2017CS231n李飞飞深度视觉识别笔记（七）——训练神经网络（下）

2018年07月18日 12:30:40 献世online 阅读数：202 更多

第七讲 训练神经网络（下）

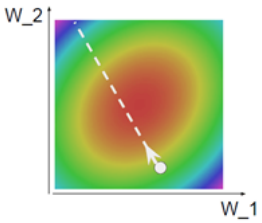
课时1 更好的优化

上一章中讨论了一些激活函数和它们不同的属性，由于sigmoid函数和tanh函数在两端都存在梯度消失的问题，所以在实际中一般建议使用ReLU函数和它的其他形式；同时也讨论了权重初始化，在深度网络变得越来越深的时候，权重的初始化会变得非常重要；在数据的预处理的方面讨论了中心化和归一化的问题，这一章中将继续讨论更多有关训练神经网络的细节问题，最重要的是与之前相比更有效的优化方法、正则化策略以及迁移学习。

回顾之前学习过的内容，训练神经网络的核心策略是一个优化问题，已经使用到的一个优化的方法是随机梯度下降法：

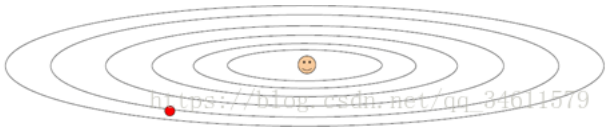
```
while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

它在红色区域收敛，如愿得到了很小的误差值：



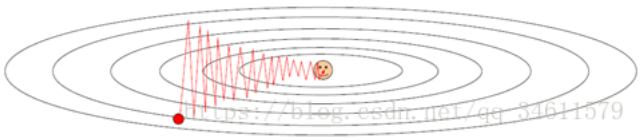
但是不幸的是这个相对简单的优化算法在实际中会产生很多问题。

问题1：



这里画了两个值W1和W2，当改变其中之一时，损失函数变化非常慢；当在水平方向改变值，损失函数变化非常慢；当在等高线图上下运动时，损失值则对竖直方向的变化非常敏感；对于损失值来说，在这点上是很坏的情况，这一点是海森矩阵中最大奇异值和最小奇异值之比，对于这样的函数随机梯度下降法会做什么呢？

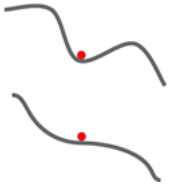
如果在这类函数上运行随机梯度算法，会得到这样特有的之字形图形：



其原因是因为这类目标函数梯度的方向并不是与最小值成一条线，当计算梯度并沿着前进时，可能会一遍遍的跨过这些等高线之字形的前进或后退，所以在水平维度上的前进速度非常慢，在这个方向上的敏感度较低，却在垂直维度上非常敏感，这并不是所期望的结果；而且这样的问题在高维空间变得更加普遍。

问题2：

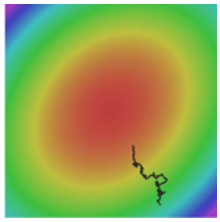
随机梯度法的另一个问题是局部最小值或鞍点，例如下图中：



在局部最小值这种情况下，随机梯度法会卡在中间，因为这里是局部极小值，梯度为0，不会做任何移动；关于鞍点，往一个方向是向上，往另一个方向是向下，在当前的位置，梯度也是0，此时会卡在鞍点处。

问题3：

如果N是整个训练集，每次计算损失都会耗费很大的计算量，事实上经常通过小批量的对损失和梯度进行估计，这意味着并不会每一步去计算真实的梯度，而是在当前点对梯度进行噪声估计，所以在下图中对在每一点的梯度上加入了随机均匀噪声扰乱梯度，在这样的噪声条件下运行随机梯度下降法：



所以这种曲折的线路实际上会花费很长时间才能得到极小值。

对于这些出现的问题，经常会有一个优化的算法来解决，在随机梯度下降中加入一个动量项，如下所示：

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

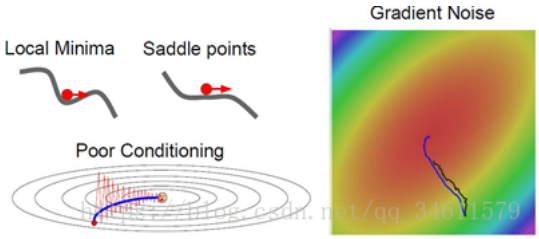
```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

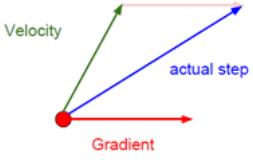

左侧是经典的随机梯度下降法，通常只是在梯度的方向上前进；而在右侧有一个非常小的方差称为带动量的随机梯度下降法，它的思想是保持一个不随时间变化的速度，并且将梯度估计添加到这个速度上，然后在这个速度上前进而不是在梯度的方向上前进，其中有一个超参数是摩擦系数ρ，这个简单的策略解决了刚才讨论的所有问题。



