机器学习中的最优化方法

https://github.com/w-gc/
东北大学计算机系

版本:0.1

更新:July 21, 2020

1 导言

1.1 最优化与机器学习

在机器学习中用到的梯度下降类算法，实际上在最优化中早有提及，而且最优化中不少优化算法也是应用到了机器学习中。那么问题来了，最优化中的算法是不是可以直接拿过来用在机器学习中呢？

显然不是的，两者虽然都是“最优化”但是又有各自面临的问题。它们解决问题的思路是有区别，这导致两者的算法并不是简单的照搬照抄，而是需要做相应的“适应”。

举个例子，假设我们在拟合一条直线，现在有 6 个样本点，见表**??**。显然，我们需要拟合的模型是 $y = 3 * x + 5$。需要说明的是样本的 y_i 是不等于 $3 * x_i + 5$，因为采样过程中存在不可避免的误差。

表 1: 样本数据

样本序号 i	x_i	y_i
1	0.5000	6.9243
2	1.0000	8.4900
3	1.5000	10.0835
4	2.0000	11.0833
5	2.5000	13.1602
6	3.0000	14.0523

我们首先从解析求解的角度看待这个问题，建立模型 $y = ax + b$。求解参数 a, b，让 y_i 和 $ax_i + b$ 尽量接近。这里如果我们采用 l_2 范式描述两者的接近程度（距离），那么得到的无约束最

优化模型为

$$\min_{a,b} \sum_{i=1}^{6} \left\| ax_i + b - y_i \right\|_2^2. \tag{1}$$

为了书写的方便记 $L(a,b) = \sum_{i=1}^{6} \left\| ax_i + b - y_i \right\|_2^2$。那么问题便是求解无约束的二元函数的极小值，我们可以用极值点的必要条件来求解。

$$\begin{cases} \dfrac{\partial L}{\partial a} = \displaystyle\sum_{i=1}^{6} 2x_i \left(ax_i + b - y_i \right) = 0 \\ \dfrac{\partial L}{\partial b} = \displaystyle\sum_{i=1}^{6} 2 \left(ax_i + b - y_i \right) = 0. \end{cases} \tag{2}$$

显然上式可以求得解析解

$$\begin{cases} a = \dfrac{\sum_{i=1}^{6} x_i y_i - \left(\frac{1}{6} \sum_{i=1}^{6} y_i \right) \sum_{i=1}^{6} x_i}{\sum_{i=1}^{6} x_i^2 - \left(\frac{1}{6} \sum_{i=1}^{6} x_i \right) \sum_{i=1}^{6} y_i} \\ b = \left(\dfrac{1}{6} \displaystyle\sum_{i=1}^{6} y_i \right) - a \left(\dfrac{1}{6} \displaystyle\sum_{i=1}^{6} x_i \right). \end{cases} \tag{3}$$

带入样本点值得到 $y = 2.8943x + 5.5672$，见图**??**。

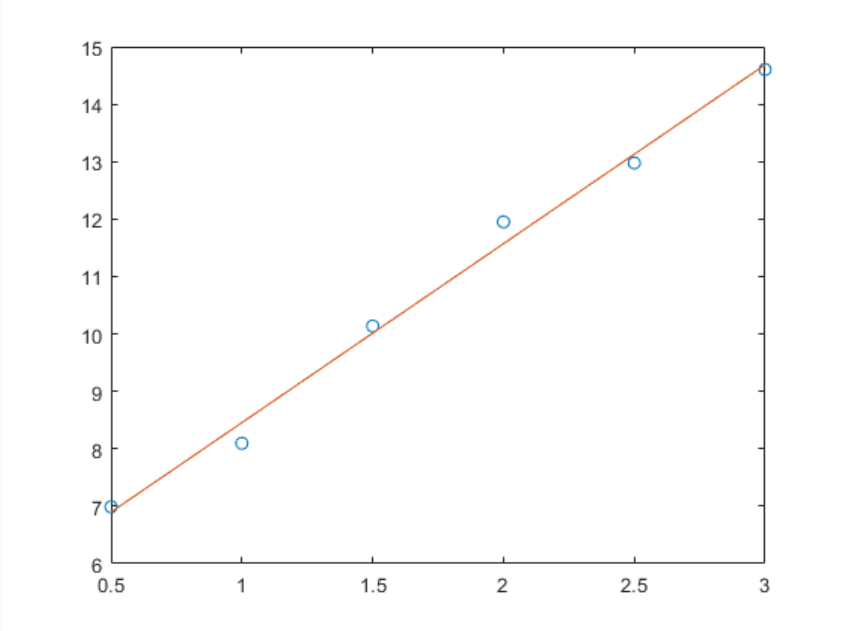

图 1: 采用解析法

显然（**??**）式解析解可以推广至有限的 n 个样本，只需将式（**??**）的 6 改成 n 即可。上述过程也称作是做一元线性回归，采用的具体解法称作最小二乘。

```
% 采用解析求解法示例代码
clear all,clc
x = (0.5 : 0.5 : 3)';
y = 3 * x + 5 + rand(length(x),1);
```

```
syms a b real
y_ = (a*x + b);
L = (y_ - y)' * (y_ - y);
d_L_a = diff(L, a);
d_L_b = diff(L, b);
[slove_a, slove_b] = solve([d_L_a, d_L_b]);
disp([slove_a, slove_b]);
y_ = slove_a * x + slove_b;
plot(x, y ,'o',x, y_)
```

当问题变得更复杂，比如样本数目增加，或者参数增加，解析法将很能求解，涉及到了多元非线性方程组。这里我们再考虑最优化教材中常用的梯度下方法，这里以最速下降为例。对于参数 θ 其通用算法为（**??**）。这里在确定步长因子 t 时，显然这个问题比较简单所以其实可以不用近似算法求解，直接整理得到关于 t 的一元函数，求解析解。但是当问题复杂时，求步长因子 t 的解析解并不简单，所以这里也同样使用近似求解。

$$\begin{cases} t = \arg\max\limits_{t} f(\theta + t\nabla\theta) \\ \theta = \theta - t\nabla\theta. \end{cases} \tag{4}$$

