收藏 | 深度学习优化算法：从SGD到AdamW原理和代码解读

计算机视觉联盟 Yesterday

计算机视觉联盟

团队成员为国内各大高校博士，专注于深度学习、机器学习、图像解译、人工...

115篇原创内容

Official Account

转载于：作者丨科技猛兽

编辑丨极市平台

本文思想来自下面这篇大佬的文章：

Juliuszh：一个框架看懂优化算法之异同 SGD/AdaGrad/Adam

https://zhuanlan.zhihu.com/p/32230623

主要是对深度学习各种优化器 (从SGD到AdamW) 使用统一的框架做一次整理，本文相比于链接从源代码的角度理解这些优化器的思路。

代码来自 PyTorch1.7.0 官方教程：

https://pytorch.org/docs/1.7.0/optim.html

首先我们来回顾一下各类优化算法。

深度学习优化算法经历了 SGD -> SGDM -> NAG ->AdaGrad -> AdaDelta -> Adam -> Nadam -> AdamW 这样的发展历程。Google一下就可以看到很多的教程文章，详细告诉你这些算法是如何一步一步演变而来的。在这里，我们换一个思路，用一个框架来梳理所有的优化算法，做一个更加高屋建瓴的对比。

- **统一框架：**

首先定义：待优化参数：w ，目标函数：$f(w)$ ，初始学习率 α。

而后，开始进行迭代优化。在每个epoch t ：

1 计算目标函数关于当前参数的梯度：

$$g_t = \nabla f(w_t) \tag{1}$$

2 根据历史梯度计算一阶动量和二阶动量：

$$m_t = \phi(g_1, g_2, \cdots, g_t); V_t = \psi(g_1, g_2, \cdots, g_t) \tag{2}$$

3 计算当前时刻的下降梯度：

$$\eta_t = \alpha \cdot m_t / \sqrt{V_t} \tag{3}$$

4 根据下降梯度进行更新：

$$w_{t+1} = w_t - \eta_t \tag{4}$$

掌握了这个框架，你可以轻轻松松设计自己的优化算法。

我们拿着这个框架，来照一照各种玄乎其玄的优化算法的真身。步骤3, 4对于各个算法都是一致的，主要的差别就体现在1和2上，也就是计算一阶动量 m_t 和二阶动量 V_t 时采用不同的套路。当计算好二者之后，都是使用固定的学习率 α 与二者作用得到当前时刻的下降梯度 η_t ，进而最后更新参数。

在所有优化器的代码里面有一些函数的作用是相通的：

> **共性的方法有：**

- `add_param_group` (param_group)：把参数放进优化器中，这在 Fine-tune 预训练网络时很有用，因为可以使冻结层可训练并随着训练的进行添加到优化器中。
- `load_state_dict` (state_dict)：把优化器的状态加载进去。
- `state_dict` ()：返回优化器的状态，以dict的形式返回。
- `step` (closure=None)：优化一步参数。
- `zero_grad` (set_to_none=False)：把所有的梯度值设为0。

使用方法：

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

下面正式开始。

SGD

先来看SGD。SGD没有动量的概念，也就是说：

$$m_t = g_t; V_t = I^2 \tag{5}$$

代入步骤3，可以看到下降梯度就是最简单的

$$\eta_t = \alpha \cdot g_t \tag{6}$$

SGD最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。

SGD with Momentum

为了抑制SGD的震荡，SGDM认为梯度下降过程可以加入惯性。下坡的时候，如果发现是陡坡，那就利用惯性跑的快一些。SGDM全称是SGD with momentum，在SGD基础上引入了一阶动量：

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \tag{7}$$

一阶动量是各个时刻梯度方向的指数移动平均值，约等于最近 $1/(1-\beta_1)$ 个时刻的梯度向量和的平均值。

也就是说， t 时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。β_1 的经验值为0.9，这就意味着下降方向主要是此前累积的下降方向，并略微偏向当前时刻的下降方向。想象高速公路上汽车转弯，在高速向前的同时略微偏向，急转弯可是要出事的。

