【译】cs231n第三课：优化 - 随机梯度下降

2016-07-19 | 9

> 本来是要学习cs224d的，但cs224d要求学几节cs231n的课程作为预备课程，故有如此翻译。
> 原文链接

简介

上两节我们介绍了图片分类任务的两个关键：

1. 一个参数化的得分函数，它负责将原始图片的像素映射成不同分类的得分（比如线性方程）。
2. 一个损失函数，它负责度量基于特定的一组参数产生的图片种类的预测与正确分类匹配的程度，对训练集每一张图片都计算损失，就能得到整个训练集的总的损失程度。有很多种损失函数，比如Softmax和SVM。

回忆一下，线性方程为：$f(x_i, W) = Wx_i$，SVM总损失函数的形式为：

$$L = \frac{1}{N}\sum_i \sum_{j \neq y_i} \left[\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + \Delta)\right] + \lambda \sum_k \sum_l W_{k,l}^2$$

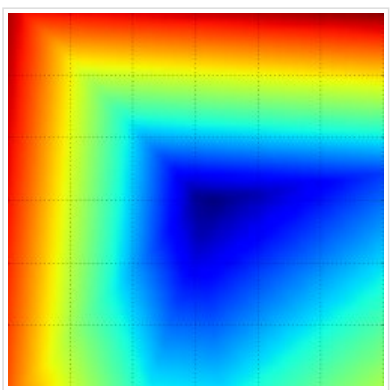
我们可以看到，如果有一组特别的参数组合W能够使尽可能预测训练集中更多的样本为正确的分类，那么将会得到非常低的总损失值。我们现在来引入和介绍第三个关键：优化。优化是找出能够得到最低损失值的特定参数组合W的过程。

预告

一旦我们明白了这三个核心部件怎么工作，我们会修正第一个部件（参数化的映射方程），并且将其拓展为更复杂的方程：神经网络和CNN。然而，损失函数和优化过程将相对保持不变。

视觉化损失函数

这节课使用的损失函数往往都定义在高维空间（比如CIFAR-10中的线性分类器的权重矩阵大小为$[10x3073]$，总共有30730个参数），很难图形化展示。但是，通过只展现高维空间中的一维（以射线形式）或者两维（以平板模式），这样我们也能获得一些启示。比如，随机产生一个权重矩阵W（它对应了空间中的任何一点），然后随着射线前进，记录损失函数的变化。也就说，我们可以选择一个的方向W_1（这个方向就是指的要改变的权重矩阵的某个元素，从值上考虑很有可能是1，当把权重矩阵看成空间中一个点时，此时代表了这个点可以移动的一个方向，通过三维，扩展到高维），并且沿着这个方向通过公式：$L(W + aW_1)$以不同的a值计算出损失。这个过程可以产生一张以a值为x轴和以损失为y轴的曲线图。我们能进行在二维进行类似的事，比如当公式为：$L(W + aW_1 + bW_2)$，我们不断的改变a和b的值，a和b对应图片中的x轴和y轴，损失的值可以以颜色来表示。



多分类SVM损失函数的图片（未添加正则化），左边和中间的图是针对一个样本点，而右边的图是针对的是CIFAR-10中数百个样本点。左侧：只改变a值而产生的一个维度的损失。中间：改变两个维度时产生的损失，蓝色表示损失低，红色表示损失高。注意图中损失函数其产生的折线