具体到我们的这个问题其算法为

Algorithm 1 采用最速下降法

Require: $(x_i, y_i), i = 1, \cdots, 6$

Ensure: a, b

随机初始化 a, b

for $i = 1; i < 100; i++$ **do**

$\hat{y}_i = ax_i + b, i = 1, \cdots, 6$

$L(a,b) = \sum_{i=1}^{6} \left\| \hat{y}_i - y_i \right\|_2^2$

$\nabla a = \sum_{i=1}^{6} 2x_i \left(\hat{y}_i - y_i \right)$

$\nabla b = \sum_{i=1}^{6} 2 \left(\hat{y}_i - y_i \right)$

寻找 $L(a + t\nabla a, b + t\nabla b)$ 包含一个局部极值点的区间 $[l, r]$

在区间 $[l, r]$ 选找 $L(a + t\nabla a, b + t\nabla b)$ 的极值点 t

$a = a + t\nabla a$

$b = b + t\nabla b$

end for

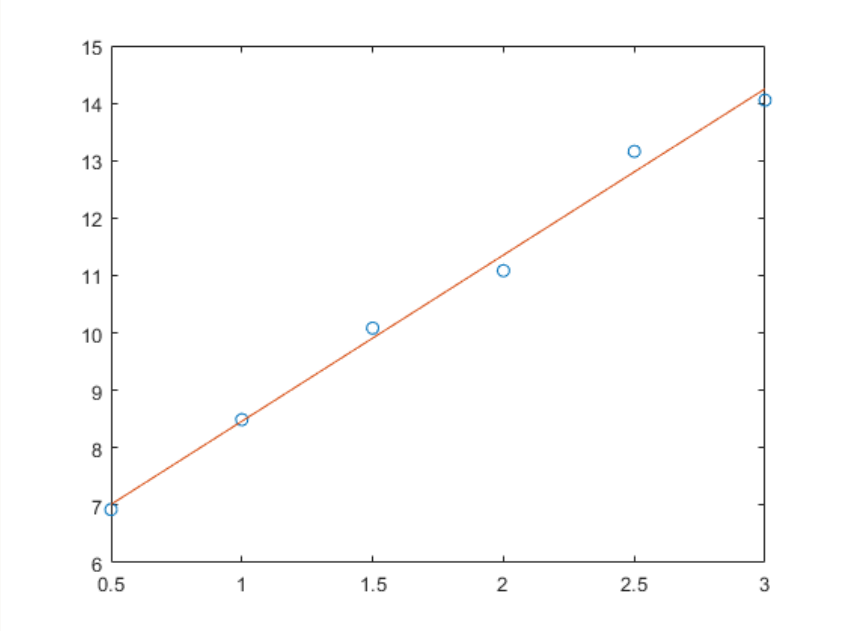

图 2: 采用最速下降法

同样的，这个算法可以推广至有限个样本 n，只需要将上面的 6 改成 n 即可。参考代码中，关于关于步长因子 t 的局部极值区间和确定步长因子 t 的局部极值部分实现，参考 [?]。

```
% 采用最速下降法示例代码
clear all,clc
x = (0.5 : 0.5 : 3)';
% y = 3 * x + 5 + rand(length(x),1);
y = [6.9243, 8.4900, 10.0835, 11.0833, 13.1602, 14.0523]';
a = rand(1);
b = rand(1);
for i = 1 : 100
    y_ = (a*x + b);
    L = (y_ - y)' * (y_ - y);
    d_L_a = 45.5 * a + 21* b - 248.6028;
    d_L_b = 21*a + 12*b - 127.5872;
    %-----------------------------------确定关于步长因子t的局部极值区间
    ft = @(t)(((a - t * d_L_a)*x + (b - t * d_L_b)) - y)' * (((a - t * d_L_a)*x + (b - t *
        d_L_b)) - y);
    [InterL, InterR] = searchInterval(ft);
    %-----------------------------------确定步长因子t的局部极值
    t = bestT(ft, InterL, InterR);
    %-----------------------------------更新参数a,b
    a = (a - t * d_L_a);
    b = (b - t * d_L_b);
end
disp([a, b]);
y_ = a * x + b;
plot(x, y ,'o',x, y_)
```

最后，我们来看看如果是从神经网络的角度怎么解决这个问题。将其看成一个非常简单的神经网络，见图**??**。神经网络的优化使用到了一个叫**学习率**，其实和最速下降法中的步长因子是同一事物，都是控制沿着梯度方向变化的程度。

```
% 采用机器学习方法示例代码
clear all,clc
x = (0.5 : 0.5 : 3)';
% y = 3 * x + 5 + rand(length(x),1);
y = [6.9243, 8.4900, 10.0835, 11.0833, 13.1602, 14.0523]';
a = rand(1);
b = rand(1);
lr = 0.0001;
for i = 1:1000
    y_ = a * x - b;
    e = y_ - y;
    d_L_a = 2 * x'* e / length(x);
    d_L_b = 2 * mean(e);
    a = a - lr * d_L_a;
    b = b - lr * d_L_b;
end
disp([a, b]);
y_ = a * x + b;
plot(x, y ,'o',x, y_)
```

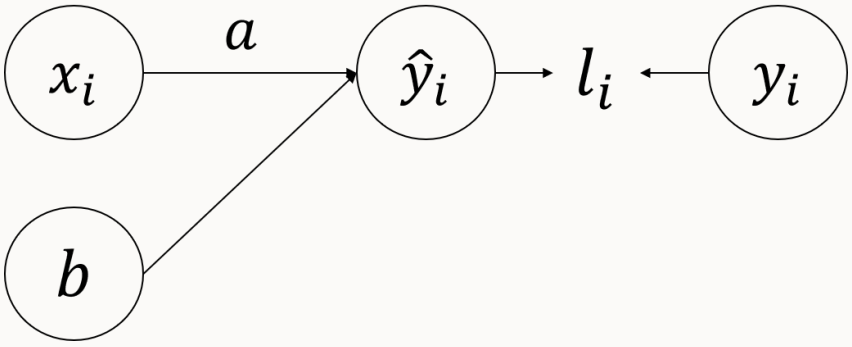

图 3: 看成一个简单的神经网络

观察图**??**，发现在这个简单的问题上，这个简单的神经网络的表现并没有好过传统的最优化方法，而且还非常不稳定。

总得来说，在以前的最优化问题中，可以采用解析法，或者梯度下降方法。虽然神经网络的优化算法也是梯度下降算法，但是两者还是有些区别。主要是关于步长因子 t，最优化教材中的算法往往是采用求解 t 的最优步长，这里其实上是需要整理到关于 t 的函数。但是在深度的网络中，虽然只有一个参数 t 但要整理得到 t 函数也是有困难的。而且还需要搜素一个包含 t 的局部极小值区间，然后再在这个区间里搜素最优值。而深度网络的优化或者说机器学习中的优化，一般是将其设置为一个很小的数值，比如 10^{-4}。

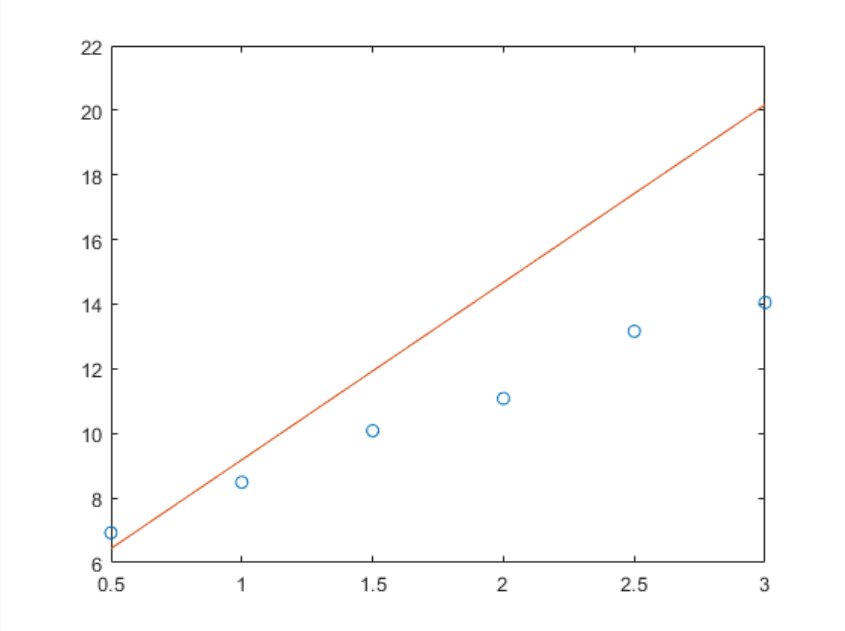

图 4: 采用 ML 的算法

1.2 一个简单的分类网络

CIFAR-10 数据集（加拿大高级研究所）是一种常用于训练图像的采集机器学习和计算机视觉算法。它是机器学习研究中使用最广泛的数据集之一。CIFAR-10 数据集包含 10 种不同类别的 60,000 张 32x32 彩色图像。10 个不同的类别分别代表飞机，汽车，鸟类，猫，鹿，狗，青蛙，马，轮船和卡车。每个类别有 6,000 张图像 [**?**]。

这里采用一个简单卷积神经网络：

网络结构

```
Net(
(conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(maxpool): MaxPool2d(kernel_size=(3, 3), stride=2, padding=(1, 1), dilation=1, ceil_mode=False)
(conv2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(avgpool): AvgPool2d(kernel_size=(3, 3), stride=2, padding=(1, 1))
(conv3): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
(conv4): Conv2d(64, 64, kernel_size=(4, 4), stride=(1, 1))
(conv5): Conv2d(64, 10, kernel_size=(1, 1), stride=(1, 1))
)
```