SGD with Nesterov Acceleration

SGD 还有一个问题是困在局部最优的沟壑里面震荡。想象一下你走到一个盆地，四周都是略高的小山，你觉得没有下坡的方向，那就只能待在这里了。可是如果你爬上高地，就会发现外面的世界还很广阔。因此，我们不能停留在当前位置去观察未来的方向，而要向前一步、多看一步、看远一些。

NAG全称Nesterov Accelerated Gradient，是在SGD、SGD-M的基础上的进一步改进，改进点在于步骤1。我们知道在时刻 t 的主要下降方向是由累积动量决定的，自己的梯度方向说了也不算，那与其看当前梯度方向，不如先看看如果跟着累积动量走了一步，那个时候再怎么走。因此，NAG在步骤1，不计算当前位置的梯度方向，而是计算如果按照累积动量走了一步，那个时候的下降方向：

$$g_t = \nabla f(w_t - \beta_1 \cdot m_{t-1} / \sqrt{V_{t-1}}) \tag{8}$$

然后用下一个点的梯度方向，与历史累积动量相结合，计算步骤2中当前时刻的累积动量。

> **定义优化器：**

```
CLASS torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weig
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate，相当于是统一框架中的 α 。
- **momentum** (float, optional) – 动量参数。(默认值：0)
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **dampening** (float, optional) – dampening for momentum (默认值：0)
- **nesterov** (bool, optional) – 允许 Nesterov momentum (默认值：False)

FLOAT：https://docs.python.org/3/library/functions.html#float bool:https://docs.python.org/3/library/functions.html#bool

源码解读：

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


[docs]class SGD(Optimizer):
    r"""Implements stochastic gradient descent (optionally with momentum).

    Nesterov momentum is based on the formula from
    `On the importance of initialization and momentum in deep learning`__.

    Args:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float): learning rate
        momentum (float, optional): momentum factor (default: 0)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        dampening (float, optional): dampening for momentum (default: 0)
        nesterov (bool, optional): enables Nesterov momentum (default: False)

    Example:
        >>> optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
        >>> optimizer.zero_grad()
        >>> loss_fn(model(input), target).backward()
        >>> optimizer.step()

    __ http://www.cs.toronto.edu/%7Ehinton/absps/momentum.pdf

    .. note::
        The implementation of SGD with Momentum/Nesterov subtly differs from
        Sutskever et. al. and implementations in some other frameworks.

        Considering the specific case of Momentum, the update can be written as

        .. math::
            \begin{aligned}
                v_{t+1} & = \mu * v_{t} + g_{t+1}, \\
                p_{t+1} & = p_{t} - \text{lr} * v_{t+1},
            \end{aligned}

        where :math:`p`, :math:`g`, :math:`v` and :math:`\mu` denote the
        parameters, gradient, velocity, and momentum respectively.

        This is in contrast to Sutskever et. al. and
        other frameworks which employ an update of the form

        .. math::
            \begin{aligned}
```

```
                v_{t+1} & = \mu * v_{t} + \text{lr} * g_{t+1}, \\
                p_{t+1} & = p_{t} - v_{t+1}.
            \end{aligned}

        The Nesterov version is analogously modified.
    """

    def __init__(self, params, lr=required, momentum=0, dampening=0,
                 weight_decay=0, nesterov=False):
        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, nesterov=nesterov)
        if nesterov and (momentum <= 0 or dampening != 0):
            raise ValueError("Nesterov momentum requires a momentum and zero dampeni
        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)

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            weight_decay = group['weight_decay']
            momentum = group['momentum']
            dampening = group['dampening']
            nesterov = group['nesterov']

            for p in group['params']:
                if p.grad is None:
                    continue
                d_p = p.grad
                if weight_decay != 0:
                    d_p = d_p.add(p, alpha=weight_decay)
                if momentum != 0:
                    param_state = self.state[p]
                    if 'momentum_buffer' not in param_state:
                        buf = param_state['momentum_buffer'] = torch.clone(d_p).deta
                    else:
                        buf = param_state['momentum_buffer']
                        buf.mul_(momentum).add_(d_p, alpha=1 - dampening)
                    if nesterov:
                        d_p = d_p.add(buf, alpha=momentum)
                    else:
                        d_p = buf

                p.add_(d_p, alpha=-group['lr'])

        return loss
```

