Lecture Notes: Part III Neural Networks, Backpropagation

Keyphrases: Neural networks. Forward computation. Backward propagation. Neuron Units. Max-margin Loss. Gradient checks. Xavier parameter initialization. Learning rates. Adagrad.

这篇note会介绍单层、多层神经网络，还有如何被应用于分类场景。然后会讨论模型是如何使用分散梯度下降技术（反向传播）进行训练的。我们会看到链式法则是如何被用于按序更新参数的。经过一个严格的神经网络数学讨论，我们会讨论一些现实的在训练过程中的贴士和技巧，包括：神经元（非线性），梯度检查，Xavier参数初始化，学习率，Adagrad等。最后，我们会引出使用循环神经网络的语言模型。

1. Neural Networks: Foundations

我们在过去的讨论中建立了我们需要非线性判别器的事实，因为大多数数据都不是线性可分的，因此我们的分类器在这些数据上的表现是很局限的。神经网络是一类有着非线性决策边界的分类器，如图一所示。

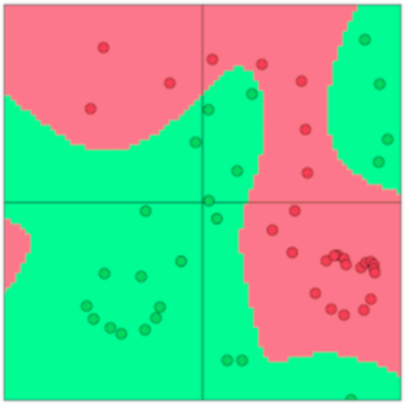

Figure 1: We see here how a non-linear decision boundary separates the data very well. This is the prowess of neural networks.
Fun Fact:
Neural networks are biologically inspired classifiers which is why they are often called "artificial neural networks" to distinguish them from the organic kind. However, in reality human neural networks are so much more capable and complex from artificial neural networks that it is usually better to not draw too many parallels between the two.

1.1 A Neuron

一个神经元是接受n个输入并产生一个输出的计算单元。让不同神经元的输出产生区别是通过其参数实现的（也称为权重）。其中一个最出名的神经元是“sigmoid”或者“二分类逻辑回归”单元。这个单元接受一个n-维的输入向量x，并产生一个尺度激活（输出）a。这个神经元也与一个n-维的权值向量w和一个偏置b有关。这个神经元的输出最终为：

$$a = \frac{1}{1 + \exp(-(w^T x + b))}$$

我们也可以将权值向量和偏置结合到上式中：

$$a = \frac{1}{1 + \exp(-[w^T \ b][x \ 1])}$$

这个式子可以可视化为图二：

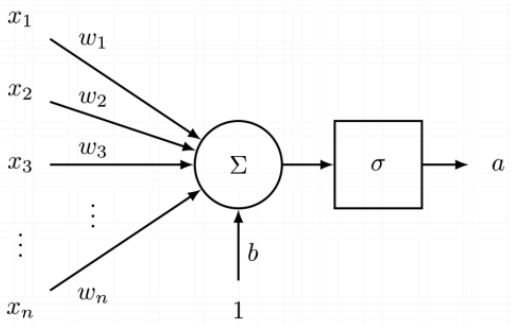

Figure 2: This image captures how in a sigmoid neuron, the input vector x is first scaled, summed, added to a bias unit, and then passed to the squashing sigmoid function.

1.2 A Single Layer of Neurons

我们可以将上述思想拓展为多个神经元，通过考虑输入x是多个神经元的输入，如图三所示：

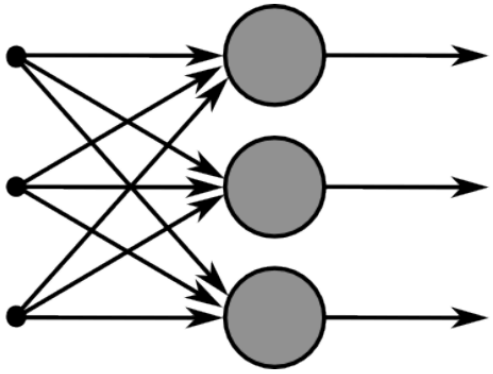

Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x.