整个运行程序为：

```
# -*- coding: utf-8 -*-
import random
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
```

```
import torch.optim as optim
# 导入不同的优化器
# from SVRG import SVRG

batch_size = 100
EPOCH = 10

transform = transforms.Compose([transforms.ToTensor(),
      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='/sdb/wgc/cifar10', train=True,
      download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size = batch_size,
      shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='/sdb/wgc/cifar10', train=False,
      download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size = batch_size,
      shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat', 'deer',
      'dog', 'frog', 'horse', 'ship', 'truck')

# dataiter = iter(trainloader)
# inputs, labels = dataiter.next()

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(in_channels = 3, out_channels = 32, kernel_size = 5,
              stride = 1, padding = (2, 2))
        self.maxpool = nn.MaxPool2d(kernel_size = (3, 3), stride = 2, padding=(1, 1))
        self.conv2 = nn.Conv2d(in_channels = 32, out_channels = 32, kernel_size = 5,
              stride = 1, padding = (2, 2))
        self.avgpool = nn.AvgPool2d(kernel_size = (3, 3), stride = 2, padding=(1, 1))
        self.conv3 = nn.Conv2d(in_channels = 32, out_channels = 64, kernel_size = 5,
              stride = 1, padding = (2, 2))
        self.conv4 = nn.Conv2d(in_channels = 64, out_channels = 64, kernel_size = 4,
              stride = 1)
        self.conv5 = nn.Conv2d(in_channels = 64, out_channels = 10, kernel_size = 1,
              stride = 1)

    def forward(self, x):
        batch_size = x.size()[0]
        x = F.relu(self.maxpool(self.conv1(x)))
        x = self.avgpool(F.relu(self.conv2(x)))
        x = self.avgpool(F.relu(self.conv3(x)))
        x = F.relu(self.conv4(x))
        x = self.conv5(x)
```

```
        x = x.view(batch_size, -1)
        return x

net = Net()

use_gpu = torch.cuda.is_available()
if(use_gpu):
    net = net.cuda()

criterion = nn.CrossEntropyLoss()

# 选择不同优化器
# optimizer = optim.SGD(net.parameters(), lr = 1e-2)

for epoch in range(EPOCH):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
        if(use_gpu):
            inputs = inputs.cuda()
            labels = labels.cuda()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        optimizer.zero_grad()

        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 100 == 99:    # print every 100 mini-batches
            print('[%d, %5d] loss: %.3f' % (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

  # 一个EPOCH训练的准确率
    correct = 0
    total = 0
    with torch.no_grad():
        for data in testloader:
            inputs, labels = data
            if(use_gpu):
                inputs = inputs.cuda()
                labels = labels.cuda()
            outputs = net(inputs)
            # outputs = F.softmax(outputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
    print('%d Accuracy of the network on the 10000 test images: %d %%' % (epoch + 1, 100 *
        correct / total))

print('---------Finished Training------------')

# 训练结束后在测试集上测试，并输出各个类别的准确率
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        if(use_gpu):
            inputs = inputs.cuda()
            labels = labels.cuda()
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1

for i in range(10):
    print('Accuracy of %5s : %2d %%' % (classes[i], 100 * class_correct[i] / class_total[i]))
```

1.3 优化方法主要类别

机器学习中的优化方法主要按导数类型，分为三大类：

- **一阶导信息** GD,SGD,NAG,AdaDelta/RMSProp,Adam,SAG,SVRG,ADMM,Frank-Wolfe
- **高阶导信息**共轭梯度法、(拟)牛顿、Natural Gradient
- **非导数**启发式，采样法

2 机器学习中常见优化模型

2.1 有监督学习

有一组样本 $(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$。定义 $f(x, \theta)$ 为机器学习模型（可以是回归，也可以是神经网络）。那么定义损失函数为 $l(y, f)$，这里的损失可以是 log-softmax，也可以是某种范式。那么建立的优化模型为

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} l_i \left(y_i, f(x_i, \theta)\right). \tag{5}$$

我们得到的这个目标函数（**??**）对于确定的样本，其实是只跟参数 θ 有关。而参数 θ 是我们需要”学习“的，它的值确定了我们的机器学习模型。θ 可以是回归模型里的参数 a, b，有可以是神经网络里的连接权重 w，当然也可以是卷积神经（CNN）中的卷积核。

为了防止过拟合，我们通常会在目标函数后面增加关于参数 θ 的惩罚项。比如在高维统计学中常用的 L_1, L_2 范式惩罚项，当然，在机器学习中也是常用 L_1, L_2 范式惩罚项。这里为了描述问题的更一般性，假设采用了 L_p 范式。则

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} l_i \left(y_i, f(x_i, \theta)\right) + \lambda \|\theta\|_{L_p}^2. \tag{6}$$

2.2 半监督学习

半监督学习是，部分数据有标签，部分数据没有标签。这里我们仅考虑 S3VM。假设有带标签数据 $(x_1, y_1), (x_1, y_1), \cdots, (x_l, y_l)$，和无标签数据 $(x_{l+1}, y_{l+1}), (x_{l+2}, y_{l+2}), \cdots, (x_{l+u}, y_{l+u})$，且 $N = l + u$。

$$\begin{cases} \min \|w\| + C \left[\sum_{i=1}^{l} \zeta^i + \sum_{j=l+1}^{N} \min(\epsilon_j^0, \epsilon_j^1) \right]. \\ s.t. y_i \left(w x_i + b\right) + \zeta^i \geq 1, \quad i = 1, 2, \cdots, l \\ \quad w x_j + b + \epsilon_j^0 \geq 1, \quad \epsilon_j^0 \geq 0, \quad j = l+1, l+2, \cdots, N \\ \quad -\left(w x_j + b\right) + \epsilon_j^1 \geq 1, \quad \epsilon_j^1 \geq 0, \quad j = l+1, l+2, \cdots, N \end{cases} \tag{7}$$

其中 ζ^i 是松弛变量，C 是一个错误率的惩罚因子，而 $\epsilon_j^0, \epsilon_j^1$ 分别是样本 x_j 分到 0、1 类的错分率。

2.3 无监督学习

这里主要介绍下 k-means、主成分分析（PCA）、和极大似然估计（MLE）的最优化模型。首先关于 k-means，假设有 K 个类别，第 k 类的样本集为 S_k，类中心为 μ_k。那么我们的目标就是找到

K，让每个样本集中的样本 $x \in S_k$，尽量接近类中心 μ_k。

$$\min_{S} \sum_{k=1}^{K} \sum_{x \in S_k} \|x - \mu_k\|_{L_p}^2. \tag{8}$$

PCA，将原来的样本从原来的样本空间投影至一个更小的空间里，目标是寻找能够是得投影前后两者相差最小的投影方式（映射）。

$$\min \sum_{i=1}^{N} \left\| \sum_{j=1}^{D'} z_{i,j} e_j - x_i \right\|_{L_p}^2, \quad D \gg D'. \tag{9}$$

其中，e_j 是小空间的基，而 z_i 是样本 x_i 在小空间上的投影坐标，而 $\sum_{j=1}^{D'} z_{i,j} e_j$ 是对样本 x_i 的重构。跟自编码器一样，先编码，再解码，解码结果与真实结果尽量接近。

极大似然估计，思想是让寻找参数让出现现在有的样本的情况最大。也就说，样本联合概率最大。

$$\max \sum_{i=1}^{N} \ln p(x_i; \theta); \tag{10}$$

2.4 强化学习

此领域不是我关心的重点，这里只简单介绍下。

$$\max_{\pi} V_{\pi}(s) = \mathbb{E}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | S_t = s \right). \tag{11}$$

（**??**）的意思是，在时刻 t，所在状态为 s，采用策略 π 获得的回报是 r，$\gamma \in [0, 1]$ 是折扣率，目标是选择最佳的策略 π 使得获得的回报是 r 最大。

3 一阶导信息

能使函数值减小的都可以称之为下降方向，而负梯度方向则是下降最快的方向。当然，需要说明的是，这里的所谓的“下降最快”是指在当前点的局部领域内下降最快。否则会以为直接用负梯度就是最好的，最速下降是最快的，实则不是。

3.1 梯度下降

这整个章节使用一阶导数信息的下降方法都可以说是梯度下降方法，但这里单独的这一小章节主要是讨论随机梯度、小批量、批量梯度下降。这三者的区别是由样本数目决定的，依次是一个样本、小批量样本、整个样本集合，也就是批量的大小 batch-size。batch-size 的取值在这里可以看出对参数更新策略是有影响的，但它的其他影响已经如何选择 batch-size 我们在之后的章节专讨论。

