Dropout、梯度消失/爆炸、Adam优化算法，神经网络优化算法看这一篇就够了

作者 | mantch
来源 | 知乎

1. 训练误差和泛化误差

对于机器学习模型在训练数据集和测试数据集上的表现。如果你改变过实验中的模型结构或者超参数，你也许发现了：当模型在训练数据集上更准确时，它在测试数据集上却不一定更准确。这是为什么呢？

因为存在着训练误差和泛化误差：

- **训练误差：模型在训练数据集上表现出的误差。**
- **泛化误差：模型在任意一个测试数据样本上表现出的误差的期望，并常常通过测试数据集上的误差来近似。**

训练误差的期望小于或等于泛化误差。也就是说，一般情况下，由训练数据集学到的模型参数会使模型在训练数据集上的表现优于或等于在测试数据集上的表现。**由于无法从训练误差估计泛化误差，一味地降低训练误差并不意味着泛化误差一定会降低。**

机器学习模型应关注降低泛化误差。

2. 该如何选择模型

在机器学习中，通常需要评估若干候选模型的表现并从中选择模型。这一过程称为模型选择（model selection）。可供选择的候选模型可以是有着不同超参数的同类模型。以多层感知机为例，我们可以选择隐藏层的个数，以及每个隐藏层中隐藏单元个数和激活函数。为了得到有效的模型，我们通常要在模型选择上下一番功夫。

2.1 验证数据集

从严格意义上讲，测试集只能在所有超参数和模型参数选定后使用一次。不可以使用测试数据选择模型，如调参。由于无法从训练误差估计泛化误差，因此也不应只依赖训练数据选择模型。鉴于此，我们可以预留一部分在训练数据集和测试数据集以外的数据来进行模型选择。这部分数据被称为验证数据集，简称验证集（validation set）。例如，我们可以从给定的训练集中随机选取一小部分作为验证集，而将剩余部分作为真正的训练集。

可以通过预留这样的验证集来进行模型选择，判断验证集在模型中的表现能力。

2.2 K 折交叉验证

由于验证数据集不参与模型训练，当训练数据不够用时，预留大量的验证数据显得太奢侈。一种改善的方法是K折交叉验证（K-fold cross-validation）。在K折交叉验证中，我们把原始训练数据集分割成K个不重合的子数据集，然后我们做K次模型训练和验证。每一次，我们使用一个子数据集验证模型，并使用其他K － 1个子数据集来训练模型。在这K次训练和验证中，每次用来验证模型的子数据集都不同。最后，我们对这K次训练误差和验证误差分别求平均。

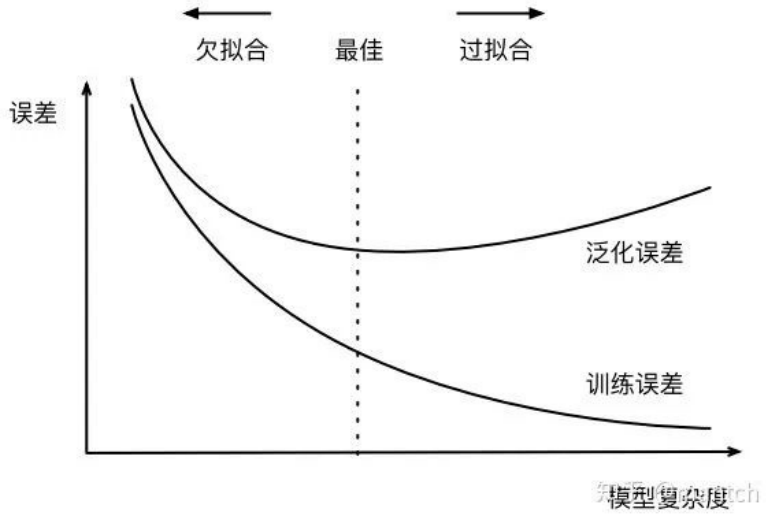
3. 欠拟合和过拟合

- **欠拟合：模型无法得到较低的训练误差。**
- **过拟合：是模型的训练误差远小于它在测试数据集上的误差。**

给定训练数据集，

- 如果模型的复杂度过低，很容易出现欠拟合；
- 如果模型复杂度过高，很容易出现过拟合。

应对欠拟合和过拟合的一个办法是针对数据集选择合适复杂度的模型。



训练数据集大小

影响欠拟合和过拟合的另一个重要因素是训练数据集的大小。一般来说，如果训练数据集中样本数过少，特别是比模型参数数量（按元素计）更少时，过拟合更容易发生。此外，泛化误差不会随训练数据集里样本数量增加而增大。因此，在计算资源允许的范围之内，我们通常希望训练数据集大一些，特别是在模型复杂度较高时，例如层数较多的深度学习模型。

