第三章第2部分

2. 神经网络：技巧

在讨论了神经网络背后的数学基础后，我们现在要探讨实践中神经网络的使用技巧。

2.1 梯度校验

在上一节，我们利用了基于微积分的解析法，仔细讨论了怎样计算了误差关于模型参数的梯度，以及如何更新参数。现在我们要介绍一种数值的方法来近似计算这些梯度，尽管这种方法在训练网络中效率很低，但它让我们非常精确地估计了每个参数的梯度。因此它可以作为我们之前用解析法进行求梯度是否正确。一个模型的参数向量θ和损失函数J，利用**centered difference formula**方法数值法计算在θ_i附近的梯度

$$f'(\theta) \approx \frac{J\left(\theta^{(i+)}\right) - J\left(\theta^{(i-)}\right)}{2\epsilon}$$

其中ϵ是一个很小的数（通常为$1e^{-5}$）。$J(\theta^{(i+)})$项是将参数θ的第i个元素增加ϵ，通过前向传播算法计算出的误差。相似的，$J(\theta^{(i-)})$项是将参数θ的第i个元素减少ϵ，通过前向传播算法计算出的误差。因此，使用两次前向传播，我们可以估计出误差对模型中任何参数的梯度。注意，数值型梯度的概念本应该和导数的概念一致，为

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

这和之前的做法有一点不同，因为这仅仅是将x往正的方向扰动了ϵ来计算梯度。尽管这种方法也可以接受，但实际情况，还是使用centered difference formula方法，通过扰动两个方向，结果更加精确和稳定。理由是，为了更好的把函数f在一个点附近的梯度/斜率给估计出来，我们需要检查f函数在这个点的左边和右边的性质。使用泰勒定理，可以知道利用centered difference formula计算出的梯度的误差大概是ϵ^2级别的，这是一个非常小的数字。相反，普通的计算导数的方法会更容易产生误差。

现在，你可能会有一个问题，既然这个方法很精确，为什么我们在计算网络的梯度时，不采用这种方法，而要采用后向传播法呢。答案很简单，之前也提到过，是因为这种方法太耗时间了。试想每次我们要计算关于一个元素的梯度时，我们需要做两次前向传播，这很浪费计算。而且，许多大型的神经网络会包含成千上万的参数，对每个参数都计算两次前向传播的方法不是最优的。并且，在优化方法中，例如SGD，我们每次迭代都要计算很多梯度，这样的迭代有几千次，显然这种方法会变得难以控制。因为这非常美效率，所以我们仅使用它来进行解析法的梯度校验，这样计算才比较快。一个标准的梯度校验的方法是如下的代码

```
def eval_numerical_gradient(f, x):
    """
    a naive implementation of numerical gradient of f at x
    :param f: should be a function that takes a single argument
    :param x: is the point (numpy array) to evaluate the gradient at
    :return:
    """
    fx = f(x) # evaluate function value at original point
    grad = np.zeros(x.shape)
    h = 0.00001
```

```
    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        # evaluate function at x+h
        ix = it.multi_index
        old_value = x[ix]
        x[ix] = old_value + h # increment by h
        fxh_left = f(x) # evaluate f(x + h)
        x[ix] = old_value - h # decrement by h
        fxh_right = f(x) # evaluate f(x - h
        x[ix] = old_value # restore to previous value (very important!)

        # compute the partial derivative
        grad[ix] = (fxh_left - fxh_right) / (2*h) # the slope
        it.iternext() # step to next dimension
    return grad
```

