PyTorch理解更多神经网络优化方法

1.了解不同优化器

2.书写优化器代码

3.Momentum

4.二维优化，随机梯度下降法进行优化实现

5.Ada自适应梯度调节法

6.RMSProp

7.Adam

8.PyTorch种优化器选择

所有的优化方法都封装在torch.optim里面，他的设计很灵活，可以扩展为自定义的优化方法。所有的优化方法都是继承了基类optim.Optimizer。并实现了自己的优化步骤。

SGD 是最普通的优化器, 也可以说没有加速效果, 而 Momentum 是 SGD 的改良版, 它加入了动量原则. 后面的 RMSprop 又是 Momentum 的升级版. 而 Adam 又是 RMSprop 的升级版. 不过从这个结果中我们看到, Adam 的效果似乎比 RMSprop 要差一点. 所以说并不是越先进的优化器, 结果越佳. 我们在自己的试验中可以尝试不同的优化器, 找到那个最适合你数据/网络的优化器。

import torch

import torch.utils.data as Data

import torch.nn.functional as F

import matplotlib.pyplot as plt

torch.manual\_seed(1) # reproducible

LR = 0.01

BATCH\_SIZE = 32

EPOCH = 12

# fake dataset

x = torch.unsqueeze(torch.linspace(-1, 1, 1000), dim=1)

y = x.pow(2) + 0.1\*torch.normal(torch.zeros(\*x.size()))

# plot dataset

plt.scatter(x.numpy(), y.numpy())

plt.show()

# 使用上节内容提到的 data loader

torch\_dataset = Data.TensorDataset(x, y)

loader = Data.DataLoader(dataset=torch\_dataset, batch\_size=BATCH\_SIZE, shuffle=True, num\_workers=2,)

# 默认的 network 形式

class Net(torch.nn.Module):

def \_\_init\_\_(self):

super(Net, self).\_\_init\_\_()

self.hidden = torch.nn.Linear(1, 20) # hidden layer

self.predict = torch.nn.Linear(20, 1) # output layer

def forward(self, x):

x = F.relu(self.hidden(x)) # activation function for hidden layer

x = self.predict(x) # linear output

return x

# 为每个优化器创建一个 net

net\_SGD = Net()

net\_Momentum = Net()

net\_RMSprop = Net()

net\_Adam = Net()

nets = [net\_SGD, net\_Momentum, net\_RMSprop, net\_Adam]

# different optimizers

opt\_SGD = torch.optim.SGD(net\_SGD.parameters(), lr=LR)

opt\_Momentum = torch.optim.SGD(net\_Momentum.parameters(), lr=LR, momentum=0.8)

opt\_RMSprop = torch.optim.RMSprop(net\_RMSprop.parameters(), lr=LR, alpha=0.9)

opt\_Adam = torch.optim.Adam(net\_Adam.parameters(), lr=LR, betas=(0.9, 0.99))

optimizers = [opt\_SGD, opt\_Momentum, opt\_RMSprop, opt\_Adam]

loss\_func = torch.nn.MSELoss()

losses\_his = [[], [], [], []] # 记录 training 时不同神经网络的 loss

for epoch in range(EPOCH):

print('Epoch: ', epoch)

for step, (b\_x, b\_y) in enumerate(loader):

# 对每个优化器, 优化属于他的神经网络

for net, opt, l\_his in zip(nets, optimizers, losses\_his):

output = net(b\_x) # get output for every net

loss = loss\_func(output, b\_y) # compute loss for every net

opt.zero\_grad() # clear gradients for next train

loss.backward() # backpropagation, compute gradients

opt.step() # apply gradients

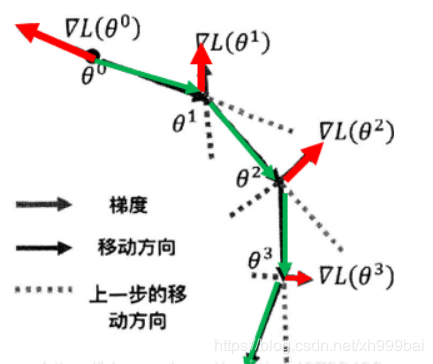
l\_his.append(loss.data.numpy()) # loss recoder

Momentum（动量）

动量法是梯度下降法的变式，在随机梯度下降的同时，增加动量。这是来自于物理中的概念，可以想象损失函数是一个山谷，一个球从山谷滑下来，在一个平坦的地势，球的滑动速度就会慢下来，可能陷入一些鞍点或者局部极小值点；