> 这里通过 d_p=p.grad 得到每个参数的梯度，也就是1式的 g_t 。
> 如果使用 weight_decay 的话，那么相当于目标函数加上 $\frac{1}{2}\lambda||W||^2$ ，所以相当于是梯度相当于要再加上 λW ，所以使用了 d_p = d_p.add(p, alpha=weight_decay)。
> 通过 buf.mul_(momentum).add_(d_p, alpha=1 - dampening) 来计算动量，momentum参数 β_1 一般取0.9，就相当于是之前的动量buf乘以 $\beta_1 = 0.9$ ，再加上此次的梯度d_p乘以 $(1 - \beta_1) = 0.1$ 。
> 如果不通过nesterov方式更新参数，那么3式中的 η_t 就相当于是上一步计算出的动量 m_t 了。如果通过nesterov方式更新参数，那么3式中的 η_t 就相当于$g_t + m_t * \beta_1$ ，和不用nesterov方式相比，相差了 。
> 最后通过 p.add_(d_p, alpha=-group['lr']) 更新梯度，相当于是上面的 3 式。

AdaGrad

此前我们都没有用到二阶动量。二阶动量的出现，才意味着“自适应学习率”优化算法时代的到来。SGD及其变种以同样的学习率更新每个参数，但深度神经网络往往包含大量的参数，这些参数并不是总会用得到（想想大规模的embedding）。对于经常更新的参数，我们已经积累了大量关于它的知识，不希望被单个样本影响太大，希望学习速率慢一些；对于偶尔更新的参数，我们了解的信息太少，希望能从每个偶然出现的样本身上多学一些，即学习速率大一些。

怎么样去度量历史更新频率呢？那就是二阶动量——该维度上，迄今为止所有梯度值的平方和：

$$V_t = \sum_{\tau=1}^{t} g_\tau^2 \tag{9}$$

我们再回顾一下步骤3中的下降梯度：

$$\eta_t = \alpha \cdot m_t / \sqrt{V_t} \tag{3}$$

可以看出，此时实质上的学习率由 α 变成了 $\alpha/\sqrt{V_t}$ 。一般为了避免分母为0，会在分母上加一个小的平滑项。因此$\sqrt{V_t}$ 是恒大于0的，而且参数更新越频繁，二阶动量越大，学习率就越小。

这一方法在稀疏数据场景下表现非常好。但也存在一些问题：因为$\sqrt{V_t}$ 是单调递增的，会使得学习率单调递减至0，可能会使得训练过程提前结束，即便后续还有数据也无法学到必要的知识。

> **定义优化器：**

```
CLASS torch.optim.Adagrad(params,lr=0.01,lr_decay=0,weight_decay=0,initial_accumulat
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate – 相当于是统一框架中的 α 。
- **lr_decay**(float,optional) – 学习率衰减 (默认值：0)
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **eps**(float,optional)：防止分母为0的一个小数 (默认值：1e-10)

> **源码解读：**

```
[docs]class Adagrad(Optimizer):
    """Implements Adagrad algorithm.

    It has been proposed in `Adaptive Subgradient Methods for Online Learning
    and Stochastic Optimization`_.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-2)
        lr_decay (float, optional): learning rate decay (default: 0)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        eps (float, optional): term added to the denominator to improve
```

```
            numerical stability (default: 1e-10)

    .. _Adaptive Subgradient Methods for Online Learning and Stochastic
        Optimization: http://jmlr.org/papers/v12/duchi11a.html
    """

    def __init__(self, params, lr=1e-2, lr_decay=0, weight_decay=0, initial_accumula
        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(in
        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, initial_accumulator_value, memory_

    def share_memory(self):
        for group in self.param_groups:
            for p in group['params']:
                state = self.state[p]
                state['sum'].share_memory_()

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            params_with_grad = []
            grads = []
            state_sums = []
            state_steps = []

            for p in group['params']:
                if p.grad is not None:
                    params_with_grad.append(p)
                    grads.append(p.grad)
                    state = self.state[p]
                    state_sums.append(state['sum'])
                    # update the steps for each param group update
                    state['step'] += 1
                    # record the step after step update
                    state_steps.append(state['step'])