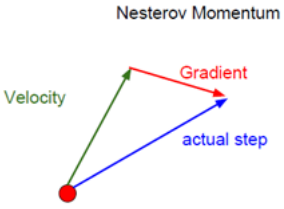
问题：带动量的随机梯度下降如何处理条件很差的坐标？

答：回顾一下速度估计以及速度计算，会发现在每一步都增加了梯度，这一定程度上取决于摩擦系数ρ的设置；如果梯度较小，这种情况下ρ表现很好，那么速度可以单调递增到一个速度会比实际梯度更大的点，然后可能会更快速地处理条件差的维度。

当在使用带动量的随机梯度下降时，可以想象如下图中的图像：



这个红色的点是当前位置，有一些红色向量表示梯度，绿色向量表示速度向量的方向，当做动量更新时，实际上是在这两者的平均权重进行前进，这有助于克服梯度估计中的一些噪声；有时会看到向量有一个轻微的变化，叫Nesterov加速梯度，如下所示：



在普通的随机梯度中，估算当前位置的梯度，然后取速度和梯度的混合；而在Nesterov加速梯度中，从红点开始，在取得速度的方向上进行前进，之后评估这个位置的梯度，随后回到原始位置将这两者混合起来，它的计算如下：

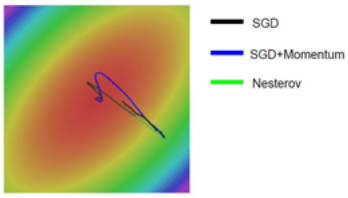
$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

但是这个公式有一些不便之处，在使用随机梯度法优化神经网络时通常希望能够同时计算损失函数和梯度，而Nesterov的动量优化形式会对此造成破坏，所以用换元法进行改进：

Change of variables $x_t = x_t + \rho v_t$ and rearrange:

$$
\begin{aligned}
v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\
\tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\
&= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)
\end{aligned}
$$

下面是三种方法处理的效果：



另一种很好的优化策略是AdaGrad算法，它的核心思想是在优化过程中需要保持一个在训练过程中的每一步梯度的平方和的持续估计，与速度项不同的是有了一个梯度平方项，在训练时会一直累加当前梯度的平方到这个梯度平方项，当更新参数向量时，会除以这个梯度平方项：

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

问题：这样的放缩对矩阵中条件数很大的情形有什么改进呢？

答：如果有两个坐标轴，沿其中一个轴有很高的梯度，而另一个轴方向却有一个很小的梯度，那么随着累加小梯度的平方，会在最后更新参数向量时除以一个很小的数字，加速了在小维度上的学习速度，而在另一个维度上，由于梯度变得特别大，会除以一个很大的数，所以会降低在这个维度方向上的训练进度。

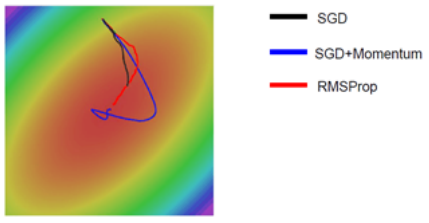
问题：当时间越来越大的时候，在训练过程中使用AdaGrad会发生什么？

答：步长会变得越来越小，因为一直在随时间更新梯度平方的估计值，所以这个估计值在训练过程中一直随时间单调递增，导致步长随时间越来越小。

对于AdaGrad算法有一个变体叫RMSProp，它不是仅仅简单的在训练中累加梯度平方，而是让平方梯度按照一定比率下降。

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

下面是效果比较图：



接下来引入的方法将上面两种的优势之处结合起来叫做Adam算法，使用它更新第一动量和第二动量的估计值：

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

问题：在最初的第一步会发生什么？

答：第一步时，已经将第二动量初始化为0，而第二动量经过一步更新后仍然非常接近0，所以会导致得到一个很大的步长；因此，Adam算法也增加了偏置校正项来避免开始时得到很大步长的问题出现。

```
first_unbias = first_moment / (1 - beta1 ** t)
second_unbias = second_moment / (1 - beta2 ** t)
```

这就得到了Adam算法的完整形式，通常情况下首选用Adam算法，一般将beta1设置为0.9，beta2设置为0.999，学习率为1e-3或者5e-4。

这里要讨论到学习率的问题，刚才提到的优化算法中都离不开学习率这个最重要的超参数，所以如何去选择合适的学习率呢？

答：不必在整个训练的过程中都一直固定使用一个学习率，有时人们会把学习率沿着时间衰减（步长衰减的方法）。

目前讨论过的优化策略都是在降低训练误差和最小化目标函数，现在的问题是怎么做减少训练和测试之间的误差间距以使得模型在没见过的数据上表现的更好呢？

答：一个方法是模型集成，它应用在机器学习的很多领域。比起使用一个模型，可以选择从不同的随机初始值上训练10个不同的模型，到了测试时，在10个模型上运行测试数据，然后平均10个模型的预测结果。

模型集成的小技巧：

（1）有时候可以不用独立地训练不同的模型，可以在训练过程中保留模型的多个快照，然后用这些模型快照来进行集成学习，测试阶段把这些多个快照的预测结果做平均；

（2）在训练模型的时候，对不同时刻的每个模型参数求指数衰减平均值，从而得到网络训练中一个比较平滑的集成模型，之后使用这些平滑衰减的平均后的模型参数而不是截止在某一时刻的模型参数，这个方法叫做Polyak averaging。