正则化

应对过拟合问题的常用方法：权重衰减（weight decay），权重衰减等价于L2范数正则化（regularization）。正则化通过为模型损失函数添加惩罚项使学出的模型参数值较小，是应对过拟合的常用手段。

4. 丢弃法(Dropout)

除了上面提到的权重衰减以外，深度学习模型常常使用丢弃法（dropout）来应对过拟合问题。丢弃法有一些不同的变体。本节中提到的丢弃法特指倒置丢弃法（inverted dropout）。

回忆一下，“多层感知机”描述了一个单隐藏层的多层感知机。其中输入个数为4，隐藏单元个数为5，且隐藏单元hi（i = 1, . . . , 5）的计算表达式为：

$$h_i = \phi(x_1 w_{1i} + x_2 w_{2i} + x_3 w_{3i} + x_4 w_{4i} + b_i$$

这里ϕ是激活函数，x1, . . . , x4是输入，隐藏单元i的权重参数为w1i, . . . , w4i，偏差参数为bi。当对该隐藏层使用丢弃法时，该层的隐藏单元将有一定概率被丢弃掉。设丢弃概率为p，那么有p的概率hi会被清零，有1 − p的概率hi会除以1 − p做拉伸。丢弃概率是丢弃法的超参数。具体来说，设随机变量ξi为0和1的概率分别为p和1 − p。使用丢弃法时我们计算新的隐藏单元

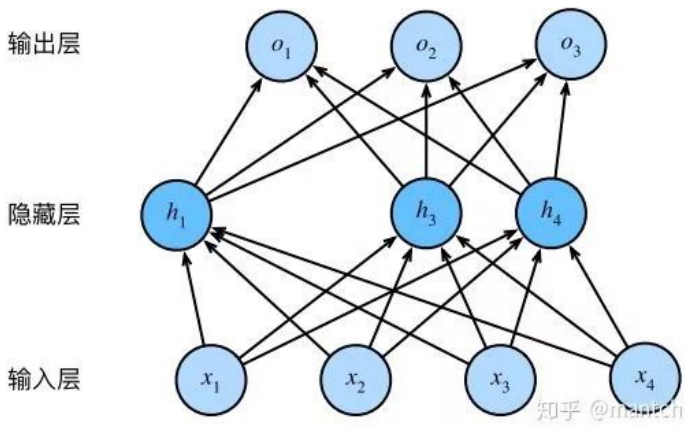
$$h_i'$$

$$h_i' = \frac{\xi_i}{1-p}$$

由于E(ξi) = 1 − p，因此：

$$E(h_i') = \frac{E(\xi_i)}{1-p} h_i = h_i$$

即丢弃法不改变其输入的期望值。让我们对隐藏层使用丢弃法，一种可能的结果如下图所示，其中h2和h5被清零。这时输出值的计算不再依赖h2和h5，在反向传播时，与这两个隐藏单元相关的权重的梯度均为0。由于在训练中隐藏层神经元的丢弃是随机的，即h1, . . . , h5都有可能被清零，输出层的计算无法过度依赖h1, . . . , h5中的任一个，从而在训练模型时起到正则化的作用，并可以用来应对过拟合。在测试模型时，我们为了拿到更加确定性的结果，一般不使用丢弃法。



5. 梯度消失/梯度爆炸（Vanishing / Exploding gradients）

训练神经网络，尤其是深度神经所面临的一个问题就是梯度消失或梯度爆炸，也就是你训练神经网络的时候，导数或坡度有时会变得非常大，或者非常小，甚至于以指数方式变小，这加大了训练的难度。
本质上，梯度消失和爆炸是一种情况。在深层网络中，由于网络过深，如果初始得到的梯度过小，或者传播途中在某一层上过小，则在之后的层上得到的梯度会越来越小，即产生了梯度消失。梯度爆炸也是同样的。一般地，不合理的初始化以及激活函数，如sigmoid等，都会导致梯度过大或者过小，从而引起消失/爆炸。