2.2 正则化

和许多机器学习模型一样，神经网络非常容易过拟合，即它会在训练集上获得了完美的表现，但是却在没见过的数据上泛化能力比较差。一个常用的压制过拟合（也叫做"高方差问题"）方法是引入L2正则惩罚。方法是在损失函数J的后面添加一项，因此计算整体的损失为

$$J_R = J + \lambda \sum_{i=1}^{L} \left\| W^{(i)} \right\|_F$$

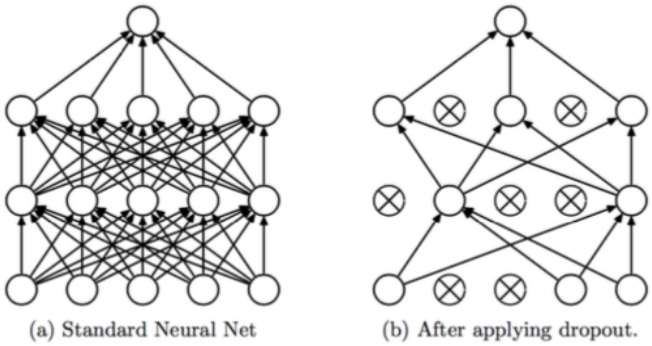
在上面的函数中，$\left\| W^{(i)} \right\|_F$是矩阵$W^{(i)}$（网络中第$i$个权重矩阵）的Frobenius范数，$\lambda$是超参数，用于控制正则项相对于原来的损失函数的比重。这相当于当我们在优化原始的损失函数，即我们试图最小化J_R时，正则化在惩罚过大的权重。由于Frobenius范数的二次现象（即要计算矩阵中每个元素的平方之和），L2正则化可以有效的减少模型的灵活度，因此会减少过拟合的现象。引入这样的约束可以解释为，我们先验性地认为最优的权重应该是接近0的。至于如何接近，这取决于λ。正确地选择λ很重要，需要通过超参数调参。λ太大会导致权重都接近0，模型就无法从训练集学到任何有意义的信息，这会导致模型在训练集、验证集和测试集上有糟糕的准确率。如果λ太小，我们又会陷入过拟合的领域中。注意，偏置项不需要被正则化，从而不需要影响损失函数，思考为什么偏置项不需要。

的确，有时也会使用其他类型的正则化方法，例如L1正则化，即把矩阵中的每个元素取绝对值后，再加和起来（而不是平方和）。然而，实际中它比较邵雍，因为它会导致参数的稀疏。再下一节中，我们会讨论dropout，这是另外一种有效的正则化方法，它通过随机将一些神经元，在前向传播中的连接打断（例如将对应的权重矩阵中的元素设置乘0）。

2.3 Dropout

Dropout是一个很强大的正则化方法，它第一次被提到是在Dropout: A Simple Way to Prevent Neural Net-works from Overfitting.这篇论文中。它的想法很简单，但是很有效，即，在进行前向、后向传播的训练过程中，我们会以$(1-p)$的概率，随机的"丢失"一些神经元（等价于，我们会让每个神经元以p的概率存活）。然后，在测试阶段，我们会用完整的网络来计算我们关于样本的预测值。通过Dropout，网络一般会从数据集中学到更加有意义的信息，过拟合的可能性更小，因此通常能够获得在指定任务中更好的表现。为什么Dropout很有效，一个直觉的原因是，Dropout能够使得每次训练的网络呈现指数级别的小，然后在预测的时候，将这些网络平均起来。

实践中，我们引入Dropout的方法是，把每层的神经元的输出h，按照概率为p的方式将其保持不变，而以$(1-p)$的概率将其设置为0。然后，在后向传播过程中，我们仅仅将梯度在那些存活的神经元中通过。最后，在测试阶段，我们在前向传播中，保持所有的神经元都是存活的。但是，一个关键的问题是，为了让Dropout能够有效地起作用，在测试过程中，神经元的输出的期望值应该要和训练过程中的大概保持一致，否则，这两个阶段，神经元的输出将会是不同的量级，则网络的表现不再是确定的。因此，我们要在测试阶段，将每个神经元的输出，除以一个特定的数。如何确定这个数的值，使得在训练和测试阶段，神经元输出的值的期望能够等加，这个问题留给读者。



Dropout applied to an artificial neural network. Image credits to Srivastava et al.