如果我们将不同神经元的权重设为$\{w^{(1)}, \ldots, w^{(m)}\}$，偏置设置为$\{b_1, \ldots, b_m\}$，那么对应的激活$\{a_1, \ldots, a_m\}$可以写作：

$$a_1 = \frac{1}{1+\exp(-(w^{(1)T}x+b_1))}$$
$$\vdots$$
$$a_m = \frac{1}{1+\exp(-(w^{(m)T}x+b_m))}$$

让我们定义下面简写来保持记号简单且会在复杂网络中更有用：

$$\sigma(z) = \begin{bmatrix} \frac{1}{1+\exp(z_1)} \\ \vdots \\ \frac{1}{1+\exp(z_m)} \end{bmatrix}$$
$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$
$$W = \begin{bmatrix} - & w^{(1)T} & - \\ & \dots & \\ - & w^{(m)T} & - \end{bmatrix} \in \mathbb{R}^{m\times n}$$

我们现在可以将未激活的输出写作：$z = Wx + b$

那么激活则写作：

$$\begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \sigma(z) = \sigma(Wx+b)$$

所以这些激活到底告诉了我们什么？你可以把这些激活看作是表现一些特征的加权结合的指示。我们后面可以用这些激活的结合去实现分类任务。

Neural Networks: Tips and Tricks

我们已经讨论了神经网络的数学基础，接下来我们会介绍在现实神经网络应用中常使用的一些贴士和技巧。

2.1 Gradient Check

在上一部分，我们详细讨论了如何通过微积分方法计算损失梯度并更新神经网络中的参数。现在我们介绍一种数值近似梯度的方法，即使直接用于训练模型计算会十分不高效，这个方法让我们可以十分准确的估计对于任何一个参数的导数。因此在检查我们计算的解析解是否正确的时候十分有用。给定一个带有参数向量θ和损失函数J的模型，θ_i附近的梯度可以简单的用centered difference formula进行计算：

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

其中ϵ是一个很小的数值（通常大约为1e-5）。$J(\theta^{(i+)})$这一项就是简单的计算在θ的第i项向前移动ϵ时error的值，反之亦然。因此使用两个方向的小移动，我们可以求出对于θ_i的近似导数。我们注意到上面的定义与数值梯度的定义不太一样：

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

当然这是一个很小的差别，上面这个定义只是往正方向做了位移去计算梯度。这样的定义也是完全可以接受的，但在实际应用中，往往第一种方式估算出来的梯度会更加精准且稳定。这个直觉来源于我们需要检验函数在向左和向右两个方向上的表现才能更好地去估计梯度。也可以看到使用泰勒定理，centered difference formula的误差只是成倍于ϵ^2，是十分小的，而后者定义更倾向于大误差。

现在，一个自然的问题：如果这个方法这么准确，那为什么不使用这个方法去计算网络的所有梯度，而是使用反向传播呢？原因很简单，因为这个方法计算十分不高效，每计算一个参数的梯度都需要两次前向传播，这是十分昂贵的。再者，许多大规模的神经网络包含了上百万个参数，每个参数都进行两次前向传播显然不是最优的。而且因为如SGD的优化算法，我们需要在上千次循环中的每一个循环都计算梯度，显然这样的方法会很快达到不可追踪。这种不高效性也决定了为什么我们只能拿数值计算的方法作为检测来用。一个标准的梯度检查的实现：

Snippet 2.1

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient
  at
  """

  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 Regularization

就如许多机器学习模型一样，神经网络也十分容易过拟合，一个很常见的解决过拟合的方法就是使用L2-正则，如下式：

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

当我们最小化J_R时，后面的正则项会阻止参数在训练过程中变得过于大，并且由于范数的特性，L2-正则可以有效地减少模型的自由度并且减少出现过拟合的可能。这样就是使得所有参数都接近于零，这雨贝叶斯信念先验是一致的。最终到底有多么接近零是由λ决定的，当其十分大的时候所有权重都会十分接近于零且难以从数据上学习到东西，当其十分小的时候，模型可能会再次过拟合。所以需要尝试调整λ的值。

还有其他的正则项，比如说L1-正则，然而它没那么常用因为它会导致权重参数变得稀疏。在下一节中我们会讨论dropout，它是另一个有效阻止过拟合的方法。

2.3 Dropout

Dropout是阻止过拟合的一个强有力的方法。主要思想很简单且有效-在训练过程中我们以一定的概率随机丢弃神经元的一个自己。。然后在测试中，我们用完整的模型去计算我们的预测。结果显示网络能够从数据上学习到更多信息，不那么容易出现过拟合并且在现实任务重都有更好的表现。一个直觉原因是dropout使得我们一次性训练了许多个更小的模型，并将它们平均起来作为预测。