解决方案

1、预训练加微调

其基本思想是每次训练一层隐节点，训练时将上一层隐节点的输出作为输入，而本层隐节点的输出作为下一层隐节点的输入，此过程就是逐层“预训练”（pre-training）；在预训练完成后，再对整个网络进行“微调”（fine-tunning）。
此方法有一定的好处，但是目前应用的不是很多了。

2、梯度剪切、正则

梯度剪切这个方案主要是针对梯度爆炸提出的，其思想是设置一个梯度剪切阈值，然后更新梯度的时候，如果梯度超过这个阈值，那么就将其强制限制在这个范围之内。这可以防止梯度爆炸。
另外一种解决梯度爆炸的手段是采用权重正则化（weithts regularization）比较常见的是L1和L2正则。

3、ReLu、leakReLu等激活函数

ReLu：其函数的导数在正数部分是恒等于1，这样在深层网络中，在激活函数部分就不存在导致梯度过大或者过小的问题，缓解了梯度消失或者爆炸。同时也方便计算。当然，其也存在存在一些缺点，例如过滤到了负数部分，导致部分信息的丢失，输出的数据分布不在以0为中心，改变了数据分布。
leakrelu：就是为了解决relu的0区间带来的影响，其数学表达为：leakrelu=max(k*x,0)其中k是leak系数，一般选择0.01或者0.02，或者通过学习而来。

4、Batch Normalization

Batch Normalization是深度学习发展以来提出的最重要的成果之一了，目前已经被广泛的应用到了各大网络中，具有加速网络收敛速度，提升训练稳定性的效果，Batch Normalization本质上是解决反向传播过程中的梯度问题。Batch Normalization，简称BN，即批规范化，通过规范化操作将输出信号x规范化到均值为0，方差为1保证网络的稳定性。

- 有一些从 0 到 1 而不是从 1 到 1000 的特征值，通过归一化所有的输入特征值x，以获得类似范围的值，可以加速学习。所以 Batch 归一化起的作用的原因，直观的一点就是，它在做类似的工作，但不仅仅对于这里的输入值，还有隐藏单元的值。
- 它可以使权重比你的网络更滞后或更深层，比如，第 10 层的权重更能经受得住变化。
- **残差结构**

残差的方式，能使得深层的网络梯度通过跳级连接路径直接返回到浅层部分，使得网络无论多深都能将梯度进行有效的回传。

LSTM

LSTM全称是长短期记忆网络（long-short term memory networks），是不那么容易发生梯度消失的，主要原因在于LSTM内部复杂的“门”(gates)。在计算时，将过程中的梯度进行了抵消。

6. 随机梯度下降法(SGD)

6.1 mini-batch梯度下降

你可以把训练集分割为小一点的子集训练，这些子集被取名为 mini-batch，假设每一个子集中只有 1000 个样本，那么把其中的x (1)到x (1000)取出来，将其称为第一个子训练集，也叫做 mini-batch，然后你再取出接下来的 1000 个样本，从x (1001)到x (2000)，然后再取 1000个样本，以此类推。

在训练集上运行 mini-batch 梯度下降法，你运行 for t=1……5000，因为我们有5000个各有 1000 个样本的组，在 for 循环里你要做得基本就是对X {t}和Y {t}执行一步梯度下降法。

- batch_size=1，就是SGD。
- batch_size=n，就是mini-batch
- batch_size=m，就是batch

其中1<n<m，m表示整个训练集大小。

优缺点：

- batch：相对噪声低些，幅度也大一些，你可以继续找最小值。
- SGD：大部分时候你向着全局最小值靠近，有时候你会远离最小值，因为那个样本恰好给你指的方向不对，因此随机梯度下降法是有很多噪声的，平均来看，它最终会靠近最小值，不过有时候也会方向错误，因为随机梯度下降法永远不会收敛，而是会一直在最小值附近波动。一次性只处理了一个训练样本，这样效率过于低下。
- mini-batch：实践中最好选择不大不小的 **mini-batch**，得到了大量向量化，效率高，收敛快。