3.1.1 随机梯度下降

随机梯度下降是指将样本集打乱后，每个样本 (x_i, y_i) 有优化模型

$$\min_{\theta} l_i\left(y_i, f\left(x_i, \theta\right)\right). \tag{12}$$

更新策略为

$$\theta \leftarrow \theta - \eta \frac{\partial l_i}{\partial \theta}. \tag{13}$$

```python
import torch
from torch.optim.optimizer import Optimizer, required
class SGD(Optimizer):
    """
    (batch, mini-batch)Stochastic gradient descent:
    $$
    \theta \gets \theta - \eta \sum_{i=1}^{B} \frac{\partial l_i}{\partial \theta}.
    $$
    """
    def __init__(self, params, lr=required, weight_decay=0):
        if lr is not required and lr < 0.0:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if weight_decay < 0.0:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        defaults = dict(lr=lr, weight_decay=weight_decay)
        super(SGD, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(SGD, self).__setstate__(state)
        # for group in self.param_groups:
        #     group.setdefault('nesterov', False)
```

```
22
23      def step(self, closure=None):
24          """Performs a single optimization step.
25          Arguments:
26              closure (callable, optional): A closure that reevaluates the model
27                  and returns the loss.
28          """
29          loss = None
30          if closure is not None:
31              loss = closure()
32          for group in self.param_groups:
33              weight_decay = group['weight_decay']
34              for p in group['params']:
35                  if p.grad is None:
36                      continue
37                  d_p = p.grad.data
38                  if weight_decay != 0:
39                      d_p.add_(weight_decay, p.data)
40                  p.data.add_(-group['lr'], d_p)
41          return loss
```

3.1.2 批量梯度下降

对整个样本集 $(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$。优化模型**??**便是批量梯度下降的优化模型，而它的更新策略为

$$\theta \leftarrow \theta - \eta \sum_{i=1}^{N} \frac{\partial l_i}{\partial \theta}. \tag{14}$$

3.1.3 小批量梯度下降

小批量梯度下降是介于随机梯度下降和批量梯度下降中间的一种方法。对于一小批样本 $(x_1, y_1), (x_2, y_2), \cdots, (x_B, y_B)$。$B$ 是批量的大小（样本的个数），一般 $N \gg B$。建立的最优模型为

$$\min_{\theta} \frac{1}{B} \sum_{i=1}^{B} l_i \left(y_i, f(x_i, \theta)\right). \tag{15}$$

更新策略为

$$\theta \leftarrow \theta - \eta \sum_{i=1}^{B} \frac{\partial l_i}{\partial \theta}. \tag{16}$$

3.2 动量法

动量法的启发源自物理，想象一个小球从山顶上滚落下来，当它到达谷底时，从数学的角度来看此时的梯度时为 0 的，但是小球有惯性会继续想前。受这种启发，有了动量法，它的好处是能

够更好的摆脱局部最小值。

$$\begin{cases} v \leftarrow \gamma v + \eta \dfrac{\partial L}{\partial \theta} \\ \theta \leftarrow \theta - v. \end{cases} \tag{17}$$

其中 L 是模型的损失函数，γ 通常取值为 0.9。

可以看到，所谓的“动量”其实是梯度的历史值的累计，这个算法说明了梯度的历史信息也是有用的，有助于收敛。当然，记录梯度的历史信息是需要额外的存储空间，这个空间的大小跟参数的大小一致。

下面的代码实现本质来说也是一种小批量梯度下降方法，也是使用了批量的技术。

```
import torch
from torch.optim.optimizer import Optimizer, required

class MSGD(Optimizer):
    """
    Momentum Stochastic gradient descent:
    $$
    \left\{
    \begin{aligned}
    v \gets &  \gamma v + \eta \frac{\partial L}{\partial \theta} \\
    \theta \gets &  \theta - v.
    \end{aligned}
    \right.
    $$
    """

    def __init__(self, params, lr=required, momentum=0.9, dampening=0, weight_decay=0):
        if lr is not required and lr < 0.0:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if momentum < 0.0:
            raise ValueError("Invalid momentum value: {}".format(momentum))
        if weight_decay < 0.0:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))

        defaults = dict(lr=lr, momentum=momentum, dampening=dampening, weight_decay=
            weight_decay)
        super(MSGD, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(MSGD, self).__setstate__(state)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
```

```
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                d_p = p.grad.data
                if group['weight_decay'] != 0:
                    d_p.add_(group['weight_decay'], p.data)
                if group['momentum'] != 0:
                    param_state = self.state[p]
                    if 'momentum_buffer' not in param_state:
                        param_state['momentum_buffer'] = torch.clone(d_p).detach()
                    else:
                        param_state['momentum_buffer'].mul_(group['momentum']).add_(1 - group['
                            dampening'], d_p)
                    d_p = param_state['momentum_buffer']
                p.data.add_(-group['lr'], d_p)
        return loss
```

3.3 NAG(Nesterov Accelerated Gradient Descent)

NAG 方比动量法具有更快的收敛速度，更新策略为

$$\begin{cases} v \leftarrow \gamma v + \eta \dfrac{\partial L}{\partial (\theta - \gamma v)} \\ \theta \leftarrow \theta - v. \end{cases} \tag{18}$$

观察图**??**和图**??**，两个最重要的差别是 $\frac{\partial L}{\partial \theta}$ 和 $\frac{\partial L}{\partial(\theta-\gamma v)}$。用 Nesterov[**?**] 本人的解释是用了下一步 $(\theta - \gamma v)$ 的梯度, 见图。当然，后来有人 [**?**] 证明上式（**??**）与下式等价：

$$\begin{cases} v_t \leftarrow \gamma v_{t-1} + \eta \dfrac{\partial L}{\partial \theta_{t-1}} + \gamma \left(\dfrac{\partial L}{\partial \theta_{t-1}} - \dfrac{\partial L}{\partial \theta_{t-2}} \right) \\ \theta_t \leftarrow \theta_{t-1} - \gamma v_t. \end{cases} \tag{19}$$

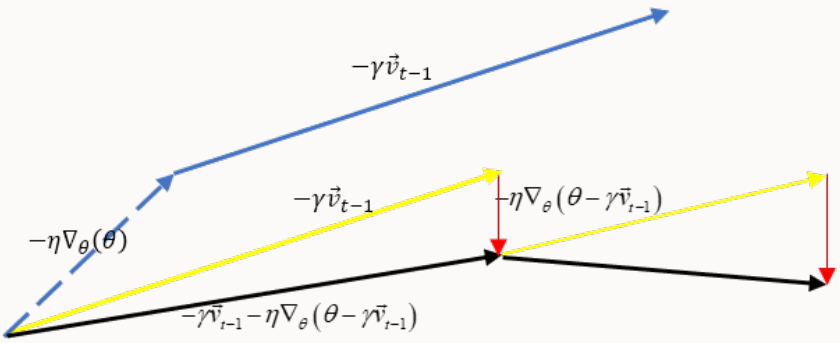

图 5: NAG 法更新

NAG 等价性证明.

$$\begin{aligned}\theta_t - \gamma v_t &= (\theta_{t-1} - v_t) - \gamma v_t \\ &= \theta_{t-1} - (1+\gamma)\, v_t \\ &= \theta_{t-1} - (1+\gamma)\left[\gamma v_{t-1} + \eta \nabla_\theta L(\theta - \gamma v_{t-1})\right] \\ &= \theta_{t-1} - \gamma v_{t-1} - \gamma^2 v_{t-1} - (1+\gamma)\,\eta \nabla_\theta L(\theta - \gamma v_{t-1}).\end{aligned} \tag{20}$$

记 $\hat{\theta}_{t-1} = \theta_{t-1} - \gamma v_{t-1}, \hat{v}_t = \gamma^2 v_{t-1} + (1+\gamma)\,\eta \nabla_\theta L(\hat{\theta}_{t-1})$. 另外

$$\begin{aligned}\hat{v}_t &= \gamma^2 v_{t-1} + (1+\gamma)\,\eta \nabla_\theta L(\hat{\theta}_{t-1}) \\ &= \gamma^2 \left[\gamma v_{t-2} + \eta \nabla_\theta L(\hat{\theta}_{t-2})\right] + (1+\gamma)\,\eta \nabla_\theta L(\hat{\theta}_{t-1}) \\ &= \gamma^3 v_{t-2} + \eta\gamma^2 \nabla_\theta L(\hat{\theta}_{t-2}) + \eta\gamma \nabla_\theta L(\hat{\theta}_{t-1}) + \eta \nabla_\theta L(\hat{\theta}_{t-1}) \\ &= \gamma^t v_1 + \eta \sum_{i=1}^{t-1} \gamma^i \nabla_\theta L(\hat{\theta}_{t-i}) + \eta \nabla_\theta L(\hat{\theta}_{t-1})\end{aligned} \tag{21}$$