2.4 神经单元

到此为止，我们已经讨论了包含sigmoid神经元，从而带来了非线性特点的神经网络。然而，在许多应用中，使用其他的激活函数，网络能够表现得更好。下面列出了一些常见的做法，包含了函数以及对应的梯度定义，用于替代我们之前讨论的sigmoid函数。

Sigmoid：函数形状如图9，这是我们讨论过的一种默认的选择，激活函数的形式如下

$$\sigma(z) = \frac{1}{1+\exp(-z)}$$

其中$\sigma(z) \in (0,1)$。它的梯度是

$$\sigma'(z) = \frac{-\exp(-z)}{1+\exp(-z)} = \sigma(z)(1-\sigma(z))$$

Tanh：函数形状如图10，Tanh函数可以替代Sigmoid函数，实践中，它通常能够更快速的收敛。它和Sigmoid函数之间主要的区别在于，Tanh函数输出的范围是-1到1，而Sigmoid函数的范围是0到1。

$$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = 2\sigma(2z) - 1$$

其中，$\tanh(z) \in (-1,1)$。它的梯度是

$$\tanh'(z) = 1 - \left(\frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}\right)^2 = 1 - \tanh^2(z)$$

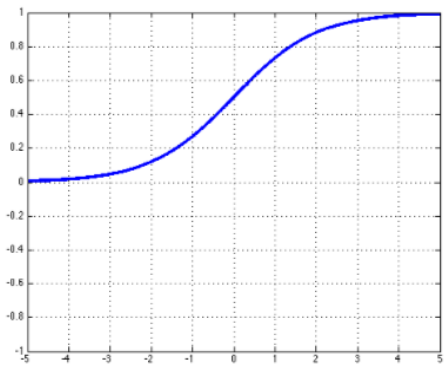


Figure 9: The response of a sigmoid nonlinearity

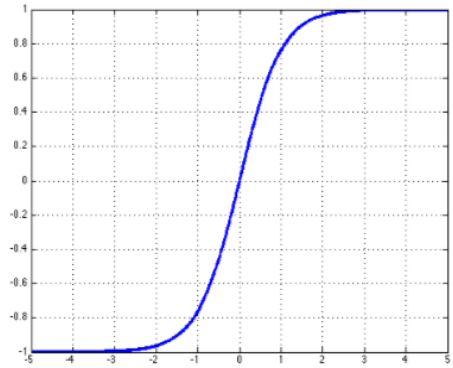


Figure 10: The response of a tanh nonlinearity

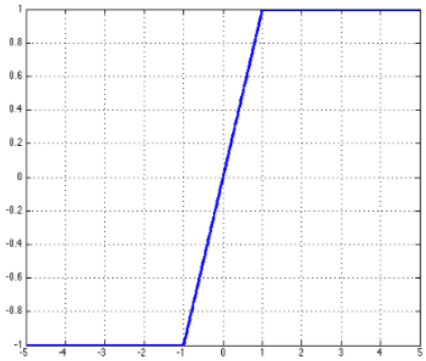


Figure 11: The response of a hard tanh nonlinearity

Hard tanh：函数形状如图11。hard tanh函数有的时候比tanh函数更加受欢迎，原因是因为它计算效率快。但是，它在z大于1的地方，函数会陷入平坦，它的激活函数是

$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \leq z \leq 1 \\ 1 & : z > 1 \end{cases}$$

它的导数也可以用分段函数来表达

$$\text{hardtanh}'(z) = \begin{cases} 1 & : -1 \leq z \leq 1 \\ 0 & : \text{otherwise} \end{cases}$$

Soft sign：函数形状如图12。soft sign函数是另外一种非线性函数，也可以被视为tanh函数的替代，因为它不会像强行截断的函数那样容易陷入平坦，它的激活函数是

$$\text{softsign}(z) = \frac{z}{1 + |z|}$$

它的导数是

$$\text{softsign}'(z) = \frac{\text{sgn}(z)}{(1 + z)^2}$$

其中sgn函数是符号函数，根据z的符号，返回± 1。

ReLU：函数形状如图13。ReLU（线性整流单元Rectified Linear Unit）函数是一个受欢迎的激活函数，因为即使z很大，它也不会陷入平坦，在计算机视觉领域已经有了很成功的案例。

$$\text{rect}(z) = \max(z, 0)$$

它的导数可以用分段函数表示

$$\text{rect}'(z) = \begin{cases} 1 & : z > 0 \\ 0 & : \text{ otherwise} \end{cases}$$