首先，如果训练集较小，直接使用 batch 梯度下降法，这里的少是说小于 2000 个样本。一般的 mini-batch 大小为 64 到 512，考虑到电脑内存设置和使用的方式，如果 mini-batch 大小是 2 的n次方，代码会运行地快一些。

6.2 调节 Batch_Size 对训练效果影响到底如何？

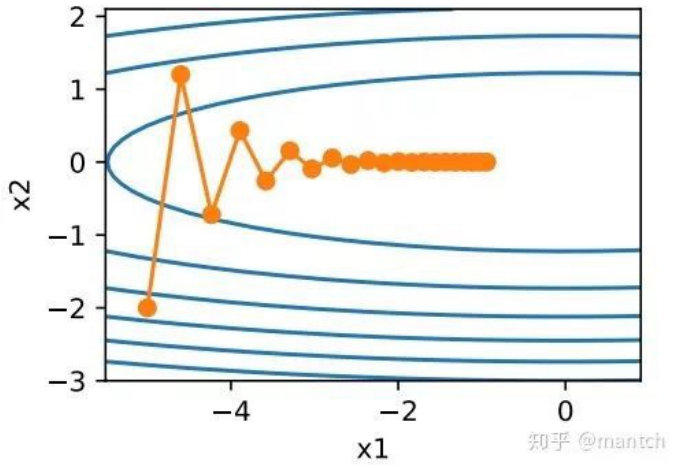
1. Batch_Size 太小，模型表现效果极其糟糕(error飙升)。
2. 随着 Batch_Size 增大，处理相同数据量的速度越快。
3. 随着 Batch_Size 增大，达到相同精度所需要的 epoch 数量越来越多。
4. 由于上述两种因素的矛盾， Batch_Size 增大到某个时候，达到时间上的最优。
5. 由于最终收敛精度会陷入不同的局部极值，因此 Batch_Size 增大到某些时候，达到最终收敛精度上的最优。

7. 优化算法

7.1 动量法

在每次迭代中，梯度下降根据自变量当前位置，沿着当前位置的梯度更新自变量。然而，如果自变量的 迭代方向仅仅取决于自变量当前位置，这可能会带来一些问题。

让我们考虑一个输入和输出分别为二维向量x = [x1, x2]T和标量的目标函数
$f(x$
=0.1x_1^2+2x_2^2)。这里将
x_1^2
系数从1减小到了0.1。下面实现基于这个目标函数的梯度下降，并演示使用学习率为0.4时自变量的迭代轨迹。



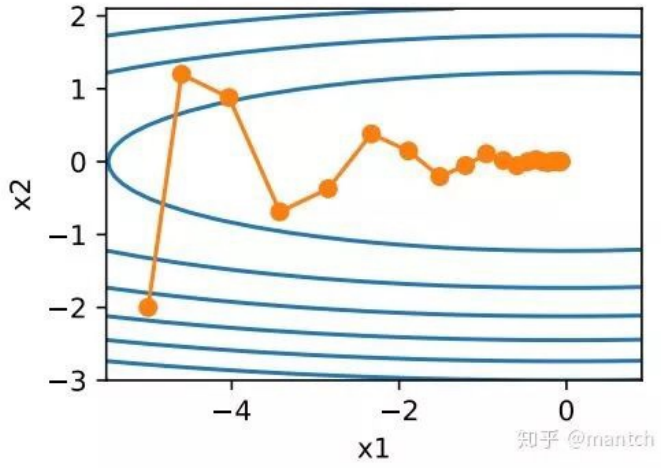
可以看到，同一位置上，目标函数在竖直方向（x2轴方向）比在水平方向（x1轴方向）的斜率的绝对值更大。**因此，给定学习率，梯度下降迭代自变量时会使自变量在竖直方向比在水平方向移动幅度更大。**那么，我们需要一个较小的学习率从而避免自变量在竖直方向上越过目标函数最优解。然而，这会造成自变量在水平方向上朝最优解移动变慢。