图像。对于多个样本点产生的损失是求了均值的，所以我们会在右边的图片中观察到椭圆的形状，它就是由多个折线图像求均值后产生的。

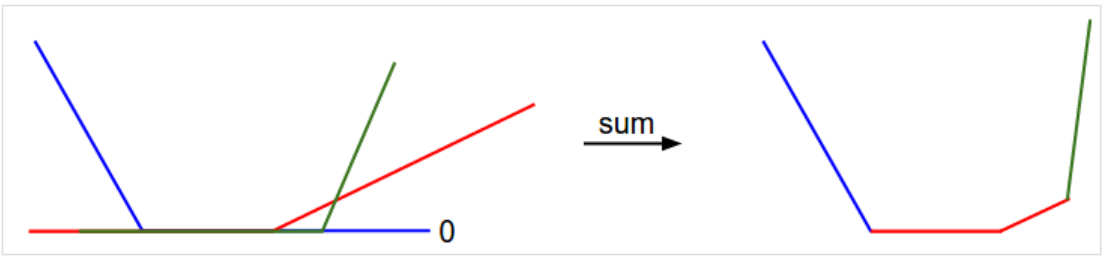
我们可以通过公式看看为什么会产生线性形状。对于一个样本点，我们有公式：

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + 1) \right]$$

公式很显然告诉我们对于一个样本点的损失是对以W为参数的线性方程式求和产生的（以$\max(0, -)$公式引入了0作为阈值），此外，W中的某一行（比如：w_j）有可能是在对应错误分类的时候却是一个正数，而有时候对应正确的分类又是一个负数（这是完全有可能发生的吧）。让我们考虑一个例子来说明这一点吧，一个简单的数据集包含只有一个维度的点和三个分类。总共SVM的损失（不含正则化项）应该如下：

$$L_0 = \max(0, w_1^T x_0 - w_0^T x_0 + 1) + \max(0, w_2^T x_0 - w_0^T x_0 + 1) \quad (1)$$
$$L_1 = \max(0, w_0^T x_1 - w_1^T x_1 + 1) + \max(0, w_2^T x_1 - w_1^T x_1 + 1) \quad (2)$$
$$L_2 = \max(0, w_0^T x_2 - w_2^T x_2 + 1) + \max(0, w_1^T x_2 - w_2^T x_2 + 1) \quad (3)$$
$$L = (L_0 + L_1 + L_2)/3 \quad (4)$$

因为这些样本点都是一维的，所以样本点x_i和权重w_j都一个数字。那么现在我们假设，有w_0的项都变为0。然后我可视化这个过程：



用一个维度来解释数据损失。X轴代表单个权重，Y轴代表损失。数据的损失是由多个项相加的，每一个都独立的依赖于一个权重，或者一个以0作为阈值的线性方程。总的SVM损失就是这个形状（折线形）的30,730维度的版本。

> 主要是这个例子没看懂，不同颜色线是啥意思，代表不同的权重嘛。为啥又要说有W_0的项变成0呢。

此外，相信你也从这个椭圆形的图像看出来了SVM损失函数实际上是一个凸函数，有许多文献叙述了如何高效地最小化凸函数，你也可以参与另一个斯坦福课程的一节来学习凸优化。一旦我们将得分函数f拓展为神经网络后，我们的目标函数将会变为非凸函数，因此视觉化后的图片不再是椭圆形，而是复杂的、凹凸不平的函数。

不可微损失函数 作为技术要点，你可能已经见过在损失函数某些点是不可微的，因为在这些点上由于max操作（max的存在把）导致这些点没有斜率。然而，次斜率是存在的，并且经常被使用。本课中，我们将不区分次梯度和梯度。

优化

重申一下，损失函数让我们度量了一组特定W的组合。而优化的目标就是找到一组W，使得损失函数最小。我们将研究和开发一个方法来最优化损失函数。如果你有机器学习的背景，那么本节在你看起来应该是很奇怪的，虽然我们的损失函数（SVM损失）是一个凸优化问题，但是我们没有使用任何凸优化的技巧，这是因为我们最终的目标是去优化神经网络，而那时凸优化技巧是没有用的。

策略#1 坏主意：随机查找

正是因为查看一组W的效果是非常简单，我们第一个主意就是随机产生权重W，然后记录效果最好的一组W即可。代码如下：

```
# X_train每一列是一个样本(比如：3073 x #  ，同时假设Y_train是正确分类的索引(是一个一维数组，大小为50,000)
# 假设L是损失函数
bestloss = float("inf") # 赋值最高的float值
for num in xrange(1000):
        W = np.random.randn(10, 3073) * 0.0001 # 随机产生参数
        loss = L(X_train, Y_train, W) # 获得整个训练集的损失
        if loss < bestloss: #    记录损失最小，也就是效果最好的记录
                bestloss = loss
                bestW = W
        print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)
# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
```

```
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (trunctated: continues for 1000 lines)
```

从上面代码可以看到，我们测试了很多随机的权重参数，有些效果好，有些效果差。我们能拿到在随机搜索到的W在测试集上测试：

```
# 测试集X_test 是 [3073 x 10000], Y_test:[10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, 测试集中的所有类别的得分
# 获得每个样本最高得分的索引（也就是预测的类别）
Yte_predict = np.argmax(scores, axis = 0)
# 计算准确率（预测正确个数的占比）
np.mean(Yte_predict == Yte) #Yte_predict==Yte 可以得到一个数组，全是True或者False，然后np.mean就可以算出平均数了，非常牛逼
# returns 0.1555
```

