[**深度学习与计算机视觉系列(6)\_神经网络结构与神经元激励函数**](https://blog.csdn.net/yaoqiang2011/article/details/50447834)

作者：[寒小阳](http://blog.csdn.net/han_xiaoyang?viewmode=contents)   
时间：2016年1月。   
出处：<http://blog.csdn.net/han_xiaoyang/article/details/50447834>   
声明：版权所有，转载请联系作者并注明出处

## 1.神经元与含义

大家都知道最开始深度学习与神经网络，是受人脑的神经元启发设计出来的。所以我们按照惯例也交代一下背景，从生物学的角度开始介绍，当然也是对神经网络研究的先驱们致一下敬。

## 1.神经元与含义

大家都知道最开始深度学习与神经网络，是受人脑的神经元启发设计出来的。所以我们按照惯例也交代一下背景，从生物学的角度开始介绍，当然也是对神经网络研究的先驱们致一下敬。

### 1.1 神经元激励与连接

大家都知道，人脑的基本计算单元叫做神经元。现代生物学表明，人的神经系统中大概有860亿神经元，而这数量巨大的神经元之间大约是通过1014−1015

个突触连接起来的。下面有一幅示意图，粗略地描绘了一下人体神经元与我们简化过后的数学模型。每个神经元都从树突接受信号，同时顺着某个轴突传递信号。而每个神经元都有很多轴突和其他的神经元树突连接。而我们可以看到右边简化的神经元计算模型中，信号也是顺着轴突(比如*x*0)传递，然后在轴突处受到激励(*w*0倍)然后变成*w*0*x*0。我们可以这么理解这个模型：在信号的传导过程中，**突触可以控制传导到下一个神经元的信号强弱(数学模型中的权重*w***

**)，而这种强弱是可以学习到的**。在基本生物模型中，树突传导信号到神经元细胞，然后这些信号被加和在一块儿了，如果加和的结果被神经元感知超过了某种阈值，那么神经元就被激活，同时沿着轴突向下一个神经元传导信号。在我们简化的数学计算模型中，**我们假定有一个『激励函数』来控制加和的结果对神经元的刺激程度，从而控制着是否激活神经元和向后传导信号**。比如说，我们在逻辑回归中用到的sigmoid函数就是一种激励函数，因为对于求和的结果输入，sigmoid函数总会输出一个0-1之间的值，我们可以认为这个值表明信号的强度、或者神经元被激活和传导信号的概率。


下面是一个简单的程序例子，表明前向传播中单个神经元做的事情：

class Neuron:

# ...

def forward(inputs):

"""

假定输入和权重都是1维的numpy数组，同时bias是一个数

"""

cell\_body\_sum = np.sum(inputs \* self.weights) + self.bias

firing\_rate = 1.0 / (1.0 + math.exp(-cell\_body\_sum)) # sigmoid activation function

return firing\_rate

稍加解释，每个神经元对于输入和权重做内积，加上偏移量bias，然后通过激励函数(比如说这里是sigmoid函数)，然后输出结果。

**特别说明**：实际生物体内的神经元相当复杂，比如说，神经元的种类就灰常灰常多，它们分别有不同的功能。而加和信号之后的激励函数的非线性变换，也比数学上模拟出来的函数复杂得多。我们用数学建模的神经网络只是一个非常简化后的模型，有兴趣的话你可以阅读[材料1](https://physics.ucsd.edu/neurophysics/courses/physics_171/annurev.neuro.28.061604.135703.pdf)或者[材料2](http://www.sciencedirect.com/science/article/pii/S0959438814000130)。

### 1.2 单个神经元的分类作用

以sigmoid函数作为神经元的激励函数为例，这个大家可能稍微熟悉一点，毕竟我们[逻辑回归部分](http://blog.csdn.net/han_xiaoyang/article/details/49123419)重点提到了这个非线性的函数，把输入值压缩成0-1之间的一个概率值。而通过这个非线性映射和设定的阈值，我们可以把空间切分开，分别对应正样本区域和负样本区域。而对应回现在的神经元场景，我们如果稍加拟人化，可以认为神经元具备了**喜欢**(概率接近1)和**不喜欢**(概率接近0)线性划分的某个空间区域的能力。这也就是说，只要调整好权重，单个神经元可以对空间做线性分割。