在实际中，dropout通常是把每一层的输出h拿出来，然后每个神经元以概率p决定保留，否则将神经元设为0.然后在反向传播时，我们只把梯度传输给值不为0的神经元。最终，在测试时，我们用网络中的所有神经元做前向传播。然而一个关键的问题时为了使dropout高效地工作，一个神经元在测试中的输出应该与训练中的输出相近，否则输出的量级会大有不同，并且网络的行为则不具有好的定义。因此在测试中，我们需要把每个神经元的输出除以一个固定的值。

2.4 Neuron Units

这部分主要是介绍几个激活函数，这里直接使用截图：

Sigmoid: This is the default choice we have discussed; the activation function σ is given by:

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

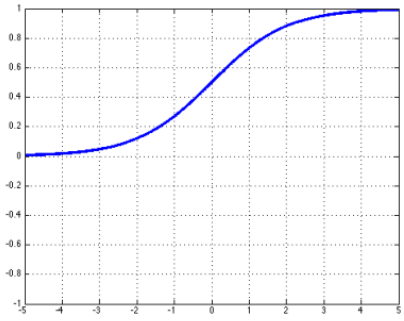

Figure 9: The response of a sigmoid nonlinearity

$$\text{where } \sigma(z) \in (0,1)$$

The gradient of $\sigma(z)$ is:

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

Tanh: The tanh function is an alternative to the sigmoid function that is often found to converge faster in practice. The primary difference between tanh and sigmoid is that tanh output ranges from -1 to 1 while the sigmoid ranges from 0 to 1.

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

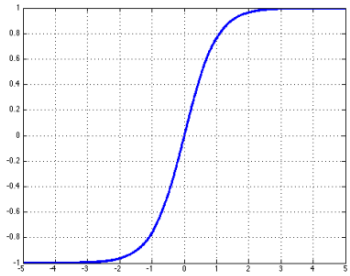

Figure 10: The response of a tanh nonlinearity

The gradient of $\tanh(z)$ is:

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

Hard tanh: The hard tanh function is sometimes preferred over the tanh function since it is computationally cheaper. It does however saturate for magnitudes of z greater than 1. The activation of the hard tanh is:

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

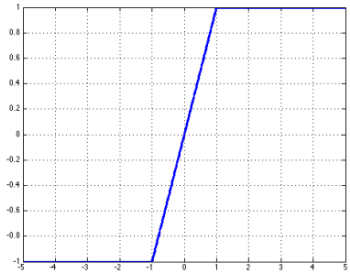

Figure 11: The response of a hard tanh nonlinearity

The derivative can also be expressed in a piecewise functional form:

$$\begin{cases} 1 & : -1 < z < 1 \end{cases}$$

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

Soft sign: The soft sign function is another nonlinearity which can be considered an alternative to tanh since it too does not saturate as easily as hard clipped functions:

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

The derivative is the expressed as:

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

where sgn is the signum function which returns ± 1 depending on the sign of z

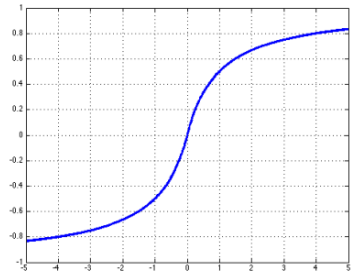

Figure 12: The response of a soft sign nonlinearity

ReLU: The ReLU (Rectified Linear Unit) function is a popular choice of activation since it does not saturate even for larger values of z and has found much success in computer vision applications:

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

The derivative is then the piecewise function:

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

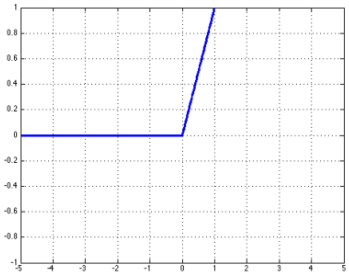

Figure 13: The response of a ReLU nonlinearity

Leaky ReLU: Traditional ReLU units by design do not propagate any error for non-positive z – the leaky ReLU modifies this such that a small error is allowed to propagate backwards even when z is negative:

$$\text{leaky}(z) = \max(z, k \cdot z)$$

$$\text{where } 0 < k < 1$$

This way, the derivative is representable as:

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

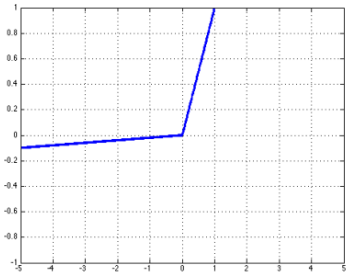

Figure 14: The response of a leaky ReLU nonlinearity

2.5 Data Preprocessing

常见的数据预处理方式有三种：

1. 消去均值：
   给定一组输入数据 X，一般把 X 中的值减去 X 的平均特征向量来使数据**零中心化**。在实践中很重要的一点是，**只计算训练集的平均值**，而且在**训练集，验证集和测试集**都是**减去同一平均值**。
2. 标准化：
   另外一个常见的技术（虽然没有 meanSubtractionmeanSubtraction 常用）是将每个输入特征维度缩小，让每个输入特征维度具有相似的幅度范围。这是很有用的，因此不同的输入特征是用不同“单位”度量，但是最初的时候我们经常认为所有的特征同样重要。实现方法是将特征除以它们各自在训练集中计算的标准差。

3. whitening（白化）
   相比上述的两个方法， **whitening** 没有那么常用，它本质上是数据经过转换后，特征之间相关性较低，所有特征具有相同的方差（协方差阵为 1 ）。首先对数据进行 **Mean Subtraction** 处理，得到 X' 。然后我们对 X'X' 进行奇异值分解得到矩阵 U , S , V ，计算UX' 将 X' 投影到由 U 的列定义的基上。我们最后将结果的每个维度除以 S 中的相应奇异值，从而适当地缩放我们的数据（如果其中有奇异值为 0 ，我们就除以一个很小的值代替）

2.6 Parameter Initialization

参数初始化在模型训练中是十分重要的，一个好的方式是将权重初始化为在0附近很小的正态分布随机数。另一个方式是提出不同的权重和偏置的初始化随着训练而动态变化。经验发现，对于sigmoid和tanh激活，更快且更低误差率的初始化方式是使用下面的均匀分布：

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

其中$n^{(l)} = fan-in, n^{(l+1)} = fan-out$，偏置设置为0.这个方法尝试保持激活函数的方差，同时通过反向传播将梯度方差传到各个层。没有这样的初始化，梯度方差会随着反向传播而减小。

2.7 Learning Strategies

学习率很重要，主要有几个方面：

1. 学习率大小的影响：
   学习率过大可能导致越过最优点。
   学习率过小可能导致无法在限时内完成收敛。
   所以固定学习率是需要经过尝试调整的。
2. 另一种方法是根据权重的维度调整学习率：
   Ronan Collobert 通过取 fan-in 的神经元 $n^{(l)}$ 的平方根的倒数来缩放权值 W_{ij} ($W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$) 的学习率。
3. 最后是一种已经背证明有效的方法**退火（annealing）**，在多次迭代之后学习率以下面这种方式减小：保证以一个高的的学习率开始训练和快速逼近最小值；当越来越接近最小值时，开始降低学习率，让我们可以在更细微的范围内找到最优值。一个常见的实现 **annealing** 的方法是在每 n 次的迭代学习后，通过一个因子 x 来降低学习率 α 。指数衰减也是很常见的方式，在t次迭代后学习率变为$\alpha(t) = \alpha_0 e^{-kt}$，其中$\alpha_0$和$k$都是超参数。还有另一种允许学习率随着时间减少的方法：$\alpha(t) = \frac{\alpha_0 \tau}{max(t,\tau)}$，$\alpha_0$是一个可调参数，代表起始学习率，$\tau$也是一个可调的参数，表示学习率应该在该时间点开始减小。在实际中这个方法很有效。

2.8 Momentum Updates

动量方法，灵感来自于物理学中的对动力学的研究，是梯度下降方法的一种变体，尝试使用更新的“速度”的一种更有效的更新方案。动量更新的伪代码如下所示：

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

2.9 Adaptive Optimization Methods

AdaGrad 是标准的随机梯度下降(**SGD**)的一种实现，但是有一点关键的不同：对每个参数学习率是不同的。每个参数的学习率取决于每个参数梯度更新的历史，参数的历史更新越小，就使用更大的学习率加快更新。换句话说，过去没有更新太大的参数现在更有可能有更高的学习率。

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

这个技术中，我们可以看到如果历史梯度的RMS十分小，那么学习率就会变得十分大，一个简单的实现如下：

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

其他常见的自适应方法是RMSProp和Adam，他们的更新规则如下所示：

```
# Update rule for RMSProp
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的一个变种，它利用了梯度平方的移动平均，特别的，不像AdaGrad，它的更新不会逐渐变小。Adam的更新规则是RMSProp的一个变种，但是加入了类似动量的更新。