动量法的提出是为了解决梯度下降的上述问题。由于小批量随机梯度下降比梯度下降更为广义，本章后续讨论将沿用“小批量随机梯度下降”一节中时间步t的小批量随机梯度gt的定义。设时间步t的自变量为xt，学习率为ηt。在时间步0，动量法创建速度变量v0，并将其元素初始化成0。在时间步t > 0，动量法对每次迭代的步骤做如下修改：

$$v_t \leftarrow \gamma v_{t-1} + \eta_t g_t$$

$$x_t \leftarrow x_{t-1} - v_t$$

其中，动量超参数γ满足0 ≤ γ < 1。当γ = 0时，动量法等价于小批量随机梯度下降。在梯度下降时候使用动量法后的迭代轨迹：



可以看到使用较小的学习率η = 0.4和动量超参数γ = 0.5时，动量法在竖直方向上的移动更加平滑，且在水平方向上更快逼近最优解。

所以，在动量法中，自变量在各个方向上的移动幅度不仅取决当前梯度，还取决于过去的各个梯度在各个方向上是否一致。在本节之前示例的优化问题中，所有梯度在水平方向上为正（向右），而在竖直方向上时正（向上）时负（向下）。这样，我们就可以使用较大的学习率，从而使自变量向最优解更快移动。

7.2 AdaGrad算法

优化算法中，目标函数自变量的每一个元素在相同时间步都使用同一个学习率来自我迭代。在“动量法”里我们看到当x1和x2的梯度值有较大差别时，需要选择足够小的学习率使得自变量在梯度值较大的维度上不发散。但这样会导致自变量在梯度值较小的维度上迭代过慢。动量法依赖指数加权移动平均使得自变量的更新方向更加一致，从而降低发散的可能。**本节我们介绍AdaGrad算法，它根据自变量在每个维度的梯度值的大小来调整各个维度上的学习率，从而避免统一的学习率难以适应所有维度的问题。**

AdaGrad算法会使用一个小批量随机梯度gt按元素平方的累加变量st。在时间步0，AdaGrad将s0中每个元素初始化为0。在时间步t，首先将小批量随机梯度gt按元素平方后累加到变量st：

$$s_t = s_{t-1} + g_t \odot g_t$$

其中⊙是按元素相乘。接着，我们将目标函数自变量中每个元素的学习率通过按元素运算重新调整一下：

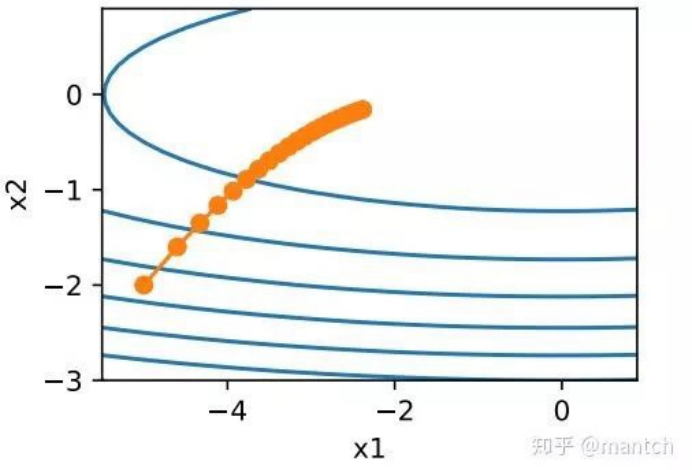
$$x_t = x_{t-1} - \frac{\eta}{\sqrt{s_t + \varepsilon}} \odot g_t$$

其中 η 是学习率，ϵ是为了维持数值稳定性而添加的常数，如10的-6次方。这里开方、除法和乘法的运算都是按元素运算的。这些按元素运算使得目标函数自变量中每个元素都分别拥有自己的学习率。

需要强调的是，小批量随机梯度按元素平方的累加变量st出现在学习率的分母项中。因此，

- **如果目标函数有关自变量中某个元素的偏导数一直都较大，那么该元素的学习率将下降较快；**
- **反之，如果目标函数有关自变量中某个元素的偏导数一直都较小，那么该元素的学习率将下降较慢。**

然而，由于st一直在累加按元素平方的梯度，自变量中每个元素的学习率在迭代过程中一直在降低（或不变）。**所以，当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad算法在迭代后期由于学习率过小，可能较难找到一个有用的解。**



7.3 RMSProp算法

当学习率在迭代早期降得较快且当前解依然不佳时，AdaGrad算法在迭代后期由于学习率过小，可能较难找到一个有用的解。为了解决这一问题，RMSProp算法对AdaGrad算法做了一点小小的修改。