因此得到 $\hat{v}_t - \gamma \hat{v}_{t-1} = \eta \nabla_\theta L(\hat{\theta}_{t-1}) + \gamma \left[\nabla_\theta L(\hat{\theta}_{t-1}) - \nabla_\theta L(\hat{\theta}_{t-2})\right]$. 整理后便有

$$\begin{aligned}\hat{v}_t &= \gamma \hat{v}_{t-1} + \eta \nabla_\theta L(\hat{\theta}_{t-1}) + \gamma \left[\nabla_\theta L(\hat{\theta}_{t-1}) - \nabla_\theta L(\hat{\theta}_{t-2})\right] \\ \hat{\theta}_t &= \hat{\theta}_{t-1} - \gamma \hat{v}_t\end{aligned}.$$

```python
import torch
from torch.optim.optimizer import Optimizer, required

class Nesterov(Optimizer):
    """
    Nesterov Accelerated Gradient Descent
    $$
    \left\{
    \begin{aligned}
    v \gets &  \gamma v + \eta \frac{\partial L}{\partial \left( \theta - \gamma v \right)} \\
    \theta \gets &  \theta - v.
    \end{aligned}
    \right.
    $$
    """

    def __init__(self, params, lr=required, momentum=0.9, weight_decay=0):
        if lr is not required and lr < 0.0:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if momentum <= 0.0:
            raise ValueError("Invalid momentum value: {}".format(momentum))
        if weight_decay < 0.0:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))

        defaults = dict(lr=lr, momentum=momentum, weight_decay=weight_decay)
```

```
26
27          super(Nesterov, self).__init__(params, defaults)
28
29      def __setstate__(self, state):
30          super(Nesterov, self).__setstate__(state)
31
32      def step(self, closure=None):
33          """Performs a single optimization step.
34          Arguments:
35              closure (callable, optional): A closure that reevaluates the model
36                  and returns the loss.
37          """
38          loss = None
39          if closure is not None:
40              loss = closure()
41
42          for group in self.param_groups:
43              weight_decay = group['weight_decay']
44              momentum = group['momentum']
45
46              for p in group['params']:
47                  if p.grad is None:
48                      continue
49                  d_p = p.grad.data
50                  if weight_decay != 0:
51                      d_p.add_(weight_decay, p.data)
52                  if momentum != 0:
53                      param_state = self.state[p]
54                      if 'momentum_buffer' not in param_state:
55                          param_state['momentum_buffer'] = torch.clone(d_p).detach()
56                      else:
57                          param_state['momentum_buffer'].mul_(momentum).add_(d_p)
58                      d_p = d_p.add(momentum, param_state['momentum_buffer'])
59                  p.data.add_(-group['lr'], d_p)
60          return loss
```

3.4 AdaGrad

AdaGrad 较少使用了，因为本来是为了用历史梯度信息来获得自适应的学习率，但是 $\frac{\eta}{V_t}$ 是单调递减的。

$$\begin{cases} g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\ V_t \leftarrow \sqrt{\displaystyle\sum_{i=0}^{t} (g_t)^2 + \epsilon} \\ \theta \leftarrow \theta - \dfrac{\eta}{V_t} g_t \end{cases} \tag{22}$$

需要存储一个和参数一样大小的 V_t 和临时梯度 $\frac{\partial L(\theta_t)}{\partial \theta}$。

因为 $\frac{\eta}{V_t}$ 是单调递减的，这里比之其他方法需要比较大的学习率 η，否则会过早梯度消失。

```
import torch
from torch.optim.optimizer import Optimizer, required

class AdaGrad(Optimizer):
    """AdaGrad algorithm.
    $$
    \left\{
    \begin{aligned}
        g_t \gets & \frac{ \partial L(\theta_t) }{ \partial \theta } \\
        V_t \gets & \sqrt{\sum_{i = 0}^{t} \left(g_t\right)^2 + \epsilon} \\
        \theta \gets &  \theta - \frac{\eta}{V_t} g_t
    \end{aligned}
    \right.
    $$
    """

    def __init__(self, params, lr=1e-2, lr_decay=0, weight_decay=0, initial_accumulator_value
        =0, eps=1e-10):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= lr_decay:
            raise ValueError("Invalid lr_decay value: {}".format(lr_decay))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        if not 0.0 <= initial_accumulator_value:
            raise ValueError("Invalid initial_accumulator_value value: {}".format(
                initial_accumulator_value))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))

        defaults = dict(lr=lr, lr_decay=lr_decay, eps=eps, weight_decay=weight_decay,
                        initial_accumulator_value=initial_accumulator_value)
        super(AdaGrad, self).__init__(params, defaults)

        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'] = 0
                state['sum'] = torch.full_like(p.data, initial_accumulator_value)

    def __setstate__(self, state):
        super(AdaGrad, self).__setstate__(state)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
```

```
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            loss = closure()
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('Adadelta does not support sparse gradients')
                state = self.state[p]
                state['step'] += 1
                if group['weight_decay'] != 0:
                    grad = grad.add(group['weight_decay'], p.data)
                clr = group['lr'] / ( 1 + (state['step'] - 1) * group['lr_decay'] )
                state['sum'].addcmul_(1, grad, grad)
                std = state['sum'].sqrt().add_(group['eps'])
                p.data.addcdiv_(-clr, grad, std)
        return loss
```

3.5 AdaDelta/RMSProp

$$
\begin{cases}
g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\
E[g^2]_t \leftarrow \gamma E[g^2]_{t-1} + (1-\gamma)(g_t)^2 \\
\Delta\theta_t \leftarrow \dfrac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \\
\theta_{t+1} \leftarrow \theta_t - \Delta\theta_t
\end{cases}
\tag{23}
$$

需要存储一个和参数一样大小的 $E[g^2]_t, \Delta\theta_t$ 和临时梯度 $\frac{\partial L(\theta_t)}{\partial \theta}$。超参数 γ 通常取值为 0.9。

上式(**??**)中的学习率 η 也可以改成自适应的。总共需要 4 倍的和参数 θ 大小的内存。

$$
\begin{cases}
g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\
E[g^2]_t \leftarrow \gamma E[g^2]_{t-1} + (1-\gamma)(g_t)^2 \\
\Delta\theta_t \leftarrow \dfrac{\sqrt{E[(\Delta\theta_t)^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t \\
E[(\Delta\theta_t)^2]_t \leftarrow \gamma E[(\Delta\theta_t)^2]_{t-1} + (1-\gamma)(\Delta\theta_t)^2 \\
\theta_{t+1} \leftarrow \theta_t - \Delta\theta_t
\end{cases}
\tag{24}
$$

式(**??**)中的 γ 取值为 0.9，则为 RMSProp 算法。

$$\begin{cases} g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\ E[g^2]_t \leftarrow 0.9E[g^2]_{t-1} + 0.1(g_t)^2 \\ \Delta\theta_t \leftarrow \dfrac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \\ \theta_{t+1} \leftarrow \theta_t - \Delta\theta_t \end{cases} \tag{25}$$