            F.adagrad(params_with_grad,
                      grads,
                      state_sums,
                      state_steps,
                      group['lr'],
                      group['weight_decay'],
                      group['lr_decay'],
                      group['eps'])

        return loss
```

AdaDelta / RMSProp

由于AdaGrad单调递减的学习率变化过于激进，我们考虑一个改变二阶动量计算方法的策略：不累积全部历史梯度，而只关注过去一段时间窗口的下降梯度。这也就是AdaDelta名称中Delta的来历。

修改的思路很简单。前面我们讲到，指数移动平均值大约就是过去一段时间的平均值，因此我们用这一方法来计算二阶累积动量：

$$V_t = \beta_2 * V_{t-1} + (1 - \beta_2) g_t^2 \tag{10}$$

接下来还是步骤3：

$$\eta_t = \alpha \cdot g_t / \sqrt{V_t} \tag{11}$$

这就避免了二阶动量持续累积、导致训练过程提前结束的问题了。

RMSProp

> **定义优化器：**

```
CLASS torch.optim.RMSprop(params, lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, mo
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate – 相当于是统一框架中的 α 。
- **momentum** (float, optional) – 动量参数。(默认值：0)。
- **alpha**(*float,optional*) – 平滑常数 (默认值：0.99)。
- **centered**(bool,optional) – if `True` , compute the centered RMSProp, the g radient is normalized by an estimation of its variance，就是这一项是 True 的话就把方差使用梯度作归一化。
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **eps**(float,optional)：防止分母为0的一个小数 (默认值：1e-10)

> **源码解读：**

```
import torch
from .optimizer import Optimizer


[docs]class RMSprop(Optimizer):
    r"""Implements RMSprop algorithm.

    Proposed by G. Hinton in his
    `course <https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pd

    The centered version first appears in `Generating Sequences
    With Recurrent Neural Networks <https://arxiv.org/pdf/1308.0850v5.pdf>`_.

    The implementation here takes the square root of the gradient average before
    adding epsilon (note that TensorFlow interchanges these two operations). The eff
    learning rate is thus :math:`\alpha/(\sqrt{v} + \epsilon)` where :math:`\alpha`
    is the scheduled learning rate and :math:`v` is the weighted moving average
    of the squared gradient.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-2)
        momentum (float, optional): momentum factor (default: 0)
        alpha (float, optional): smoothing constant (default: 0.99)
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        centered (bool, optional) : if ``True``, compute the centered RMSProp,
            the gradient is normalized by an estimation of its variance
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

    """

    def __init__(self, params, lr=1e-2, alpha=0.99, eps=1e-8, weight_decay=0, moment
        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 <= momentum:
            raise ValueError("Invalid momentum value: {}".format(momentum))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        if not 0.0 <= alpha:
            raise ValueError("Invalid alpha value: {}".format(alpha))

        defaults = dict(lr=lr, momentum=momentum, alpha=alpha, eps=eps, centered=cen
        super(RMSprop, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(RMSprop, self).__setstate__(state)
        for group in self.param_groups:
            group.setdefault('momentum', 0)
            group.setdefault('centered', False)

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

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

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    state['square_avg'] = torch.zeros_like(p, memory_format=torch.pr
                    if group['momentum'] > 0:
                        state['momentum_buffer'] = torch.zeros_like(p, memory_format
                    if group['centered']:
                        state['grad_avg'] = torch.zeros_like(p, memory_format=torch.

                square_avg = state['square_avg']
                alpha = group['alpha']

                state['step'] += 1

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

                square_avg.mul_(alpha).addcmul_(grad, grad, value=1 - alpha)

                if group['centered']:
                    grad_avg = state['grad_avg']
                    grad_avg.mul_(alpha).add_(grad, alpha=1 - alpha)
                    avg = square_avg.addcmul(grad_avg, grad_avg, value=-1).sqrt_().a
                else:
                    avg = square_avg.sqrt().add_(group['eps'])

                if group['momentum'] > 0:
                    buf = state['momentum_buffer']
                    buf.mul_(group['momentum']).addcdiv_(grad, avg)
                    p.add_(buf, alpha=-group['lr'])
                else:
                    p.addcdiv_(grad, avg, value=-group['lr'])

        return loss
```