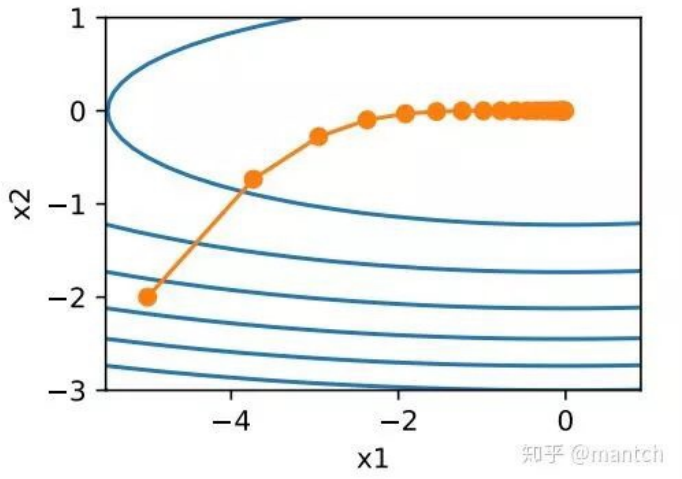
不同于AdaGrad算法里状态变量st是截至时间步t所有小批量随机梯度gt按元素平方和，RMSProp算法将这些梯度按元素平方做指数加权移动平均。具体来说，给定超参数0 ≤ γ < 1，RMSProp算法在时间步t > 0计算：

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t \odot g_t$$

和AdaGrad算法一样，RMSProp算法将目标函数自变量中每个元素的学习率通过按元素运算重新调整，然后更新自变量：

$$x_t = x_{t-1} - \frac{\eta}{\sqrt{s_t + \varepsilon}} \odot g_t$$

其中η是学习率，ϵ是为了维持数值稳定性而添加的常数，如10的-6次方。因为RMSProp算法的状态变量st是对平方项gt ⊙ gt的指数加权移动平均，所以可以看作是最近1/(1 − γ)个时间步的小批量随机梯度平方项的加权平均。如此一来，自变量每个元素的学习率在迭代过程中就不再一直降低（或不变）。



7.4 AdaDelta算法

除了RMSProp算法以外，另一个常用优化算法AdaDelta算法也针对AdaGrad算法在迭代后期可能较难找到有用解的问题做了改进。有意思的是，AdaDelta算法没有学习率这一超参数。

AdaDelta算法也像RMSProp算法一样，使用了小批量随机梯度gt按元素平方的指数加权移动平均变量st。在时间步0，它的所有元素被初始化为0。给定超参数0 ≤ ρ < 1（对应RMSProp算法中的γ），在时间步t > 0，同RMSProp算法一样计算：

$$s_t = p s_{t-1} + (1-p) g_t \odot g_t$$

与RMSProp算法不同的是，AdaDelta算法还维护一个额外的状态变量Δxt，其元素同样在时间步0时被初始化为0。我们使用Δxt-1来计算自变量的变化量：

$$g_t' = \sqrt{\frac{\Delta x_{t-1} + \varepsilon}{s_t + \varepsilon}} \odot g_t$$

最后，我们使用Δxt来记录自变量变化量

g'_t

按元素平方的指数加权移动平均：

$$\Delta x_t = p\Delta x_{t-1} + (1-p)g'_t \odot g'_t$$

可以看到，如不考虑ϵ的影响，AdaDelta算法与RMSProp算法的不同之处在于使用

$\sqrt{\Delta x_{t-1}}$

来替代超参数η。

7.5 Adam算法

Adam算法在RMSProp算法基础上对小批量随机梯度也做了指数加权移动平均。

Adam算法使用了动量变量vt和RMSProp算法中小批量随机梯度按元素平方的指数加权移动平均变量st，并在时间步0将它们中每个元素初始化为0。给定超参数0 ≤ β1 < 1（算法作者建议设为0.9），时间步t的动量变量vt即小批量随机梯度gt的指数加权移动平均：

$$v_t = \beta_1 v_{t-1} + (1-\beta_1$$

和RMSProp算法中一样，给定超参数0 ≤ β2 < 1（算法作者建议设为0.999），将小批量随机梯度按元素平方后的项gt ⊙ gt做指数加权移动平均得到st：