这里，我们使用的是学习率自适应法。学习率不自适应的时候，表现非常出色。而学习率自适应的时候，表现得发挥不稳定，有时候表现极好非常快就收敛了，有时候却始终不更新权重(也算是提前收敛了)。这是跟 $E[g^2]_t, E[(\Delta\theta_t)^2]_t$ 的初始化有关系。

```python
import torch
from torch.optim.optimizer import Optimizer, required

class AdaDelta(Optimizer):
    """Adadelta algorithm.
    $$
    \left\{
    \begin{aligned}
        g_t \gets & \frac{ \partial L(\theta_t) }{ \partial \theta } \\
        E[g^2]_t \gets & \gamma E[g^2]_{t-1} + (1 - \gamma) (g_t)^2 \\
        \Delta \theta_t \gets & \frac{\eta}{\sqrt{ E[g^2]_t +\epsilon }} g_t \\
        \theta_{t+1} \gets &  \theta_t - \Delta \theta_t
    \end{aligned}
    \right.
    $$
    """

    def __init__(self, params, lr=1.0, rho=0.9, eps=1e-6, weight_decay=0):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= rho <= 1.0:
            raise ValueError("Invalid rho value: {}".format(rho))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))

        defaults = dict(lr=lr, rho=rho, eps=eps, weight_decay=weight_decay)
        super(AdaDelta, self).__init__(params, defaults)

        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['square_avg'] = torch.zeros_like(p.data)
                if group['lr'] == 1:
                    state['acc_delta'] = torch.zeros_like(p.data)
```

```
37
38      def step(self, closure=None):
39          """Performs a single optimization step.
40          Arguments:
41              closure (callable, optional): A closure that reevaluates the model
42                  and returns the loss.
43          """
44          loss = None
45          if closure is not None:
46              loss = closure()
47
48          for group in self.param_groups:
49              for p in group['params']:
50                  if p.grad is None:
51                      continue
52                  grad = p.grad.data
53                  if grad.is_sparse:
54                      raise RuntimeError('Adadelta does not support sparse gradients')
55                  state = self.state[p]
56
57
58                  rho, eps = group['rho'], group['eps']
59
60                  if group['weight_decay'] != 0:
61                      grad = grad.add(group['weight_decay'], p.data)
62
63                  state['square_avg'].mul_(rho).addcmul_(1 - rho, grad, grad)
64                  std = state['square_avg'].add(eps).sqrt_()
65
66                  if group['lr'] == 1:
67                      delta = state['acc_delta'].add(eps).sqrt_().div_(std).mul_(grad)
68                      state['acc_delta'].mul_(rho).addcmul_(1 - rho, delta, delta)
69                  else:
70                      delta = 1 / std * grad #1.div_(std).mul_(grad)
71
72                  p.data.add_(-group['lr'], delta)
73
74          return loss
```

3.6 Adam

Adam 是目前最为常用的算法，但是它需要 4 倍的和参数 θ 大小的内存，同时拥有 3 个超参。通常取 β_1 为 0.9, β_2 为 0.999, 而 ϵ 取值为 10^{-8}。

$$
\begin{cases}
g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\
m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)(g_t)^2 \\
\hat{m}_t \leftarrow \dfrac{m_t}{1-(\beta_1)^t} \\
\hat{v}_t \leftarrow \dfrac{v_t}{1-(\beta_2)^t} \\
\theta \leftarrow \theta - \dfrac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t .
\end{cases} \tag{26}
$$

```python
import math
import torch
from torch.optim.optimizer import Optimizer, required

class Adam(Optimizer):
    """Adam algorithm.
    $$
    \left\{
    \begin{aligned}
        g_t \gets & \frac{ \partial L(\theta_t) }{ \partial \theta } \\
        m_t \gets & \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
        v_t \gets & \beta_2 v_{t-1} + (1 - \beta_2) (g_t)^2 \\
        \hat{m}_t \gets & \frac{m_t}{1 - (\beta_1)^t} \\
        \hat{v}_t \gets & \frac{v_t}{1 - (\beta_2)^t} \\
        \theta \gets & \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t.
    \end{aligned}
    \right.
    $$
    """

    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=
        False):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= betas[0] < 1.0:
            raise ValueError("Invalid beta parameter at index 0: {}".format(betas[0]))
        if not 0.0 <= betas[1] < 1.0:
            raise ValueError("Invalid beta parameter at index 1: {}".format(betas[1]))
        defaults = dict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay, amsgrad=amsgrad
            )
        super(Adam, self).__init__(params, defaults)

        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
```

```
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(p.data)
                state['exp_avg_sq'] = torch.zeros_like(p.data)

    def __setstate__(self, state):
        super(Adam, self).__setstate__(state)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('Adam does not support sparse gradients')
                amsgrad = group['amsgrad']

                state = self.state[p]

                beta1, beta2 = group['betas']
                state['step'] += 1
                bias_correction1 = 1 - beta1 ** state['step']
                bias_correction2 = 1 - beta2 ** state['step']

                if group['weight_decay'] != 0:
                    grad.add_(group['weight_decay'], p.data)

                state['exp_avg'].mul_(beta1).add_(1 - beta1, grad)
                state['exp_avg_sq'].mul_(beta2).addcmul_(1 - beta2, grad, grad)
                denom = (state['exp_avg_sq'].sqrt() / math.sqrt(bias_correction2)).add_(group['
                    eps'])
                exp_avg_hat =  state['exp_avg'] / bias_correction1

                p.data.addcdiv_(-group['lr'], exp_avg_hat, denom)
        return loss
```

3.7 Nadam

Nadam 是采用了 NAG 的思想，对式（**??**）经行了调整，将 $\hat{m}_{t-1}$ 改成了“下一步”的。

$$\begin{cases} g_t \leftarrow \dfrac{\partial L(\theta_t)}{\partial \theta} \\ m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2)(g_t)^2 \\ \hat{m}_t \leftarrow \dfrac{m_t}{1-(\beta_1)^t} \\ \hat{v}_t \leftarrow \dfrac{v_t}{1-(\beta_2)^t} \\ \theta \leftarrow \theta - \dfrac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_t + \dfrac{1-\beta_1}{1-(\beta_1)^t} g_t\right). \end{cases} \tag{27}$$

简要推导下式（**??**）和式（**??**）的关系

$$\begin{aligned} \theta_{t+1} =& \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \\ =& \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\frac{m_t}{1-(\beta_1)^t}\right) \\ =& \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\frac{\beta_1 m_{t-1} + (1-\beta_1) g_t}{1-(\beta_1)^t}\right) \\ =& \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\frac{\beta_1 m_{t-1}}{1-(\beta_1)^t} + \frac{(1-\beta_1) g_t}{1-(\beta_1)^t}\right) \\ =& \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1) g_t}{1-(\beta_1)^t}\right) \end{aligned} \tag{28}$$

将 $\hat{m}_{t-1}$ 改最新的 $\hat{m}_t$，得到

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\left(\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-(\beta_1)^t}.\right) \tag{29}$$

```python
import math
import torch
from torch.optim.optimizer import Optimizer, required

class Nadam(Optimizer):
    """Nadam algorithm.
    $$
    \left\{
    \begin{aligned}
        g_t \gets & \frac{ \partial L(\theta_t) }{ \partial \theta } \\
        m_t \gets & \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
        v_t \gets & \beta_2 v_{t-1} + (1 - \beta_2) (g_t)^2 \\
        \hat{m}_t \gets & \frac{m_t}{1 - (\beta_1)^t} \\
        \hat{v}_t \gets & \frac{v_t}{1 - (\beta_2)^t} \\
        \theta \gets & \theta - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(
            \beta_1 \hat{m}_t + \frac{1 - \beta_1}{1 - (\beta_1)^t} g_t \right).
    \end{aligned}
```

```
    \right.
    $$
    """

    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=
        False):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= betas[0] < 1.0:
            raise ValueError("Invalid beta parameter at index 0: {}".format(betas[0]))
        if not 0.0 <= betas[1] < 1.0:
            raise ValueError("Invalid beta parameter at index 1: {}".format(betas[1]))
        defaults = dict(lr=lr, betas=betas, eps=eps, weight_decay=weight_decay, amsgrad=amsgrad
            )
        super(Nadam, self).__init__(params, defaults)

        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['step'] = 0
                state['exp_avg'] = torch.zeros_like(p.data)
                state['exp_avg_sq'] = torch.zeros_like(p.data)

    def __setstate__(self, state):
        super(Nadam, self).__setstate__(state)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data
                if grad.is_sparse:
                    raise RuntimeError('Nadam does not support sparse gradients.')
                amsgrad = group['amsgrad']

                state = self.state[p]
```

```
                beta1, beta2 = group['betas']
                state['step'] += 1
                bias_correction1 = 1 - beta1 ** state['step']
                bias_correction2 = 1 - beta2 ** state['step']

                if group['weight_decay'] != 0:
                    grad.add_(group['weight_decay'], p.data)

                state['exp_avg'].mul_(beta1).add_(1 - beta1, grad)
                state['exp_avg_sq'].mul_(beta2).addcmul_(1 - beta2, grad, grad)
                denom = (state['exp_avg_sq'].sqrt() / math.sqrt(bias_correction2)).add_(group['
                    eps'])
                exp_avg_hat =  state['exp_avg'] / bias_correction1
                exp_avg_hat = beta1 * exp_avg_hat + (1 - beta1) / bias_correction1 * grad
                p.data.addcdiv_(-group['lr'], exp_avg_hat, denom)
        return loss
```