课时2 正则化

如何能提高单一模型的效果，使用集成学习时，测试的时候还是要在10个模型基础上学习，这并不是很好；正则化正是能提高单一模型的效果的方法。

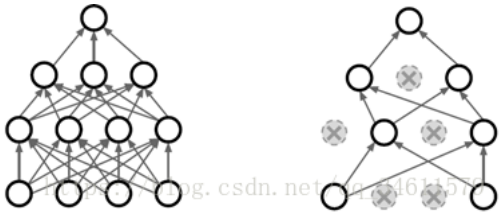
在之前的学习中，已经讨论过一些正则化方法，如下图所示：

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i;W)_j - f(x_i;W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l}\rvert$
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l}\rvert$

一个在神经网络中常用的正则化方法是dropout：每次在网络中正向传递时，在每一层随机将一部分神经元置0（激活函数置位0），每次正向传递时，随机被置0的神经元都不是完全相同的。



如果把左边这个全连接网络和右边经过dropout的版本进行对比，会发现dropout后的网络像是同样的网络变小了一号，只用到了其中一部分神经元，并且每次遍历和正向传递都是不同的部分。实现过程如下：

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3
```

对于dropout的解释：

这是在单一模型中进行集成学习，每一种可能的dropout方式都可以产生一个不同的子网络，所以dropout像是同时对一群共享参数的网络进行集成学习；因此dropout的可能性随神经元个数呈指数倍增长，不可能穷举每种情况，可以看作一个巨大的网络集在同时被训练。

dropout在测试时候想把随机的值平均化，用到的公式：

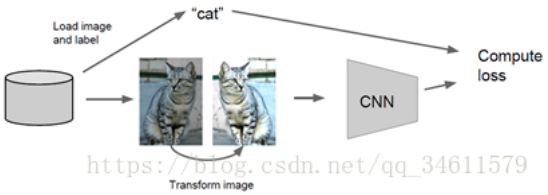
$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

实际上可以用一种省事的方式，局部逼近这个积分，如果考虑单个神经元，输入是x和y以及两个权重，然后测试时得到a的值是；在训练期间，使用了dropout丢弃神经网络单元的概率是0.5，可以算出a的期望值在训练期间为；在测试和训练时两者的结果并不统一，所以这里一种省事的方面就是在测试时没有任何随机性，而是用dropout的概率乘以这个输出；这就有点像简易地局部逼近这个复杂积分。

更通用的正则化策略就是：

在训练期间，给网络加一些随机性以防止过拟合训练数据：；在测试期间，要抵消掉所有的随机性希望能够提高泛化能力。

另一种符合这种范数的策略是数据增强的想法，在训练的时候有一个最初的版本，有自己的数据和自己的标签，每次迭代时使用它去更新卷积神经网络；但在训练过程中能以某种方式随机地转换图像使得标签可以保留不变，使用这些随机转换的图像进行训练：



所以可以从这样的图像中抽取不同尺度大小的裁剪图像，然后在测试过程中通过评估一些固定的裁剪图像来抵消这种随机性。

有时候在数据增强中会使用色彩抖动的方法，可能会在训练时随机改变图像的对比度和亮度，可以通过色彩抖动得到更复杂的结果。

还有一些其他的正则化策略包括：批量归一化（通常使用这个）、DropConnect、部分最大池化、随机深度。

课时3 迁移学习

接下来介绍有关迁移学习的思想。在使用正则化的时候通过加入不同的正则策略可以帮助减小训练误差和测试误差的间隙，这种情况下往往需要更大的数据集，而迁移学习的思想不再需要大数据集也能训练卷积神经网络。

首先找到一些卷积神经网络，这里有足够的数据去训练整个网络；然后尝试把从这个数据集中训练出的提取特征的能力用到新的小数据集上；接着一般做法是从最后一层的特征到最后的分类输出之间的全连接层，需要重新随机初始化这部分矩阵；现在如果要训练一个线性分类器只需训练最后这层让它在数据上收敛；若有更多的数据，可以尝试更新网络的更大一部分。

通用的策略是更新网络时将学习率调低，因为最初的网络参数可能在大数据集上是收敛的，泛化能力已经很强，这时只希望让它们做一个微小的调整来适应新的数据集。

在使用迁移学习时可以想象成如下图的2*2的网格：

	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

迁移学习的思想是非常普遍的，目前无论是计算机视觉的哪方面应用，大多数人都不会从头训练这些东西，大多数情况下卷及神经网络在ImageNet上预训练，然后根据任务在进行精调。

（1）不管遇到什么样的模型，对于要处理的问题没有大数据集，应该做的是下载一些相关的预训练模型；

（2）然后要么重新初始化部分模型，要么在数据上进行精调。

这里给出了一些预训练模型的下载地址：

Caffe：https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow：https://github.com/tensorflow/models
PyTorch：https://github.com/pytorch/vision

总结：

（1）讨论了优化算法用来改进训练效果；

（2）讨论了正则化在改变测试集上的性能应用；

（3）讨论了迁移学习可以在小样本上训练的一样好。