**二值Softmax分类器**   
对于Softmax分类器详细的内容欢迎参见[前面的博文系列](http://blog.csdn.net/han_xiaoyang/article/details/49999583)，我们标记*σ*

为sigmoid映射函数，则*σ*(∑*iwixi*+*b*)可视作**二分类问题**中属于某个类的概率*P*(*yi*=1∣*xi*;*w*)，当然，这样我们也可以得到相反的那个类别的概率为*P*(*yi*=0∣*xi*;*w*)=1−*P*(*yi*=1∣*xi*;*w*)。根据前面博文提到的知识，我们可以使用交叉熵损失作为这个二值线性分类器的损失函数(loss function)，而最优化损失函数得到的一组参数*W*,*b*

，就能帮助我们将空间线性分割，得到二值分类器。当然，和逻辑回归中看到的一样，最后神经元预测的结果y值如果大于0.5，那我们会判定它属于这个类别，反之则属于另外一个类别。

**二值SVM分类器**   
同样的，我们可以设定max-margin hinge loss作为损失函数，从而将神经元训练成一个二值支持向量机分类器。详细的内容依旧欢迎大家查看[之前的博客](http://blog.csdn.net/han_xiaoyang/article/details/49999583)。

**对于正则化的解释**   
对于正则化的损失函数(不管是SVM还是Softmax)，其实我们在神经元的生物特性上都能找到对应的解释，我们可以将其(正则化项的作用)视作信号在神经元传递过程中的逐步淡化/衰减(gradual forgetting)，因为正则化项的作用是在每次迭代过程中，控制住权重*w*

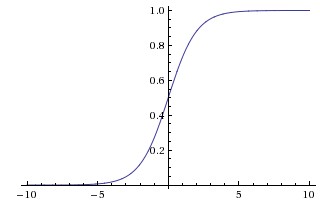
的幅度，往0上靠拢。

单个神经元的作用，可视作完成一个二分类的分类器(比如Softmax或者SVM分类器)

### 1.3 常用激励函数

每一次输入和权重w线性组合之后，都会通过一个激励函数(也可以叫做非线性激励函数)，经非线性变换后输出。实际的神经网络中有一些可选的激励函数，我们一一说明一下最常见的几种：

#### 1.3.1 sigmoid



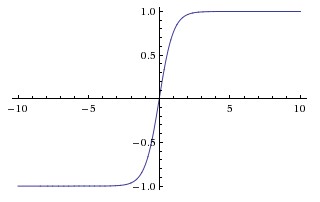
sigmoid函数提到的次数太多，相信大家都知道了。数学形式很简单，是*σ*(*x*)=1/(1+*e*−*x*)，图像如上图所示，功能是把一个实数压缩至0到1之间。输入的数字非常大的时候，结果会接近1，而非常大的负数作为输入，则会得到接近0的结果。不得不说，早期的神经网络中，sigmoid函数作为激励函数使用非常之多，因为大家觉得它很好地解释了神经元受到刺激后是否被激活和向后传递的场景(从几乎没有被激活，也就是0，到完全被激活，也就是1)。不过似乎近几年的实际应用场景中，比较少见到它的身影，它主要的缺点有2个：

* sigmoid函数在实际梯度下降中，容易**饱和**和**终止梯度传递**。我们来解释一下，大家知道反向传播过程，依赖于计算的梯度，在一元函数中，即斜率。而在sigmoid函数图像上，大家可以很明显看到，在纵坐标接近0和1的那些位置(也就是输入信号的幅度很大的时候)，斜率都趋于0了。我们回想一下反向传播的过程，我们最后用于迭代的梯度，是由中间这些梯度值结果相乘得到的，因此如果中间的局部梯度值非常小，直接会把最终梯度结果拉近0，也就是说，残差回传的过程，因为sigmoid函数的**饱和**被**杀死了**。说个极端的情况，如果一开始初始化权重的时候，我们取值不是很恰当，而激励函数又全用的sigmoid函数，那么很有可能神经元一个不剩地**饱和**到**无法学习**，整个神经网络也根本没办法训练起来。
* sigmoid函数的输出没有0中心化，这是一个比较闹心的事情，因为每一层的输出都要作为下一层的输入，而未0中心化会直接影响梯度下降，我们这么举个例子吧，如果输出的结果均值不为0，举个极端的例子，全部为正的话(例如*f*=*wTx*+*b*