> 这里通过 grad = p.grad 得到每个参数的梯度，也就是1式的 g_t 。
> 如果使用 weight_decay 的话，那么相当于目标函数加上 $\frac{1}{2}\lambda||W||^2$ ，所以相当于是梯度相当于要再加上 λW ，故使用了 grad = grad.add(p, alpha=group['weight_decay'])。
> square_avg.mul_(alpha).addcmul_(grad, grad, value=1 - alpha) 对应10式，计算当前步的 V_t 。
> centered 这一项是 False 的话直接 square_avg.sqrt().add_(group['eps']) 对 V_t 开根号。
> centered 这一项是 True 的话就把方差使用梯度作归一化。
> 最后通过 p.addcdiv_(grad, avg, value=-group['lr']) 更新梯度，相当于是上面的 3 式。
> RMSprop算是Adagrad的一种发展，和Adadelta的变体，效果趋于二者之间

AdaDelta

> **定义优化器：**

```
CLASS torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06, weight_decay=0)
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate – 相当于是统一框架中的 α 。
- **rho**(float,optional) – 计算梯度平方的滑动平均超参数 (默认值：0.9)
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **eps**(float,optional)：防止分母为0的一个小数 (默认值：1e-10)

> **源码解读：**

```
import torch

from .optimizer import Optimizer


[docs]class Adadelta(Optimizer):
    """Implements Adadelta algorithm.

    It has been proposed in `ADADELTA: An Adaptive Learning Rate Method`__.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        rho (float, optional): coefficient used for computing a running average
            of squared gradients (default: 0.9)
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-6)
        lr (float, optional): coefficient that scale delta before it is applied
            to the parameters (default: 1.0)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)

    __ https://arxiv.org/abs/1212.5701
    """

    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)

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

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

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    state['square_avg'] = torch.zeros_like(p, memory_format=torch.pr
                    state['acc_delta'] = torch.zeros_like(p, memory_format=torch.pre

                square_avg, acc_delta = state['square_avg'], state['acc_delta']
                rho, eps = group['rho'], group['eps']

                state['step'] += 1

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

                square_avg.mul_(rho).addcmul_(grad, grad, value=1 - rho)
                std = square_avg.add(eps).sqrt_()
                delta = acc_delta.add(eps).sqrt_().div_(std).mul_(grad)
                p.add_(delta, alpha=-group['lr'])
                acc_delta.mul_(rho).addcmul_(delta, delta, value=1 - rho)

        return loss
```

> 这里通过 grad = p.grad 得到每个参数的梯度，也就是1式的 g_t 。
> 如果使用 weight_decay 的话，那么相当于目标函数加上 $\frac{1}{2}\lambda||W||^2$ ，所以相当于是梯度相当于要再加上 λW ，故使用了 grad = grad.add(p, alpha=group['weight_decay'])。
> square_avg.mul_(rho).addcmul_(grad, grad, value=1 - rho) 对应10式，计算当前步的 V_t 。std = square_avg.add(eps).sqrt_() 对 V_t 开根号。
> 最后通过 p.add_(delta, alpha=-group['lr']) 更新梯度，相当于是上面的 3 式。
> delta 的分子项是 g_t ，分母项是 V_t 开根号。acc_delta 是对 delta 的滑动平均。

Adam

谈到这里，Adam和Nadam的出现就很自然而然了——它们是前述方法的集大成者。我们看到，SGD-M在SGD基础上增加了一阶动量，AdaGrad和AdaDelta在SGD基础上增加了二阶动量。把一阶动量和二阶动量都用起来，就是Adam了——Adaptive + Momentum。

SGD的一阶动量：

加上AdaDelta的二阶动量：

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{14}$$

$$\hat{V}_t = \frac{V_t}{1-\beta_2^t} \tag{15}$$

优化算法里最常见的两个超参数 β_1, β_2 就都在这里了，前者控制一阶动量，后者控制二阶动量。

Nadam

最后是Nadam。我们说Adam是集大成者，但它居然遗漏了Nesterov，这还能忍？必须给它加上，按照NAG的步骤1：

这就是Nesterov + Adam = Nadam了。

> **定义优化器：**

```
CLASS torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate – 相当于是统一框架中的 α 。
- **betas**(Tuple[float,float],optional) – coefficients used for computing running averages of gradient and its square ((默认值：(0.9, 0.999))
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **eps**(float,optional)：防止分母为0的一个小数 (默认值：1e-10)