这个时候给它增加动量就可以让它从高处滑落时的势能转换为平地的动能，相当于惯性增加了小球在平地滑动的速度，从而帮助其跳出鞍点或者局部极小点。

动量怎么计算呢？动量的计算基于前面的梯度，也就是说参数更新不仅仅基于当前的梯度，也基于之前的梯度，可用下图来简单说明：



二维优化，随机梯度下降法进行优化实现

Ada自适应梯度调节法

用一阶导去估计海森矩阵的方法，但是是来源于Ada-Grad方法，具有5个优点：

自适应，省去了人工设定学习率的过程；

只用到一阶信息，计算开销小；

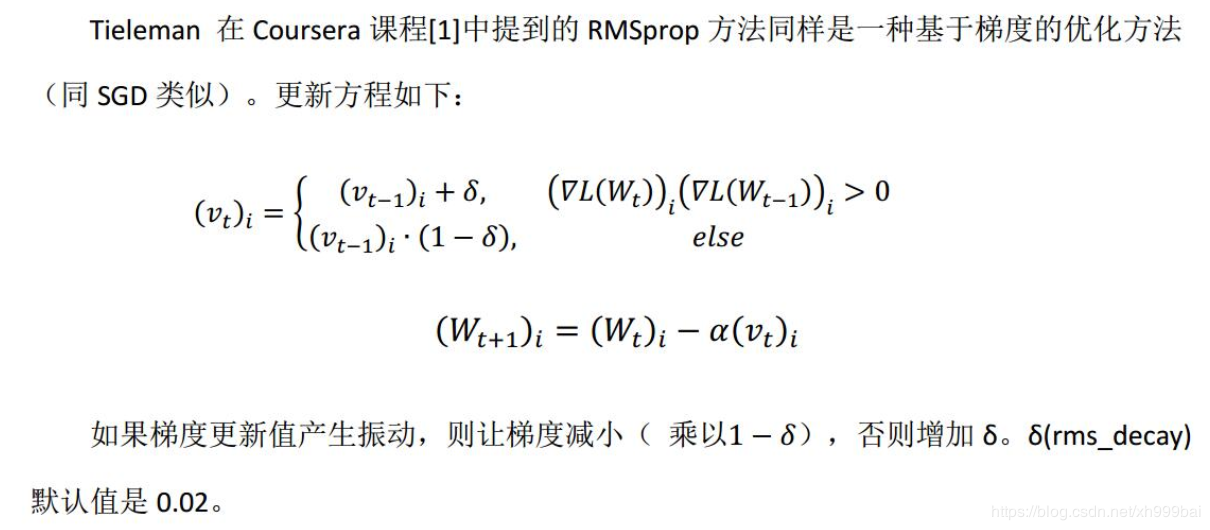
超参数不敏感性，其公式中额外增加的参数的选择对求解结果没有很大影响；

鲁棒性；

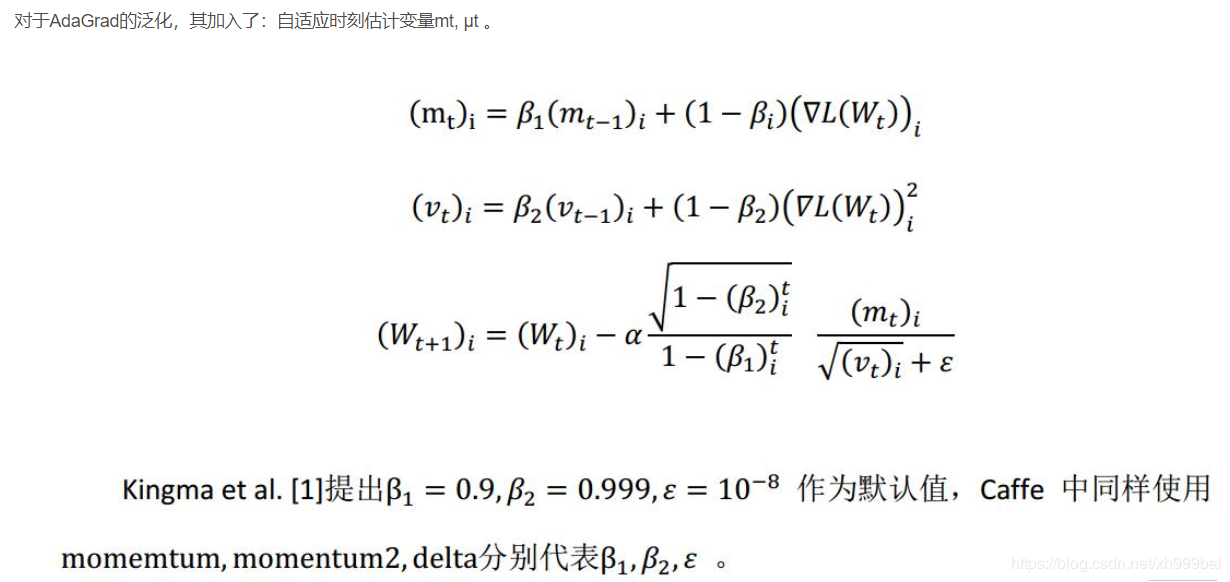
按维度分开计算学习率；

由于学习率在网络的训练过程中是应该逐渐减小的，这就是学习率的退火处理。

RMSProp



Adam



PyTorch种优化器选择

对于稀疏数据，尽量使用学习率可自适应的优化方法，不用手动调节，而且最好采用默认值

SGD通常训练时间更长，容易陷入鞍点，但是在好的初始化和学习率调度方案的情况下，结果更可靠

如果在意更快的收敛，并且需要训练较深较复杂的网络时，推荐使用学习率自适应的优化方法。

Adadelta，RMSprop，Adam是比较相近的算法，在相似的情况下表现差不多。