中所有*x*>0)，那么反向传播回传到*w*

* 上的梯度将要么全部为正要么全部为负(取决于f的梯度正负性)，这带来的后果是，反向传播得到的梯度用于权重更新的时候，不是平缓地迭代变化，而是类似锯齿状的突变。当然，要多说一句的是，这个缺点相对于第一个缺点，还稍微好一点，第一个缺点的后果是，很多场景下，神经网络根本没办法学习。

#### 1.3.2 Tanh



Tanh函数的图像如上图所示。它会将输入值压缩至-1到1之间，当然，它同样也有sigmoid函数里说到的第一个缺点，在很大或者很小的输入值下，神经元很容易饱和。但是它缓解了第二个缺点，它的输出是0中心化的。所以在实际应用中，tanh激励函数还是比sigmoid要用的多一些的。

#### 1.3.3 ReLU

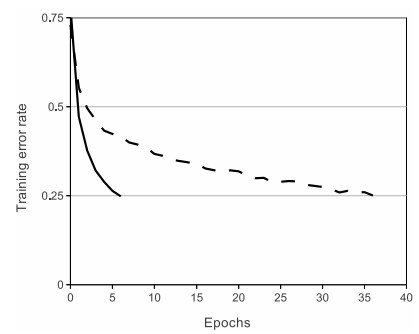


ReLU是修正线性单元(The Rectified Linear Unit)的简称，近些年使用的非常多，图像如上图所示。它对于输入x计算*f*(*x*)=*max*(0,*x*)

。换言之，以0为分界线，左侧都为0，右侧是y=x这条直线。

它有它对应的优势，也有缺点：

* 优点1：实验表明，它的使用，相对于sigmoid和tanh，可以非常大程度地提升随机梯度下降的收敛速度。不过有意思的是，很多人说，这个结果的原因是它是线性的，而不像sigmoid和tanh一样是非线性的。具体的收敛速度结果对比如下图，收敛速度大概能快上6倍：



* 优点2：相对于tanh和sigmoid激励神经元，求梯度不要简单太多好么！！！毕竟，是线性的嘛。。。
* 缺点1：ReLU单元也有它的缺点，在训练过程中，它其实挺脆弱的，有时候甚至会**挂掉**。举个例子说吧，如果一个很大的梯度流经ReLU单元，那权重的更新结果可能是，在此之后任何的数据点都没有办法再激活它了。一旦这种情况发生，那本应经这个ReLU回传的梯度，将永远变为0。当然，这和参数设置有关系，所以我们要特别小心，再举个实际的例子哈，如果学习速率被设的太高，结果你会发现，训练的过程中可能有高达40%的ReLU单元都挂掉了。所以我们要小心设定初始的学习率等参数，在一定程度上控制这个问题。

#### 1.3.4 Leaky ReLU

上面不是提到ReLU单元的弱点了嘛，所以孜孜不倦的ML researcher们，就尝试修复这个问题咯，他们做了这么一件事，在x<0的部分，leaky ReLU不再让y的取值为0了，而是也设定为一个坡度很小(比如斜率0.01)的直线。f(x)因此是一个分段函数，x<0时，*f*(*x*)=*αx*

(*α*是一个很小的常数)，x>0时，*f*(*x*)=*x*

。有一些researcher们说这样一个形式的激励函数帮助他们取得更好的效果，不过似乎并不是每次都比ReLU有优势。

#### 1.3.5 Maxout

也有一些其他的激励函数，它们并不是对*WTX*+*b*

做非线性映射*f*(*WTX*+*b*)。一个近些年非常popular的激励函数是Maxout(详细内容请参见[Maxout](http://www-etud.iro.umontreal.ca/~goodfeli/maxout.html))。简单说来，它是ReLU和Leaky ReLU的一个泛化版本。对于输入x，Maxout神经元计算max(*wT*1*x*+*b*1,*wT*2*x*+*b*2)。有意思的是，如果你仔细观察，你会发现ReLU和Leaky ReLU都是它的一个特殊形式(比如ReLU，你只需要把*w*1,*b*1

设为0)。因此Maxout神经元继承了ReLU单元的优点，同时又没有『一不小心就挂了』的担忧。如果要说缺点的话，你也看到了，相比之于ReLU，因为有2次线性映射运算，因此计算量也double了。