最好的W给出的准确率只有15.5%。因为随便乱猜分类只有10%的准确率，所以15.5%对于这样不费脑子的随机搜索已经算不错的效果了。

核心思想：迭代修正

当然，我们肯定能做得更好。关键要点是找出最佳W本身就非常难，难道几乎不可能（尤其是当W包含整个神经网络的权重时）。但是有一个办法就是不断的修正一个特定的W，这个比较简单。换句话说，我们从一个随机产生的特定的W开始，然后不断的修正它，使他的效果一次比一次好。

> 我们的策略是从一个随机的W开始，然后不断的修正它，每次都得到更低的损失

像是在蒙住双眼下山

这个类比是帮助你理解这个算法，想象是自己被蒙住了双眼，但是想要找到下山的路，这时我们只有伸出脚来一点一点的试。在CIFAR-10的例子中，山是30,730维度的，因为W有3073 x 10个参数，每一点我们都能算出一个损失（山的高度）

策略#2：附近随机搜索

这种策略你可以想象自己在随机的方向上走了一步，只有当这一步产生的损失更低时，才会过去（上一种策略就是人在随便跳，在山上不停的跳来跳去，跳到更低的点时，就切换位置）。具体说来，我们从一个随机的W开始， 然后再加一个修正的δW，如果修正后：$W + \delta W$的损失更低，那么我就执行这个更新。代码如下：

```
W = np.random.randn(10, 3073) * 0.001 #起始的随机的W
bestloss = float("inf")
for i in xrange(1000):
        step_size = 0.0001
        Wtry = W + np.random.randn(10, 3073) * step_size
        loss = L(Xtr_cols, Ytr, Wtry)
        if loss < bestloss:
                W = Wtry
                bestloss = loss
        print 'iter %d loss is %f' % (i, bestloss)
```

使用和上一个策略相同的迭代次数：1000次，这样的方式能在测试集上获得21.4%的准确率。虽然效果好了那么一点，但是十分浪费计算资源。

随梯度下降

上一节中，我们试着在权重空间中找到一个能够改进权重的方向（就是得到更低的损失）。实际上，我们没必要在方向上随机选择，我们可以利用数学来获得最好、下降幅度最大的方向，这个方向与损失函数的梯度相关。在我们的下山类比中，这种方式放佛是我们用脚去感受坡度，然后选择一个最陡峭的方向。

在一维的公式中，我们需要的这个坡度就是在任意点改变的速度（斜率）。所谓梯度就是斜率的泛化，斜率是接受一个数字为变量的公式，而梯度是接受一个向量的公式。此外，梯度就是一个承载斜率的向量（通常被会称为导数），向量中每个元素代表输入空间的一个维度。对于一维公式，导数的数学表达式如下：

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

当这个公式的输入为一个向量，而非一个数字时，称这个导数为偏导，梯度就是一个向量的偏导，可以简单的理解梯度就是一个向量，向量里每个元素都是在不同维度上的偏导。

计算梯度

有两种计算梯度的方式，一个慢、不准、简单的方式：数值梯度，一个快、准、易错的、需要代数学的方式：分析梯度。我们两者都会介绍。

以有限的差值计算数值梯度

上面的公式告诉了我们如何计算数值梯度，这里有一个函数，它接受一个公式f和一个向量x，来求出梯度，并且返回x在f处的梯度。

```
def eval_numerical_gradient(f, x):
        """
        一个数值梯度的简单的实现，返回f在x点处的梯度
        - f 只接受一个参数的函数
        - x 是多维空间的一个点(所以是一个numpy 数组) ，需要评价这个点的梯度
            注意x是不是图片，而且权重矩阵
        """

        fx = f(x) # 在原点的函数值
        grad = np.zeros(x.shape)
        h = 0.00001

        # 遍历x的索引,注意x是权重矩阵所以3073 x 10
        # 而下面这句代码就是为了能够对x里面的 3073 x 10个元素遍历
        # 并最后产生一个相同的矩阵
        it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
        while not it.finished:

                # 算出在x+h点处的函数值
                ix = it.multi_index
                old_value = x[ix]
                x[ix] = old_value + h # 在原来的点上增加一点h
                fxh = f(x) # 算出新点的函数值f(x + h)
                x[ix] = old_value # 把原来的值再还回去(极其重要!)

                # 算出偏导
                grad[ix] = (fxh - fx) / h # 记录在这个方向上的偏导
                it.iternext() # 下一个维度

        return grad