> **源码解读：**

```
import math
import torch
from .optimizer import Optimizer


[docs]class Adam(Optimizer):
    r"""Implements Adam algorithm.

    It has been proposed in `Adam: A Method for Stochastic Optimization`_.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-3)
        betas (Tuple[float, float], optional): coefficients used for computing
            running averages of gradient and its square (default: (0.9, 0.999))
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        weight_decay (float, optional): weight decay (L2 penalty) (default: 0)
        amsgrad (boolean, optional): whether to use the AMSGrad variant of this
            algorithm from the paper `On the Convergence of Adam and Beyond`_
            (default: False)
    .. _Adam\: A Method for Stochastic Optimization:
        https://arxiv.org/abs/1412.6980
    .. _On the Convergence of Adam and Beyond:
        https://openreview.net/forum?id=ryQu7f-RZ
    """

    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]
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        defaults = dict(lr=lr, betas=betas, eps=eps,
                        weight_decay=weight_decay, amsgrad=amsgrad)
        super(Adam, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(Adam, self).__setstate__(state)
        for group in self.param_groups:
            group.setdefault('amsgrad', False)

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

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

                state = self.state[p]

                # State initialization
                if len(state) == 0:
                    state['step'] = 0
                    # Exponential moving average of gradient values
                    state['exp_avg'] = torch.zeros_like(p, memory_format=torch.prese
                    # Exponential moving average of squared gradient values
                    state['exp_avg_sq'] = torch.zeros_like(p, memory_format=torch.pr
                    if amsgrad:
                        # Maintains max of all exp. moving avg. of sq. grad. values
                        state['max_exp_avg_sq'] = torch.zeros_like(p, memory_format=

                exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
                if amsgrad:
                    max_exp_avg_sq = state['max_exp_avg_sq']
                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 = grad.add(p, alpha=group['weight_decay'])

                # Decay the first and second moment running average coefficient
                exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
                exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)
                if amsgrad:
                    # Maintains the maximum of all 2nd moment running avg. till now
                    torch.max(max_exp_avg_sq, exp_avg_sq, out=max_exp_avg_sq)
                    # Use the max. for normalizing running avg. of gradient
                    denom = (max_exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).ad
                else:
                    denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(g

                step_size = group['lr'] / bias_correction1

                p.addcdiv_(exp_avg, denom, value=-step_size)

        return loss
```

> 这里通过 grad = p.grad 得到每个参数的梯度，也就是1式的 g_t 。
> 如果使用 weight_decay 的话，那么相当于目标函数加上 $\frac{1}{2}\lambda||W||^2$ ，所以相当于是梯度相当于要再加上 λW ，故使用了 grad = grad.add(p, alpha=group['weight_decay'])。
> exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1) 计算12式。
> exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2) 计算13式。
> 因为15式的缘故，要给分母除以 math**.**sqrt(bias_correction2)。
> 因为14式的缘故，要给分子除以 bias_correction1。
> 最后通过 p.addcdiv_(exp_avg, denom, value=-step_size) 更新梯度，相当于是上面的 3 式。

