, train_acc
342 assert test_acc <= 1 and test_acc > 0.7, test_acc
AssertionError: 0.5418769829432169
```



此种情况下，模型训练损失下降速度更为缓慢，直到epoch=10时损失仍在0.5以上，训练效果非常糟糕。

降低dropout1, dropout2; dropout1< ropout2

第一层暂退法概率0.1 第二层暂退法概率0.3

```
In [11]: dropout1, dropout2 = 0.1, 0.3

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)
```



此种情况下，模型训练损失下降速度提高，训练和测试准确性均为稳定且数值较高，只是测试的准确性在epoch=4时出现微小波动，后回归稳定。

最终训练损失在epoch=10时下降到0.35以下，低于dropout1, dropout2 = 0.2, 0.5时的训练损失0.35

模型训练效果优于dropout1, dropout2 = 0.2, 0.5

降低dropout1, dropout2; dropout1> ropout2

第一层暂退法概率0.3 第二层暂退法概率0.1

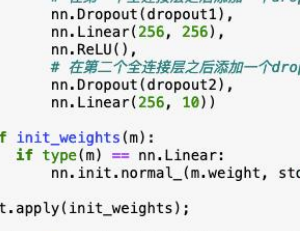
```
In [12]: dropout1, dropout2 = 0.3, 0.1

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)
```



此种情况下，模型训练损失下降速度，训练和测试准确性基本和一二层概率交换前相同。

模型训练效果和dropout1, dropout2 = 0.1, 0.3相近。

再度降低dropout1, dropout2; dropout1< ropout2

第一层暂退法概率0.02 第二层暂退法概率0.05

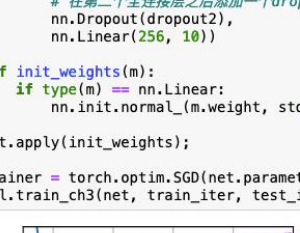
```
In [13]: dropout1, dropout2 = 0.02, 0.05

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)
```



此种情况下，模型训练损失下降速度再次大幅提高，训练和测试准确性稳定且数值较高。

最终训练损失在epoch=10时下降到0.3左右，模型训练效果优于所有未有的优势。

降低dropout1, dropout2; dropout1=ropout2

第一层暂退法概率0.2 第二层暂退法概率0.2

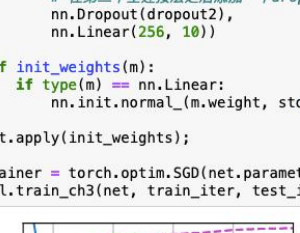
```
In [14]: dropout1 = dropout2 = 0.2

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)
```



此种情况下，模型训练损失下降速度较快，训练和测试准确性稳定且数值较高。

最终训练损失在epoch=10时下降到0.38左右，高于dropout1=dropout2 = 0.2时的训练损失

模型训练效果稍差于dropout1=dropout2 = 0.2，但训练效果仍优于 dropout1=dropout2 = 0.1, 0.3时模型结果相似，说明第一二层暂退概率取值可以相同。

再度降低dropout1, dropout2; dropout1= ropout2

第一层暂退法概率0.02 第二层暂退法概率0.02

```
In [15]: dropout1 = dropout2 = 0.02

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)
```



此种情况下，模型训练损失下降速度再次大幅提高，训练和测试准确性稳定且数值高。

最终训练损失在epoch=10时下降到0.3左右，高于dropout1=dropout2 = 0.2时的训练损失

模型训练效果稍差于dropout1=dropout2 = 0.2，但训练效果仍优于 dropout1=dropout2 = 0.1, 0.3时模型结果相似，说明第一二层暂退概率取值可以相同。

升高dropout1, dropout2; dropout1=ropout2

第一层暂退法概率0.6 第二层暂退法概率0.6

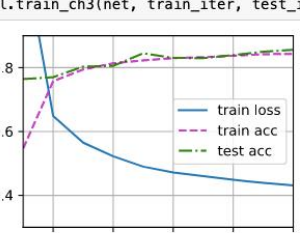
```
In [17]: dropout1 = dropout2 = 0.6

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)
```



此种情况下，模型训练损失下降速度较dropout1=dropout2 = 0.4低，训练和测试准确性稳定且数值较高。

最终训练损失在epoch=10时下降到0.42左右，高于dropout1=dropout2 = 0.4时的训练损失。

定性结论

- 1、两层暂退法概率不可以过高，过高会影响训练效果，使训练损失居高不下。
- 2、本模型采用降低的暂退概率，在所有案例中，当一二层概率均较低时，为0.02时模型训练效果最优。
- 3、第一层的暂退法概率和第二层的暂退法概率交换或数值相同时，模型训练表现基本相同，说明不同层暂退概率的顺序并非模型训练的重要影响因素，数值才是主要影响因素。