```

使用上面给的梯度公式，代码一个一个地遍历了所有维度，然后对每一个维度的值上做一点变化，算出了在这个维度上公式的偏导数，确定了变化率。最后变量 grad 装载了所有的梯度。

实践的考虑

注意虽然数学公式在定义时，梯度是当h趋近于0时产生的，但是实践中，通常使用一个非常小的数就行了（比如例子中的1e-5）。理想情况下，你会使用尽可能小而又不会引发数值问题的数。此外，实践中使用中心差值公式（ntered difference formula）：$[f(x+h)-f(x-h)]/2h$计算数值梯度效果会更好，详情请见wiki。

我们可以用上面的函数算出任何公式、任何点的梯度，让我们计算在权重空间任意点在CIFAR-10损失公式的权重。

```
# 为了使用上面那个通用的函数，一个公式只接受一个参数
# 所以我们做了一次封装
def CIFAR10_loss_fun(W):
        return L(X_train, Y_train, W)

        W = np.random.rand(10, 3073) * 0.001 # 随机权重
        df = eval_numerical_gradient(CIFAR10_loss_fun, W) # 拿到梯度
```

梯度给了我们在损失函数在每个维度上的斜率，我们可以用这个梯度来做一个更新。

```
loss_original = CIFAR10_loss_fun(W) # 原来的损失
print 'original loss: %f' % (loss_original, )

# 同时我们看看不同步长的影响（梯度确定了方向，但是变化量是多少由步长决定）
for step_size_log in [-10, -9, -8, -7, -6, -5,-4,-3,-2,-1]:
        step_size = 10 ** step_size_log
        W_new = W - step_size * df #  这里为什么是减？
        loss_new = CIFAR10_loss_fun(W_new)
        print 'for step size %f new loss: %f' % (step_size, loss_new)

# prints:
# original loss: 2.200718
# for step size 1.000000e-10 new loss: 2.200652
# for step size 1.000000e-09 new loss: 2.200057
# for step size 1.000000e-08 new loss: 2.194116
# for step size 1.000000e-07 new loss: 2.135493
# for step size 1.000000e-06 new loss: 1.647802
# for step size 1.000000e-05 new loss: 2.844355
# for step size 1.000000e-04 new loss: 25.558142
# for step size 1.000000e-03 new loss: 254.086573
# for step size 1.000000e-02 new loss: 2539.370888
# for step size 1.000000e-01 new loss: 25392.214036
```

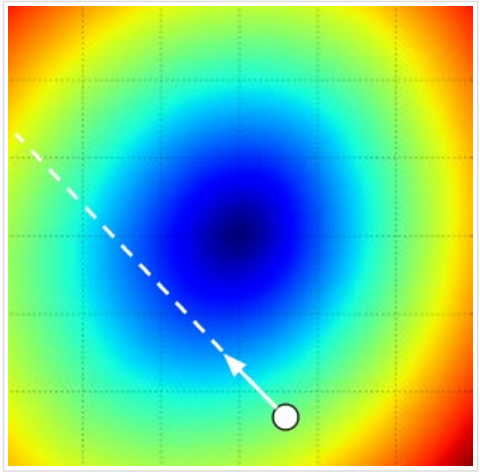
> W_new = W - step_size * df # 这里为什么是减？
> 因为我们是想获得更小的损失函数，所以我们的W是往能够获得更小的损失函数的方向变化，所以就得减。我们可以通过一维的情况来分析。当导数大于0时，是增函数，所以x应该减少，那么x减去导数就可以获得更小的y值；当导数大于0时，是减函数，所以x应该增加，那么x减去负的导数值就等于加上了正值，所以也能获得更小的y值。