$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t \odot g_t$$

由于我们将 v0 和 s0 中的元素都初始化为 0，在时间步 t 我们得到
$v_t = (1-\beta_1$
sum_{i=1}^teta_1^{t-i}g_i)。将过去各时间步小批量随机梯度的权值相加，得到
$(1-\beta_1$
sum_{i=1}^teta_1^{t-i}=1-eta_1^t)。需要注意的是，当 t 较小时，过去各时间步小批量随机梯度权值之和会较小。例如，当β1 = 0.9时，v1 = 0.1g1。为了消除这样的影响，对于任意时间步 t，我们可以将 vt 再除以
$1-\beta_1^t$
，从而使过去各时间步小批量随机梯度权值之和为1。这也叫作偏差修正。在Adam算法中，我们对变量 vt 和 st 均作偏差修正：

$$\check{v}_t = \frac{v_t}{1-\beta_1^t}$$
$$\check{s}_t = \frac{s_t}{1-\beta_2^t}$$

接下来，Adam算法使用以上偏差修正后的变量$\check{v}_t$和$\check{s}_t$，将模型参数中每个元素的学习率通过按元素运算重新调整：

$$g'_t = \frac{\eta \check{v}_t}{\sqrt{\check{s}_t}+\varepsilon}$$

其中 η 是学习率，ϵ是为了维持数值稳定性而添加的常数，如10的-8次方。和AdaGrad算法、RMSProp算法以及AdaDelta算法一样，目标函数自变量中每个元素都分别拥有自己的学习率。最后，使用

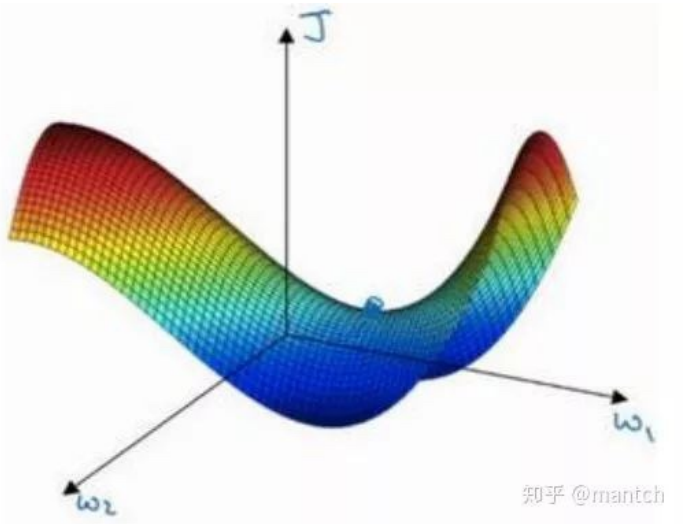
g'_t

迭代自变量：

$$x_t = x_{t-1} - g'_t$$

7.6 局部最优--鞍点问题

一个具有高维度空间的函数，如果梯度为 0，那么在每个方向，它可能是凸函数，也可能是凹函数。如果你在 2 万维空间中，那么想要得到局部最优，所有的 2 万个方向都需要是这样，但发生的机率也许很小，也许是2的-20000次方，你更有可能遇到有些方向的曲线会这样向上弯曲，另一些方向曲线向下弯，而不是所有的都向上弯曲，因此在高维度空间，你更可能碰到鞍点。



而不会碰到局部最优。**至于为什么会把一个曲面叫做鞍点**，你想象一下，就像是放在马背上的马鞍一样，如果这是马，这是马的头，这就是马的眼睛，画得不好请多包涵，然后你就是骑马的人，要坐在马鞍上，因此这里的这个点，导数为 0 的点，这个点叫做鞍点。我想那确实是你坐在马鞍上的那个点，而这里导数为 0。

鞍点中的平稳段是一个问题，这样使得学习十分缓慢，**这也是像 Momentum 或是RMSprop，Adam 这样的算法，能够加速学习算法的地方。**在这些情况下，更成熟的优化算法，如 Adam 算法，能够加快速度，让你尽早往下走出平稳段。

8. 如何解决训练样本少的问题

1. 利用预训练模型进行迁移微调（fine-tuning），预训练模型通常在特征上拥有很好的语义表达。此时，只需将模型在小数据集上进行微调就能取得不错的效果。CV有ImageNet，NLP有BERT等。
2. 数据集进行下采样操作，使得符合数据同分布。
3. 数据集增强、正则或者半监督学习等方式来解决小样本数据集的训练问题。