### 1.4 激励函数/神经元小总结

以上就是我们总结的常用的神经元和激励函数类型。顺便说一句，即使从计算和训练的角度看来是可行的，实际应用中，其实我们很少会把多种激励函数混在一起使用。

那我们咋选用神经元/激励函数呢？一般说来，用的最多的依旧是ReLU，但是我们确实得小心设定学习率，同时在训练过程中，还得时不时看看神经元此时的状态(是否还『活着』)。当然，如果你非常担心神经元训练过程中挂掉，你可以试试Leaky ReLU和Maxout。额，少用sigmoid老古董吧，有兴趣倒是可以试试tanh，不过话说回来，通常状况下，它的效果不如ReLU/Maxout。

## 2. 神经网络结构

### 2.1 层级连接结构

神经网络的结构其实之前也提过，是一种单向的层级连接结构，每一层可能有多个神经元。再形象一点说，就是每一层的输出将会作为下一层的输入数据，当然，这个图一定是没有循环的，不然数据流就有点混乱了。一般情况下，单层内的这些神经元之间是没有连接的。最常见的一种神经网络结构就是**全连接层级**神经网络，也就是相邻两层之间，每个神经元和每个神经元都是相连的，单层内的神经元之间是没有关联的。下面是两个全连接层级神经网的示意图：


**命名习俗**   
有一点需要注意，我们再说N层神经网络的时候，通常的习惯是不把输入层计算在内，因此输入层直接连接输出层的，叫做单层神经网络。从这个角度上说，其实我们的**逻辑回归**和**SVM**是单层神经网络的特例。上图中两个神经网络分别是2层和3层的神经网络。

**输出层**   
输出层是神经网络中比较特殊的一层，由于输出的内容通常是各类别的打分/概率(在分类问题中)，我们通常都不在输出层神经元中加激励函数。

**关于神经网络中的组件个数**   
通常我们在确定一个神经网络的时候，有几个描述神经网络大小的参数会提及到。最常见的两个是神经元个数，以及细化一点说，我们可以认为是参数的个数。还是拿上面的图举例：

* 第一个神经网络有4+2=6个神经元(我们不算输入层)，因此有[3\*4]+[4\*2]=20个权重和4+2=6个偏移量(bias项)，总共26个参数。
* 第二个神经网络有4+4+1个神经元，有[3\*4]+[4\*4]+[4\*1]=32个权重，再加上4+4+1=9个偏移量(bias项)，一共有41个待学习的参数。

给大家个具体的概念哈，现在实用的卷积神经网，大概有亿级别的参数，甚至可能有10-20层(因此是深度学习嘛)。不过不用担心这么多参数的训练问题，因此我们在卷积神经网里会有一些有效的方法，来共享参数，从而减少需要训练的量。

### 2.2 神经网络的前向计算示例

神经网络组织成以上的结构，一个重要的原因是，每一层到下一层的计算可以很方便地表示成矩阵之间的运算，就是一直重复**权重和输入做内积后经过激励函数变换的过程**。为了形象一点说明，我们还举上面的3层神经网络为例，输入是一个3\*1的向量，而层和层之间的连接权重可以看做一个矩阵，比如第一个隐藏层的权重*W*1

是一个[4\*3]的矩阵，偏移量*b*1

是[4\*1]的向量，因此用python中的numpy做内积操作np.dot(W1,x)实际上就计算出输入下一层的激励函数之前的结果，经激励函数作用之后的结果又作为新的输出。用简单的代码表示如下：

# 3层神经网络的前向运算:

f = lambda x: 1.0/(1.0 + np.exp(-x)) # 简单起见，我们还是用sigmoid作为激励函数吧

x = np.random.randn(3, 1) # 随机化一个输入

h1 = f(np.dot(W1, x) + b1) # 计算第一层的输出

h2 = f(np.dot(W2, h1) + b2) # 计算第二层的输出

out = np.dot(W3, h2) + b3 # 最终结果 (1x1)

* 1
* 2
* 3
* 4
* 5
* 6

上述代码中，W1,W2,W3,b1,b2,b3都是**待学习**的神经网络参数。注意到我们这里所有的运算都是向量化/矩阵化之后的，x不再是一个数，而是包含训练集中一个batch的输入，这样并行运算会加快计算的速度，仔细看代码，最后一层是没有经过激励函数，直接输出的。