减去梯度以更新

注意到我们的W_new做更新时是减去了负的梯度值，这是因为我们希望损失函数减少，而不是增加。

步长的影响

梯度只是告诉了我们能最快减少损失函数值的方向（注意是分了维度的，也就是每个维度上，但是我认为当不两个维度综合起来，未必就是最快的方向（最陡峭的地方）），但是并没有告诉我们沿着这个方向上走多远。在我们之后的课程中可以看到，选择步长（也就是学习速率）是在神经网络中最重要也是最头痛的超参数之一。这种感觉仿佛是在下山的行动，我们能够用脚感觉到坡度，但是我们应该跨多大的步子却不清楚。如果我们一点一点移动我们的脚步，并且坚持这么做（这对应着我们选择了一个数值小的数作为步长），也许只能获得很小的进展。相反，如果我们选择跨大步子，也许我们一下就能下降的很快，但也许我们又没有效果。上面代码的例子表明，在某些点跨更大的步子反而获得更高的损失。



图示化步长的影响。我们从一个点W开始，然后算出梯度（负数，就是白色的方向）告诉了我们能够最快降低损失函数的方向。小步伐能够见效但是速度太慢，大步伐速度虽快但是有可能跨过去了。仔细观察图片，一次大的步伐就有可能跨过去了并且导致更大的损失。所以步长（后面我们称之为学习速率）将会是最重要的超参数，也就是我们最需要调整的超参数。

效率问题

你可能已经注意到了，计算一次数值梯度的复杂度是线性的，因为必须遍历一遍矩阵里的所有参数。在我们例子中，因为有30730个参数，所以我们要计算30730次损失函数，从而算出梯度，并且这样之后也只能对W进行一次更新。当使用神经网络的时候，因为有上百万个参数，所以这个问题更加严重。显然，这个方案不能扩展到更复杂的模型上使用，所以我们需要更好的办法。

使用分析代数计算梯度

用一个固定的差值代替无限趋近于0来计算数据梯度非常简单，但是缺点也非常明显，就是尽管我挑了一个非常小数值（h），但是真正的梯度毕竟还是定义为无限趋近于0，同时计算量也非常大。第二种计算梯度的方式就是代数分析法，它可以允许我们直接推倒出一个公式来计算梯度（而非近似值），并且计算速度非常快。然而，不像数值梯度，代数分析法的梯度的代码实现非常容易出错，所以在实践中我们通常会计算出代数分析法的梯度，然后和数值梯度做比较，确认实现是否正确，这个过程称为梯度检查。

让我们使用SVM损失函数和一个样本点作为例子：

$$L_i = \sum_{j \neq y_i} \left[\max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta) \right]$$

我们可以对权重求导，比如我们对w_{y_i}求导，即可得到（下一节有推导过程）：

$$\nabla_{w_j} L_i = \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

其中1表示一个指示函数，当内部条件为真是则取1，否则为0。虽然公式看起来有点吓人，但是实际上代码实现的时候非常简单，因为只需要数一数有多少分类没有满足差值（这些没有满足的将对损失值产生贡献），然后按乘以输入向量x_i即可，结果就是梯度。注意这个梯度只是对应了W里面分类正确的一行。对于其他$j \neq y_i$的行，梯度如下：

$$\nabla_{w_j} L_i = \mathbb{1}(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0) x_i$$

一旦我们推倒出公式，实现起来是非常简单的，并且也方便执行梯度更新。

梯度下降

现在我们可以计算损失函数的梯度了，这个过程就是反复的计算梯度，然后更新参数，这个过程被称为梯度下降。它的vanilla版本（没有个性化设置的版本）如下：

```
# Vanilla 版本梯度下降

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # 权重参数更新
```

这是神经网络库中的核心循环。当然也有其他优化方法（LBFGS），但是在优化神经网络的损失函数中，梯度下降是目前最被常用和建立。在本课的学习过程当中，我们会往这个循环里不断的添加东西（比如更新的细节），但是核心思想就是随着梯度下降，直到我们对一直不变的结果满意(也就是趋于稳定，找到了最低点)。

分小批梯度下降

在大规模的应用中（比如ILSVRC的比赛中），数据集可能有百万量级。因此，为了一次的参数更新，就计算整个训练所有样本点的损失，这样太浪费资源了。通常的做法是只取一部分（一批）训练集计算梯度。比如在目前进行的art ConvNets，一批是从1.2百万中取出256个样本点。每次使用一批进行参数更新：