Leaky ReLU：函数形状如图14。传统的ReLU单元对于不是正数的z，不会传播误差。leaky ReLU激活函数修改了这个问题，当z是负数时，它仍会后向传播一个小的误差。

$$\text{leaky}(z) = \max(z, k \cdot z)$$
$$\text{where } 0 < k < 1$$

它的导数是

$$\text{leaky}'(z) = \begin{cases} 1 & : z > 0 \\ k & : \text{ otherwise} \end{cases}$$

2.5 数据预处理

与通常的机器学习模型一样，模型能够在任务中取得合理的表现的一个关键步骤，是数据预处理。以下介绍一些常见的预处理技术。

去均值化

给定一组输入数据集X，我们习惯于把数据集X减去它的特征向量的平均值，将数据给去中心化。实践中一个很重要的一点是，这里的平均值，仅仅是通过训练集计算的，并且这个均值将被用于训练集、验证集和测试集中。

归一化

另外一个常见的做法（即使可能没有比去均值化那么常见）是把输入数据的每个特征，放缩到一个相似的量级范围内。这种方法很有用，因为输入的特征通常有不同的单位，但是我们想要让所有的特征都同等的重要。为此，我们简单地除以在训练集中每个特征各自的标准差。

白化

白化不像去均值和归一化那么常见，它使得数据集的协方差矩阵是一个单位矩阵，这意味着特征之间是不相关的，并且每个特征的方差等于1。首先，要把数据集X中的每个特征分别减去对应的均值，得到X'，然后对X'进行SVD分解，得到矩阵U, S, V。然后计算UX'，把X'投影到以U的列作为坐标轴的空间中。最后，在结果中的每个维度上，除以各自对应的奇异值，从而将数据放缩到合适的大小（如果奇异值为0，我们就除以一个小的数字）。

2.6 参数初始化

将网络的参数按照合理的方式进行初始化，是取得显著表现的一个关键步骤。一个好的初始化策略是将网络的权重以一个均值为0，方差为一个小的数字正态分布进行随机采样。实践中，常常使用这种方法。但是，在Understanding the difficulty of training deep feedforward　neural networks(2010), Xavier et al这篇论文中，研究了在训练中不同的权重和偏置的初始化方案。发现表明，对于以sigmoid和tanh为激活函数的神经元，如果要初始化的权重为$W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$，按照以下的初始化方法可以得到更快的收敛速度，同时错误率也会更低。

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}\right]$$

其中，$n^{(l)}$为矩阵W的输入维度，$n^{(l+1)}$为矩阵W的输出维度。在这个参数初始化反感中，偏置单元初始化为0。这种方法想要保持，数据的方差和梯度的方差一致，无论是在前向传播还是在后向传播中。如果不这么初始化，梯度的方差（代表了信息）通常会随着后向传播而逐渐减少。

2.7 学习策略

模型的参数在训练过程中的更新程度/量级，是通过学习率(learning rate)来控制的。以下描述了普通的梯度下降法的公式中，α表示学习率。

$$\theta^{\text{new}} = \theta^{\text{old}} - \alpha \nabla_\theta J_t(\theta)$$

你可能会思考，为了快速收敛，我们应该将α设置得大一些。然而，快速收敛不能保证收敛程度高。事实上，如果学习率设置过高，损失函数可能会无法收敛，因为参数的更新可能会导致模型跨过了最低点，如图15。对一个非凸的模型(我们研究的大部分模型)，大的学习率带来的后果往往无法估计，但是损失函数无法收敛的可能性非常高。