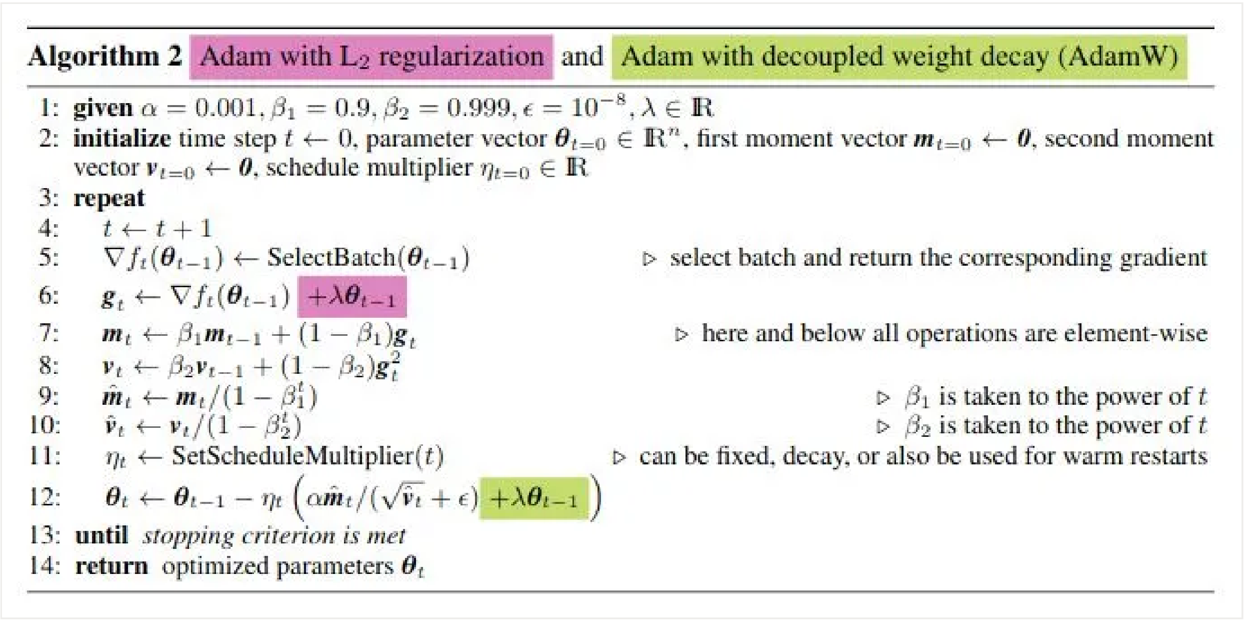
AdamW

下图1所示为Adam的另一个改进版：AdamW。

简单来说，AdamW就是Adam优化器加上L2正则，来限制参数值不可太大，这一点属于机器学习入门知识了。以往的L2正则是直接加在损失函数上，比如这样子：加入正则，损失函数就会变成这样子：

所以在计算梯度 g_t 时要加上粉色的这一项。

但AdamW稍有不同，如下图所示，将正则加在了绿色位置。



Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

```
1:  given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
2:  initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment
    vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3:  repeat
4:      $t \leftarrow t + 1$
5:      $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$      ▷ select batch and return the corresponding gradient
6:      $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda\boldsymbol{\theta}_{t-1}$
7:      $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$      ▷ here and below all operations are element-wise
8:      $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
9:      $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t/(1 - \beta_1^t)$      ▷ $\beta_1$ is taken to the power of $t$
10:     $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t/(1 - \beta_2^t)$      ▷ $\beta_2$ is taken to the power of $t$
11:     $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$      ▷ can be fixed, decay, or also be used for warm restarts
12:     $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left( \alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda\boldsymbol{\theta}_{t-1} \right)$
13: until stopping criterion is met
14: return optimized parameters $\boldsymbol{\theta}_t$
```

图1：AdamW

至于为何这么做？直接摘录BERT里面的原话看看：

> **Just** adding the square of the weights to the loss function is *not* the correct way of using L2 regularization/weight decay with Adam, since that will interact with the m and v parameters in strange ways. Instead we want to decay the weights in a manner that doesn't interact with the m/v parameters. This is equivalent to adding the square of the weights to the loss with plain (non-momentum) SGD. Add weight decay at the end (fixed version).

这段话意思是说，如果直接将L2正则加到loss上去，由于Adam优化器的后序操作，该正则项将会与m_t和v_t产生奇怪的作用。因而，AdamW选择将L_2正则项加在了Adam的m_t和v_t等参数被计算完之后、在与学习率η相乘之前，所以这也表明了weight_decay和L_2正则虽目的一致、公式一致，但用法还是不同，二者有着明显的差别。以 PyTorch1.7.0 中的AdamW代码为例：

> **定义优化器：**

```
CLASS torch.optim.AdamW(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_deca
```

> **参数：**

- **params** (iterable) – 优化器作用的模型参数。
- **lr** (float) – learning rate – 相当于是统一框架中的 α 。
- **betas**(Tuple[float,float],optional) – coefficients used for computing running averages of gradient and its square ((默认值：(0.9, 0.999))
- **weight_decay** (float, optional) – 权重衰减系数 weight decay (L2 penalty) (默认值：0)
- **eps**(float,optional)：防止分母为0的一个小数 (默认值：1e-10)