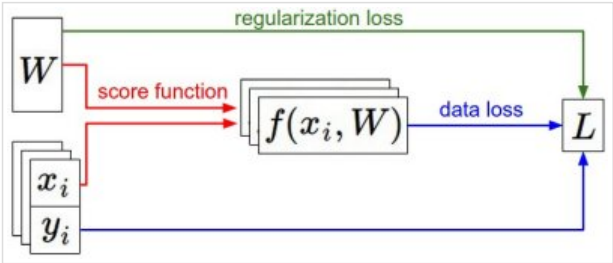
```
# Vanilla 版本的分批处理的梯度下降

while True:
  data_batch = sample_training_data(data, 256) # 抽样获得256个样本
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

因为在训练集中的样本点是相关的，所以这样抽样效果也会非常好。为什么明白这一点，可以想象一个极端的例子，在一百二十万张图片的数据集ILSVRC中，可以认为是由1000张不同的照片复制而成（每一张照片都是一个分类，也就每张照片有1200份复制）。很显然我们对于一个分类的1200张照片计算出来的梯度是一样的，那么我们计算一百二十万张照片再平均得到的损失，其实和抽样获得的1000张照片的损失的平均值是一样的。当然在实践中，数据集中不可能包含重复的图片（1200张照片是近似的），所以从抽样获得梯度是由全部样本计算的梯度的一个近似值。此外，因为使用抽样时计算的样本量少，所以计算速度非常快，那么经过更多的参数更新，最终也能达到效果。

分批处理的极端例子就是每次只使用1个样本点，这种极端的方法被称为随机梯度下降（SGD）（有时也被称为在线梯度下降）。这在实践中比较少见，因为由于代码中的向量计算经过了优化，所以对100个样本点计算一次梯度，比对一个样本点计算100次的速度要快得多。尽管SGD的定义是每次使用一个样本点计算梯度，但是在实践中你会经常听说人们用SGD这个词表示做了分批处理（一次用了多个样本点），实际上该称为（MGD，Minibatch Gradient Descent，也有少数人称为BGD，Batch gradient descent）。每次分批处理的大小量也是一个超参数，但是通常并不会用交叉验证确定它。通常它的确定基于内存的限制（如果有），否则就是一些常见的数值：32、64、128。我们使用2的次方是因为很多向量优化的计算在处理2的次方时会快很多。

总结



对信息流的总结。数据集成对(x,y)固定，权重以初始化为随机值并且也能被改变。先利用得分函数计算出每个分类的得分，并存储在向量f中。在损失函数中包含两个部分：数据损失代笔了得分f与真实分类y的匹配程度；正则化损失是只是一个公式的权重。在梯度下降中，我们计算了权重的梯度（也许只使用了部分数据），然后我们使用它更新了参数。

本小节中，

- 我们将损失函数看作为在高维空间寻找最低点的优化过程。这项工作仿佛是我们是一个蒙住眼像要下山的人。此外，我们也发现SVM损失函数最后呈现了折线形或椭圆形。
- 我们在优化损失函数时使用了迭代修正的方法，开始时我们以随机数初始化权重，然后一步一步的修正它，直到损失函数的值最低。
- 我们可以看到一个公式的梯度给出了下降最快速的方向。并且我们讨论一种利用固定差值计算梯度近似值（也就是h）的方案，这种方案简单、低效。
- 我们看到权重参数更新时需要一个步长（也就是学习速率），它的大小设置必须谨慎：当数值太低时进展比较慢，但当数值太大时进程虽快，确有跃过最低点的风险。我们将在以后的章节中讨论如何权衡这个参数。
- 我们对比着讨论过数值权重和分析权重。数值权重虽然简单，但是它的结果是近似值，而且计算代价非常大。分析权重虽然准确，易于计算，但是更容易出错，因为它需要用数学计算导数。因此，当我们在实践中使用分析梯度时，都会算出数值梯度与分析梯度对比确认结

果，这个过程称为梯度检查。

- 我们引入了梯度下降的算法，它递归计算一次梯度，然后进行一次参数更新。

说明：本节有一个关键没有讲，就是如何用代数分析法推倒出损失函数对权重的导数（目前只是看到了结果，大概有个感觉），这是在设计、训练、理解神经网络中最重要的能力。在下一节中，我们学习如何熟练地用链式法则（反向传播）计算梯度，这个方法能够使我们高效的优化在各种的神经网络（包括CNN）中的损失函数。

#cs231n

‹ 【译】cs231n第二课：线性分类 - SVM与Softmax

【译】cs231n第四课：理解后向传播 ›