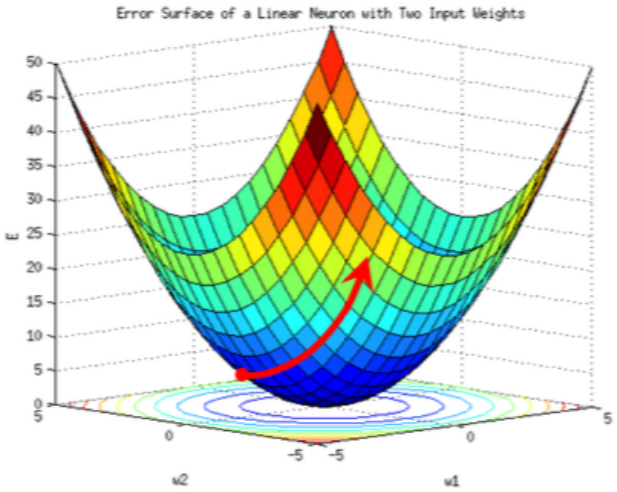


Figure 15: Here we see that updating parameter w_2 with a large learning rate can lead to divergence of the error.

要想解决这个无法收敛损失函数的问题，一个很简单的做法就是将学习率设置得小一点，以至于我们能够扫描整个参数空间。当然，如果我们设置的学习率太小了，在常规时间我们可能无法收敛，或者有可能会陷入局部最优点。因此，和其他的超参数一样，学习率需要一个有效的调参方法。

在深度学习系统中，因为训练是最耗费的过程，一些研究试图通过设置学习率来改善这个普通的训练方法。例如，Ronan Collobert通过权重W_{ij}（其中$W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$）来放缩学习率，放缩因子是权重$W$的输入维度($n^{(l)}$)的平方根倒数。

也有一些其他已经被证明有效的方法。其中一种叫做退火**(annealing)**，即在几次迭代后，通过某种方法减小学习率。这个方法使得我们一开始的学习率很高，然后迅速下降去靠近最低点。当我们离最低点很近的时候，我们开始降低学习率，从而可以在一个更加细致的区域内找到最优点。一个常见的退火方式为，每迭代n次，通过一个因子x来降低学习率α。指数衰减也很常见，即迭代至t的学习率$\alpha(t) = \alpha_0 e^{-kt}$，其中，$\alpha_0$是初始的学习率，$k$是超参数。另外一种降低学习率的方法是

$$\alpha(t) = \frac{\alpha_0 \tau}{\max(t, \tau)}$$

在这个方案中，α_0是一个可以调整的参数，表示初始的学习率。τ也是一个可调参数，代表了学习率可以开始降低的时刻。实践中，这种方法非常有效。在下一节中，我们会讨论另外一种自适应的梯度下降法，不需要手工设置学习率。

2.8 动量更新

动量方法，是梯度下降法的一种变体。它的灵感来源于物理中的运动过程。这种方法试图采用"速度"的概念来进行更加有效的参数更新方案。下面是伪代码

```
# Computes a standard momentum update on parameters x
v = mu*v - alpha*grad_x
x += v
```

2.9 自适应的优化方法

AdaGrad的实现过程和标准的随机梯度下降SGD很类似，惟独有一个关键的不同是，学习率对于每个参数是不同的。每个参数的学习率是多少，取决于历史过往中，该参数的梯度更新。当历史更新很少时，该参数会通过大学习率，使得更新很快。换句话说，如果参数过去没有没有被更新很多次，那么就很有可能会有一个大的学习率。

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i} \text{ where } g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$$

我们看到了，如果历史梯度的均方根RMS非常小，学习率就会很高。下面是一个简单的实现代码。

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += -learning_rate * dx / np.sqrt(cache + 1e-8)
```

另外一种常见的自适应方法是RMSProp和Adam，它们的更新方法如下（感谢Andrej Karpathy）

```
# Update rule for RMS prop
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / (np.sqrt(cache) + eps)
```

```
# Update rule for Adam
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += -learning_rate * m / (np.sqrt(v) + eps)
```

RMSProp是AdaGrad的变体，它使用了梯度平方的移动平均。特别是，不像AdaGrad，它的更新不会一致变小。Adam的更新方法是RMSProp的变体，但是引入了动量更新的思想。我们建议读者去阅读各自方法的原文，从而找到更多细致的分析。