### 2.3 神经网络的表达力与size

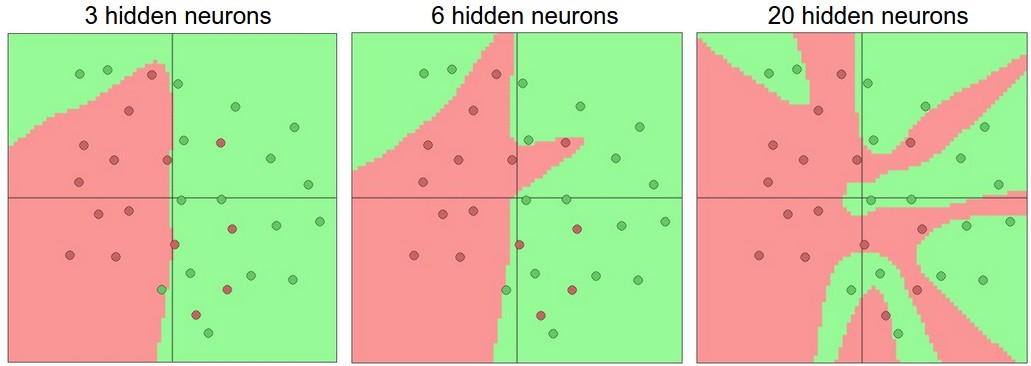
一个神经网络结构搭起来之后，它就包含了数以亿计的参数和函数。我们可以把它看做对输入的做了一个很复杂的函数映射，得到最后的结果用于完成空间的分割(分类问题中)。那我们的参数对于这个所谓的复杂映射有什么样的影响呢？

其实，包含一个隐藏层(2层神经网络)的神经网络已经具备大家期待的能力，即只要隐藏层的神经元个数足够，我们总能用它(2层神经网络)去逼近任何连续函数(即输入到输出的映射关系)。详细的内容可以参加[Approximation by Superpositions of Sigmoidal Function](http://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf)或者[Michael Nielsen的介绍](http://neuralnetworksanddeeplearning.com/chap4.html)。我们之前的博文[手把手入门神经网络系列(1)\_从初等数学的角度初探神经网络](http://blog.csdn.net/han_xiaoyang/article/details/50100367)也有提到。

问题是，如果单隐藏层的神经网络已经可以近似逼近任意的连续值函数，那么为什么我们还要用那么多层呢？很可惜的是，即使数学上我们可以用2层神经网近似几乎所有函数，但在实际的工程实践中，却是没啥大作用的。**多隐藏层的神经网络比单隐藏层的神经网络工程效果好很多**，即使从数学上看，表达能力应该是一致的。

不过还得说一句的是，通常情况下，**我们工程中发现，3层神经网络效果优于2层神经网络，但是如果把层数再不断增加(4,5,6层)，对最后结果的帮助就没有那么大的跳变了。**不过在卷积神经网上还是不一样的，**深层的网络结构对于它的准确率有很大的帮助**，直观理解的方式是，图像是一种深层的结构化数据，因此深层的卷积神经网络能够更准确地把这些层级信息表达出来。

### 2.4 层数与参数设定的影响

一个很现实的问题是，我们拿到一个实际问题的时候，怎么知道应该如何去搭建一个网络结构，可以最好地解决这个问题？应该搭建几层？每一层又应该有多少个神经元？   
我们直观理解一下这个问题，当我们加大层数以及每一层的神经元个数的时候，我们的神经网络容量变大了。更通俗一点说，**神经网络的空间表达能力变得更丰富了**。放到一个具体的例子里我们看看，假如我们现在要处理一个2分类问题，输入是2维的，我们训练3个不同神经元个数的单隐层神经网络，它们的平面表达能力对比画出来如下：   


在上图中，我们可以看出来，更多的神经元，让神经网络有更好的拟合复杂空间函数的能力。但是任何事物都有双面性，拟合越来越精确带来的另外一个问题是，**太容易过拟合了！！！**，如果你很任性地做一个实验，在隐藏层中放入20个神经元，那对于上图这个一个平面，你完全可以做到100%把两类点分隔开，但是这样一个分类器太努力地学习和记住我们现在图上的这些点的分布状况了，以至于连噪声和离群点都被它学习下来了，这对于我们在新数据上的泛化能力，也是一个噩梦。