3.8 SAG(Stochastic Average Gradient)

每次迭代只用一个样本，在第 t 次迭代中，SAG[**?**] 需要记录过去 n 个（$n \leq t$）的梯度。同时在每次迭代中都随机选择一个 $i_t \in \{1, 2, \ldots, n\}$，用当前迭代点的梯度 $\frac{\partial L(\theta, x_k)}{\partial \theta}$ 替换 n 个记录中的第 i_t 个梯度。即

$$d_i^t = \begin{cases} \dfrac{\partial L(\theta, x_k)}{\partial \theta} & if i = i_t, \\ d_i^{t-1} & otherwise. \end{cases} \tag{30}$$

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \left(\frac{1}{n} \sum_{i=1}^{n} d_i^t \right). \tag{31}$$

Algorithm 2 SAG

Require: 目标函数 $\min \frac{1}{n} \sum_{i=0}^{n} l_i(\theta)$ ，学习率 η

Ensure: θ

$d = 0$

for $i = 1; i \leq n; i++$ **do**

$\hat{g}_i = 0$

end for

for $t = 0; t < EPOCH; t++$ **do**

sample i from $\{1, 2, \ldots, n\}$

$d = d - \hat{g}_i + \frac{\partial L}{\partial \theta}$

$\hat{g}_i = \frac{\partial L}{\partial \theta}$

$\theta = \theta - \frac{\eta}{n} d$

end for

SAG 比随机梯度下降快得多，是线性收敛算法。但是它要求损失函数是光滑的，而且目标函数是凸函数。而且需要存储不少历史梯度信息，用在非凸的神经网络上不太合适。

```
import math
import random
import torch
from torch.optim.optimizer import Optimizer, required

class SAG(Optimizer):
    """Stochastic Average Gradient.
    """

    def __init__(self, params, lr=1e-2, old_grad_N = 10, weight_decay=0):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        self.lold_grad_N = old_grad_N
        defaults = dict(lr = lr, weight_decay = weight_decay)
        super(SAG, self).__init__(params, defaults)

        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['old_grad'] = torch.zeros(torch.Size([self.lold_grad_N]) + p.data.size())
                state['d'] = torch.zeros_like(p.data)

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            loss = closure()

        r = random.randrange(0, self.lold_grad_N, 1)

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
                grad = p.grad.data

                if grad.is_sparse:
                    raise RuntimeError('ASGD does not support sparse gradients')
                state = self.state[p]
```

```
47
48                 if group['weight_decay'] != 0:
49                     grad = grad.add(group['weight_decay'], p.data)
50
51                 if(torch.cuda.is_available()):
52                     state['d'] = state['d'] - state['old_grad'][r].cuda() + grad
53                 else:
54                     state['d'] = state['d'] - state['old_grad'][r] + grad
55
56                 state['old_grad'][r] = grad
57                 p.data = p.data - group['lr'] / self.lold_grad_N * state['d']
58
59         return loss
```

3.9 SVRG(Stochastic Variance Reduction Gradient)

SVRG[**?**] 是对 SAG 的改进。同样需要保存前面的 n 次梯度，在第 t 次迭代中，随机取一个 $i_t \in \{1, 2, \ldots, n\}$，取代 n 个中的第 i_t 个。除此之外，每迭代 m 次，更新 $\tilde{\theta}$，可以用最后一次的 θ，也可以是 m 次中随机的一次。

$$\begin{cases} \tilde{\mu} \leftarrow \dfrac{1}{n} \sum_{i=0}^{n} g_i(\tilde{\theta}) \\ \theta_t \leftarrow \theta_{t-1} - \eta \left(g_{i_t}(\theta_{t-1}) - g_{i_t}(\tilde{\theta}) + \tilde{\mu} \right) \end{cases} \tag{32}$$

Algorithm 3 SVRG

Require: 目标函数 $\min \frac{1}{n} \sum_{i=0}^{n} l_i(\theta)$，更新频率 m ，学习率 η

Ensure: θ

Initialize $\hat{\theta}$

for $s = 1; s < EPOCH; s++$ **do**

$\hat{\theta} = \hat{\theta}_{s-1}$

$\hat{\mu} = \frac{1}{n} \sum_{i=0}^{n} \frac{\partial l_i}{\partial \hat{\theta}}$

$\theta_0 = \hat{\theta}$

for $t = 1; t \le m; t++$ **do**

Randomly pick $i_t \in \{1, 2, \ldots, n\}$

$\theta_t = \theta_{t-1} - \eta \left(\frac{\partial l_{i_t}}{\partial \theta_{t-1}} - \frac{\partial l_{i_t}}{\partial \hat{\mu}} + \hat{\mu} \right)$

end for

OPTION I: set $\hat{\theta}_s = \theta_m$

OPTION II: set $\hat{\theta}_s = \theta_t$, randomly chosen $t \in \{1, 2, \ldots, m-1\}$

end for

```
1 import math
```

```
import torch
from torch.optim.optimizer import Optimizer, required

class SVRG(Optimizer):
    """Stochastic Variance Reduced Gradient.
    """

    def __init__(self, params, lr=1e-2, weight_decay=0, epoch = 10, batch_size = None):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))

        self.epoch = epoch
        self.batch_size = batch_size
        defaults = dict(lr = lr, weight_decay = weight_decay)
        super(SVRG, self).__init__(params, defaults)

        # State initialization
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['old_grad'] = torch.zeros_like(p.data)
                state['mu'] = torch.zeros_like(p.data)

    def save_grad(self):
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['old_grads'] = p.grad.data.clone()

    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is None:
            raise ValueError("Invalid closure function")

        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['mu'] = p.grad.data.clone()

        for t in range(self.epoch):
            closure()
            for group in self.param_groups:
```

```
                for p in group['params']:
                    if p.grad is None:
                        continue
                    grad = p.grad.data
                    if grad.is_sparse:
                        raise RuntimeError('SVRD does not support sparse gradients')
                    state = self.state[p]
                    if group['weight_decay'] != 0:
                        grad = grad.add(group['weight_decay'], p.data)
                    p.data = p.data - group['lr'] * ( grad - state['old_grad'] + state['mu'])
        # return loss
```

3.10 小结

4 二阶导信息

4.1 共轭梯度法

Algorithm 4 用于正定二次函数的共轭梯度法（Conjugate Gradient）

Require: $f(\vec{x}) = \frac{1}{2}\vec{x}^T Q\vec{x} + \vec{b}^T\vec{x} + \vec{c}$, H 终止准则