> **源码解读：**

```
import math
import torch
from .optimizer import Optimizer


[docs]class AdamW(Optimizer):
    r"""Implements AdamW algorithm.

    The original Adam algorithm was proposed in `Adam: A Method for Stochastic Optim
    The AdamW variant was proposed in `Decoupled Weight Decay Regularization`_.

    Arguments:
        params (iterable): iterable of parameters to optimize or dicts defining
            parameter groups
        lr (float, optional): learning rate (default: 1e-3)
        betas (Tuple[float, float], optional): coefficients used for computing
            running averages of gradient and its square (default: (0.9, 0.999))
        eps (float, optional): term added to the denominator to improve
            numerical stability (default: 1e-8)
        weight_decay (float, optional): weight decay coefficient (default: 1e-2)
        amsgrad (boolean, optional): whether to use the AMSGrad variant of this
            algorithm from the paper `On the Convergence of Adam and Beyond`_
            (default: False)

    .. _Adam\: A Method for Stochastic Optimization:
        https://arxiv.org/abs/1412.6980
    .. _Decoupled Weight Decay Regularization:
        https://arxiv.org/abs/1711.05101
    .. _On the Convergence of Adam and Beyond:
        https://openreview.net/forum?id=ryQu7f-RZ
    """

    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8,
                 weight_decay=1e-2, 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]
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        defaults = dict(lr=lr, betas=betas, eps=eps,
                        weight_decay=weight_decay, amsgrad=amsgrad)
        super(AdamW, self).__init__(params, defaults)

    def __setstate__(self, state):
        super(AdamW, self).__setstate__(state)
        for group in self.param_groups:
            group.setdefault('amsgrad', False)

[docs]    @torch.no_grad()
    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:
            with torch.enable_grad():
                loss = closure()

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

```
                continue

            # Perform stepweight decay
            p.mul_(1 - group['lr'] * group['weight_decay'])

            # Perform optimization step
            grad = p.grad
            if grad.is_sparse:
                raise RuntimeError('Adam does not support sparse gradients, plea
            amsgrad = group['amsgrad']

            state = self.state[p]

            # State initialization
            if len(state) == 0:
                state['step'] = 0
                # Exponential moving average of gradient values
                state['exp_avg'] = torch.zeros_like(p, memory_format=torch.prese
                # Exponential moving average of squared gradient values
                state['exp_avg_sq'] = torch.zeros_like(p, memory_format=torch.pr
                if amsgrad:
                    # Maintains max of all exp. moving avg. of sq. grad. values
                    state['max_exp_avg_sq'] = torch.zeros_like(p, memory_format=

            exp_avg, exp_avg_sq = state['exp_avg'], state['exp_avg_sq']
            if amsgrad:
                max_exp_avg_sq = state['max_exp_avg_sq']
            beta1, beta2 = group['betas']

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

            # Decay the first and second moment running average coefficient
            exp_avg.mul_(beta1).add_(grad, alpha=1 - beta1)
            exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)
            if amsgrad:
                # Maintains the maximum of all 2nd moment running avg. till now
                torch.max(max_exp_avg_sq, exp_avg_sq, out=max_exp_avg_sq)
                # Use the max. for normalizing running avg. of gradient
                denom = (max_exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).ad
            else:
                denom = (exp_avg_sq.sqrt() / math.sqrt(bias_correction2)).add_(g

            step_size = group['lr'] / bias_correction1

            p.addcdiv_(exp_avg, denom, value=-step_size)

    return loss
```

> 与 Adam 不一样的地方是：
> Adam 如果使用 weight_decay 的话，那么相当于目标函数加上 $1/2\gamma||\theta||^2$ ，所以相

> 当于是梯度相当于要再加上 $\gamma\theta$ ，故使用了 grad = grad.add(p, alpha=group['weight_decay'])。
> 而 AdamW 是 p.mul_(1 - group['lr'] * group['weight_decay']) 直接让参数：
> $\theta_t = \theta_{t-1} - \alpha \cdot \lambda \cdot \theta_{t-1} - \alpha \cdot \eta_t$
> 这样才能和绿色框一致

-------------------- ***END*** --------------------