经我们上面的讨论之后，也许你会觉得，好像对于不那么复杂的问题，我们用更少数目的层数和神经元，会更不容易过拟合，效果好一些。但是**这个想法是错误的！！！**。**永远不要用减少层数和神经元的方法来缓解过拟合！！！这会极大影响神经网络的表达能力！！！我们有其他的方法，比如说之前一直提到的正则化来缓解这个问题。**

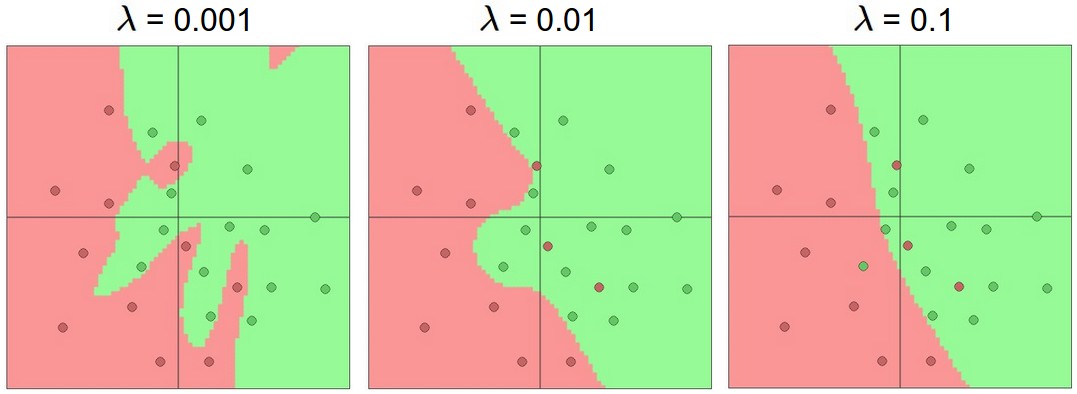
不要使用少层少神经元的简单神经网络的另外一个原因是，其实我们用梯度下降等方法，在这种简单神经网上，更难训练得到合适的参数结果。对，你会和我说，简单神经网络的损失函数有更少的局部最低点，应该更好收敛。是的，确实是的，更好收敛，但是很快收敛到的这些个局部最低点，通常都是全局很差的。相反，大的神经网络，确实损失函数有更多的局部最低点，但是这些局部最低点，相对于上面的局部最低点，在实际中效果却更好一些。对于非凸的函数，我们很难从数学上给出100%精准的性质证明，大家要是感兴趣的话，可以参考论文[The Loss Surfaces of Multilayer Networks](http://arxiv.org/abs/1412.0233)。

如果你愿意做多次实验，会发现，训练小的神经网络，最后的损失函数收敛到的最小值变动非常大。这意味着，如果你运气够好，那你maybe能找到一组相对较为合适的参数，但大多数情况下，你得到的参数只是在一个不太好的局部最低点上的。相反，大的神经网络，依旧不能保证收敛到最小的全局最低点，但是众多的局部最低点，都有相差不太大的效果，这意味着你不需要借助”运气”也能找到一个近似较优的参数组。

最后，我们提一下正则化，我们说了要用正则化来控制过拟合问题。正则话的参数是*λ*

，它的大小体现我们对参数搜索空间的限制，设置小的话，参数可变动范围大，同时更可能过拟合，设置太大的话，对参数的抑制作用太强，以至于不太能很好地表征类别分布了。下图是我们在上个问题中，使用不同大小的正则化参数*λ*

得到的平面分割结果。



恩，总之一句话，我们在很多实际问题中，还是得使用多层多神经元的大神经网络，而使用正则化来减缓过拟合现象。

## 3. 其他参考资料

* [Theano的深度学习导读](http://www.deeplearning.net/tutorial/mlp.html)
* [Michael Nielsen的神经网络导论](http://neuralnetworksanddeeplearning.com/chap1.html)
* [ConvNetJS demo](http://cs.stanford.edu/people/karpathy/convnetjs/)

## 参考资料与原文

[cs231n 神经网络结构与神经元激励函数](http://cs231n.github.io/neural-networks-1/)