9. 如何提升模型的稳定性？

1. 正则化（L2, L1, dropout）：模型方差大，很可能来自于过拟合。正则化能有效的降低模型的复杂度，增加对更多分布的适应性。
2. 前停止训练：提前停止是指模型在验证集上取得不错的性能时停止训练。这种方式本质和正则化是一个道理，能减少方差的同时增加的偏差。目的为了平衡训练集和未知数据之间在模型的表现差异。
3. 扩充训练集：正则化通过控制模型复杂度，来增加更多样本的适应性。

4. 特征选择：过高的特征维度会使模型过拟合，减少特征维度和正则一样可能会处理好方差问题，但是同时会增大偏差。

10. 有哪些改善模型的思路

1、数据角度

增强数据集。无论是有监督还是无监督学习，数据永远是最重要的驱动力。更多的类型数据对良好的模型能带来更好的稳定性和对未知数据的可预见性。对模型来说，“看到过的总比没看到的更具有判别的信心”。

2、模型角度

模型的容限能力决定着模型可优化的空间。在数据量充足的前提下，对同类型的模型，增大模型规模来提升容限无疑是最直接和有效的手段。

3、调参优化角度

如果你知道模型的性能为什么不再提高了，那已经向提升性能跨出了一大步。超参数调整本身是一个比较大的问题。一般可以包含模型初始化的配置，优化算法的选取、学习率的策略以及如何配置正则和损失函数等等。

4、训练角度

在越大规模的数据集或者模型上，诚然一个好的优化算法总能加速收敛。但你在未探索到模型的上限之前，永远不知道训练多久算训练完成。所以在改善模型上充分训练永远是最必要的过程。充分训练的含义不仅仅只是增大训练轮数。有效的学习率衰减和正则同样是充分训练中非常必要的手段。

11. 如何提高深度学习系统的性能

1. 提高模型的结构。
2. 改进模型的初始化方式，保证早期梯度具有某些有益的性质，或者具备大量的稀疏性，或者利用线性代数原理的优势。
3. 选择更强大的学习算法。

参考文献

- 动手学--深度学习
- 吴恩达--深度学习笔记
- GitHub
- 百面机器学习

链接：https://zhuanlan.zhihu.com/p/78854514

*(*本文为 AI科技大本营转载文章，转载请联系作者)*

原价~~1099~~元，现只需199元
名额有限

识别二维码，进入大会官网抢购

培训日程

Part 1 深度学习基础

李沐，亚马逊首席科学家

09:00-10:40

1.数据操作
2.自动求导
3.线性回归的从零开始实现
4.线性回归的简洁实现
5.图像分类数据集（Fashion-MNIST）
6.Softmax回归的从零开始实现
7.Softmax回归的简洁实现
8.多层感知机的从零开始实现
9.多层感知机的简洁实现

Part 2 卷积神经网络基础

李沐，亚马逊首席科学家

11:00-12:40

1.使用GPU
2.卷积层
3.池化层
4.卷积神经网络：LeNet
5.深度卷积神经网络：AlexNet

Part 3 现代卷积神经网络

李沐，亚马逊首席科学家

14:00-15:40

1.用重复元素的网络：VGG
2.含并行连结的网络：GoogLeNet
3.残差网络：ResNet
4.多GPU和分布式计算
5.图像增广
6.微调

Part 4 计算机视觉

李沐，亚马逊首席科学家
何通，亚马逊应用科学家

16:00-17:40

1.目标检测模型
2.目标检测预测
3.目标检测训练
4.训练调参技巧
5.部署模型

推荐阅读

- NLP这两年：15个预训练模型对比分析与剖析
- **60+业内技术专家，9大核心技术专题，AI ProCon倒计时一周！**
- **小团队如何玩转物联网开发？**
- **一文看懂机器学习中的常用损失函数**
- **DeepMind提图像生成的递归神经网络DRAW，158行Python代码复现**
- **KDD 2019高维稀疏数据上的深度学习Workshop论文汇总**
- **5G 改变社会的真相在这里！**
- **程序员如何解决并发冲突的难题？**

你点的每个“在看”，我都认真当成了喜欢