Normalization

一种流行且有效的技术，可持续加速深层网络的收敛速度。 再结合下章的残差块，批量规范化使得研究人员能够训练100层以上的网络。

常用Batch Normalization.

Traditional Normalization Code

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

# Traditional Normalization
def batch_norm(X,gamma,beta,moving_mean,moving_var,eps,momentum): # X为输入，gamma、beta为学的参数。mean、var为全局的均值、方差。eps为避免除0的参数。
    if not torch.is_grad_enabled():
        X_hat = (X - moving_mean)/torch.sqrt(moving_var + eps)
        # 做推理时，可能只有一个图片进来，没有一个批量进来，因此这里用的全局的均值、方差。
        # 在预测中，一般用整个预测数据集的均值和方差。加eps为了避免方差为0，除以0了。
    else:
        assert len(X.shape) in (2,4) # 批量数+通道数+图片高+图片宽=4
        if len(X.shape) == 2:  # 2 表示2表示有两个维度，样本和特征，表示全连接层应该是：2代表全连接层 (batch_size, feature)
            mean = X.mean(dim=0) # 按行求均值，即对每一列求一个均值出来。mean为1Xn的行向量
            var = ((X-mean)**2).mean(dim=0) # 方差也是行向量
        else: # 4 表示卷积层
            mean = X.mean(dim=(0,2,3),keepdim=True) # 0为批量大小，1为输出通道，2、3为高宽。这里是沿着通道维度求均值，0->batch内不同样本，2 3 ->同一通道层的所有值求均值，获得一个1xnx1x1的4D向量。
            var = ((X-mean)**2).mean(dim=(0,2,3),keepdim=True) # 同样对批量维度、高宽取方差。每个通道的每个像素位置 计算均值方差。
        X_hat = (X-mean) / torch.sqrt(var + eps) # 训练用的计算出来的均值、方差，推理用的全局的均值、方差
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean # 累加，将计算的均值累积到全局的均值上，更新moving_mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var # 当前全局的方差与当前算的方差做加权平均，最后会无限逼近真实的方差。仅训练时更新，推理时不更新。
    Y = gamma * X_hat + beta # Y 为归一化后的输出
    return Y, moving_mean.data, moving_var.data
# 创建一个正确的BatchNorm图层
class BatchNorm(nn.Module):
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features) # num_features 为 feature map 的多少，即通道数的多少
        else:
            shape = (1, num_features,1,1)
        self.gamma = nn.Parameter(torch.ones(shape)) # 伽马初始化为全1，贝塔初始化为全0
        self.beta = nn.Parameter(torch.zeros(shape)) # 伽马为要拟合的均值，贝塔为要拟合的方差
        self.moving_mean = torch.zeros(shape)
        # 伽马、贝塔需要在反向传播时更新，所以放在nn.Parameter里面，moving_mean、moving_var不需要迭代，所以不放在里面
```

```
        self.moving_var = torch.ones(shape)

    def forward(self, X):
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device) #
            self.moving_var = self.moving_var.to(X.device)
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta,self.moving_mean,self.moving_var,
            eps=1e-5,momentum=0.9)
        return Y
# 应用BatchNorm于LeNet模型
net = nn.Sequential(nn.Conv2d(1,6,kernel_size=5),BatchNorm(6,num_dims=4), # 在第一个卷积后面加了BatchNorm
                    nn.Sigmoid(),nn.MaxPool2d(kernel_size=2,stride=2),
                    nn.Conv2d(6,16,kernel_size=5),BatchNorm(16,num_dims=4), # BatchNorm的feature map为卷积层的输出通道数。这里BatchNorm加在激活函数前面。
                    nn.Sigmoid(),nn.MaxPool2d(kernel_size=2,stride=2),
                    nn.Flatten(),nn.Linear(16*4*4,120),
                    BatchNorm(120,num_dims=2),nn.Sigmoid(),
                    nn.Linear(120,84),BatchNorm(84,num_dims=2),
                    nn.Sigmoid(),nn.Linear(84,10))
# 在Fashion-MNIST数据集上训练网络
lr,num_epochs,batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())  # 变快是指收敛所需的迭代步数变少了，但每次迭代计算量更大了呀，所以从时间上来讲跑得慢了
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

Pytorch Code

```
net = nn.Sequential(nn.Conv2d(1,6,kernel_size=5),nn.BatchNorm2d(6),
                    nn.Sigmoid(),nn.MaxPool2d(kernel_size=2,stride=2),
                    nn.Conv2d(6,16,kernel_size=5),nn.BatchNorm2d(16),
                    nn.Sigmoid(),nn.MaxPool2d(kernel_size=2,stride=2),
                    nn.Flatten(),nn.Linear(256,120),nn.BatchNorm1d(120),
                    nn.Sigmoid(),nn.Linear(120,84),nn.BatchNorm1d(84),
                    nn.Sigmoid(),nn.Linear(84,10))
# 使用相同超参数来训练模型
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
d2l.plt.show()
```