Ensure: 极小值点 $\vec{x}^*$

set $\vec{x}_0$, $\vec{p}_0 = -\nabla f(\vec{x}_0)$, $k = 0$

for $k = 0; k < n; k++$ **do**

$t_k = \frac{\nabla f(\vec{x}_k)^T \nabla f(\vec{x}_k)}{\vec{p}_k^T Q\vec{p}_k}$

$\vec{x}_{k+1} = \vec{x}_k + t_k\vec{p}_k$

if $\|\nabla f(\vec{x}_{k+1})\| < \epsilon$ **then**

return $\vec{x}^* = \vec{x}_{k+1}$

else

$a_k = \frac{\nabla f(\vec{x}_{k+1})^T Q\vec{p}_k}{\vec{p}_k^T Q\vec{p}_k}$

$\vec{p}_{k+1} = -\nabla f(\vec{x}_{k+1}) + a_k\vec{p}_k$

end if

$k = k + 1$

end for

Algorithm 5 Fletcher-Reeves 共轭梯度法

Require: $L = \sum_{i=0}^{n} l_i$, 初始 θ_0

Ensure: θ

for $t = 1; t \leq m; t++$ **do**

$L_0 = L(\theta_0), \quad g_0 = g(\theta_0), \quad p_0 = -g_0, \quad k = 0$

for $k = 0; k < n; k++$ **do**

$\theta_{k+1} = ls(\theta_k, p_k), \quad L_{k+1} = L(\theta_{k+1}), \quad g_{k+1} = g(\theta_{k+1})$

if 满足 H 准则 **then**

return $\theta = \theta_{k+1}$

else

if $k == n$ **then**

$\theta_0 = \theta_{k+1}$ break;

else

$\alpha_k = \frac{\|g_{k+1}\|^2}{\|g_k\|^2}, \quad p_{k+1} = -g_{k+1} + \alpha_k p_k$

if $|p_{k+1}^T g_{k+1}| \leq \epsilon$ **then**

$\theta_0 = \theta_{k+1}$ break;

else

if $p_{k+1}^T g_{k+1} \geq \epsilon$ **then**

$p_{k+1} = -p_{k+1}$

end if

end if

end if

end if

$k = k + 1$

end for

end for

4.2 牛顿法

Algorithm 6 Newton 法 (Newton Method)

Require: 目标函数 $f(\vec{x})$ 及其梯度 $G(\vec{x})$,Hesse 矩阵度 $G(\vec{x})$,H 终止准则

Ensure: 极小值点 $\vec{x}^*$

set $\vec{x}_0$, $k = 0$, $f_0 = f(\vec{x}_0)$, $G_0 = G(\vec{x}_0)$
for $k = 0; k < n; k++$ **do**
  $G_k = G(\vec{x}_k)$
  求解 $G_k \vec{p}_k = -G_k$
  $\vec{x}_{k+1} = \vec{x}_k + \vec{p}_k$
  $f_{k+1} = f(\vec{x}_{k+1})$, $G_{k+1} = G(\vec{x}_{k+1})$
  if H 终止准则 **then**
    return $\vec{x}^* = \vec{x}_{k+1}$
  else
    $k = k + 1$
  end if
end for

4.3 拟牛顿法

Algorithm 7 拟 Newton 法 (Quasi-Newton Method)

Require: 目标函数 $f(\vec{x})$ 及其梯度 $\vec{g}(\vec{x})$, H 终止准则

Ensure: 极小值点 $\vec{x}^*$

set $\vec{x}_0$ 和初始对称正定矩阵 H_0, $k = 0$, $f_0 = f(\vec{x}_0)$, $\vec{g}_0 = \vec{g}(\vec{x}_0)$
for $k = 0; k < n; k++$ **do**
  $\vec{x}_{k+1} = ls(\vec{x}_k, -H_k \vec{g}_k)$
  $f_{k+1} = f(\vec{x}_{k+1})$, $\vec{g}_{k+1} = \vec{g}(\vec{x}_{k+1})$
  $\vec{s}_k = \vec{x}_{k+1} - \vec{x}_k$, $\vec{y}_k = \vec{g}_{k+1} - \vec{g}_k$
  if H 终止准则 **then**
    return $\vec{x}^* = \vec{x}_{k+1}$
  else
    $H_{k+1} = H_k + E_k$
    $k = k + 1$
  end if
end for

DFP 校正公式

$$H_{k+1} = H_k + \frac{\vec{s}_k \vec{s}_k^T}{\vec{s}_k^T \vec{y}_k} - \frac{H_k \vec{y}_k \vec{y}_k^T H_k}{\vec{y}_k^T H_k \vec{y}_k}. \tag{33}$$

BFGS 校正公式

$$H_{k+1} = H_k + \frac{1}{\vec{s}_k^T \vec{y}_k}\left[\left(1 + \frac{\vec{y}_k^T H_k \vec{y}_k}{\vec{s}_k^T \vec{y}_k}\right)\vec{s}_k \vec{s}_k^T - H_k \vec{y}_k \vec{s}_k^T - \vec{s}_k \vec{y}_k^T H_k\right]. \tag{34}$$

Broyden 校正公式

$$\begin{cases} H_{k+1} = H_k + \dfrac{\vec{s}_k \vec{s}_k^T}{\vec{y}_k^T \vec{s}_k} - \dfrac{H_k \vec{y}_k \vec{y}_k^T H_k}{\vec{y}_k^T H_k y_k} + \beta\left(\vec{y}_k^T \vec{s}_k\right)\left(\vec{y}_k^T H_k \vec{y}_k\right)\vec{w}_k \vec{w}_k^T \\ \vec{w}_k = \dfrac{\vec{s}_k}{\vec{y}_k^T \vec{s}_k} - \dfrac{H_k \vec{y}_k}{\vec{y}_k^T H_k y_k}. \end{cases} \tag{35}$$

Broyden 校正公式（**??**）中的 β 取值为 0 时得到 DFP 校正公式（**??**），取值为 $\frac{1}{\vec{s}^T \vec{y}}$ 则是 BFGS 校正公式（**??**）。

另外，当 $\beta \geq 0$，且 H_0 为正定阵时，则 H_k 每一个都是正定的。

4.4 随机拟牛顿法

Algorithm 8 SQN Framework

Require: θ_0, V, m, η_t

Ensure: θ

for $t = 0; t < EPOCH; t++$ **do**

$s_t^{'} = H_t g_t$ using the two-loop recursion.

$s_t = -\eta_t s_t^{'}$

$\theta_{t+1} = \theta_t + s_t^{'}$

if update pairs **then**

Compute s_t and u_t

Add a new displacement pair s_t, u_t to V

if $|V| > m$ **then**

Remove the eldest pair from V

end if

end if

end for

Algorithm 9 Two-Lopp Recursion for $H_t g_t$

Require: $\nabla f_t, u_t, s_t$

Ensure: $H_{t+1} g_{t+1}$

$g_t = \nabla f_t$

$H_t^0 = \frac{s_t^T u_t}{\|u_t\|^2} I$

for $l = t-1; l \geq t-p; l++$ **do**

$\eta_l = \rho_l s_l^T g_{l+1}$

$g_l = g_{l+1} - \eta_l u_l$

end for

$r_{t-p-1} = H_t^0 g_{t-p}$

for $l = t-p; l \geq t-1; l--$ **do**

$\beta_l = \rho_l u_l^T \rho_{l-1}$

$\rho_l = \rho_{l-1} + s_l(\eta_t - \beta_l)$

end for

$H_{t+1} g_{t+1} = \rho$

4.5 Hessian Free Method

Algorithm 10 Hessian-Free Optimization Method

Require: $\theta_0, \nabla f(\theta_0), \lambda$

Ensure: θ

for $t = 0; t < EPOCH; t++$ **do**

$g_t = \nabla f(\theta_t)$

Compute λ by some methods

$B_t(v) = H(\theta_t)v + \lambda v$

Compute the step size η_t

$d_t = CG(B_t, -g_t)$

$\theta_{t+1} = \theta_t + \eta_t d_t$

end for

4.6 Sub-sampled Hessian Free Method

S 是整个样本的小样本集，在第 t 次迭代中，随机梯度可以估计为

$$\nabla L_{S_t}(\theta_t) = \frac{1}{|S_t|} \sum_{i \in S_t} \nabla l_i(\theta_t). \tag{36}$$

随机 Hessian 估计

$$\nabla^2 L_{S_t^H}(\theta_t) = \frac{1}{|S_t^H|} \sum_{i \in S_t^H} \nabla^2 l_i(\theta_t). \tag{37}$$

求解下降方向

$$\nabla^2 L_{S_t^H}(\theta_t) d_t = -\nabla L_{S_t}(\theta_t). \tag{38}$$

更新策略为

$$\theta \leftarrow \theta + \eta d \tag{39}$$

4.7 Natural Gradient

$$\begin{cases} \nabla_N L \leftarrow \left(\mathbb{E}_\theta \left[\dfrac{\partial^2 L}{\partial \theta^2} \right] \right)^{-1} \dfrac{\partial L}{\partial \theta} \\ \theta \leftarrow \theta - \eta \nabla_N L. \end{cases} \tag{40}$$

5 其他类型优化方法

6 学习率 η 问题

7 批量大小问题