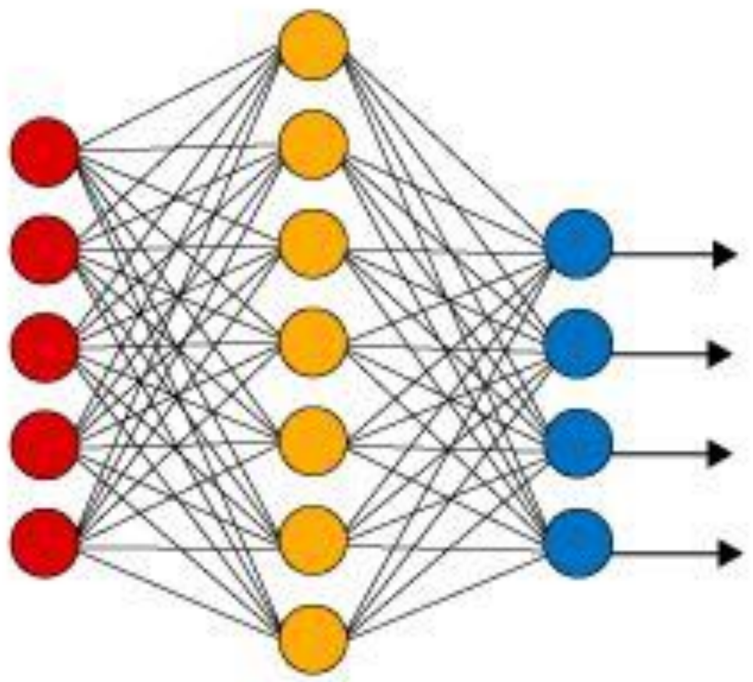
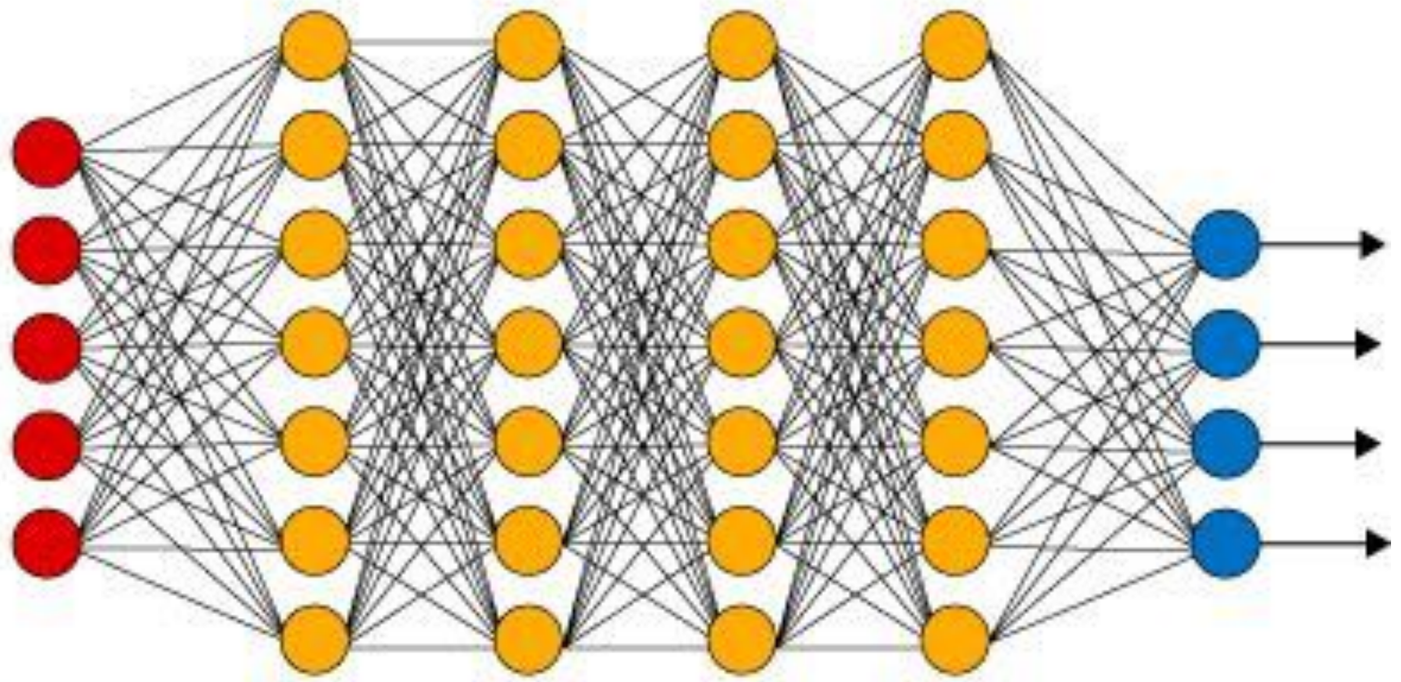
深度神经网络

简单神经网络



深度神经网络



🔴 输入层 🟠 隐藏层 🔵 输出层

深度神经网络面临的挑战

梯度消亡（Gradient Vanishing)：

训练过程非常慢

过拟合 （Overfitting)

在训练数据上面的效果好，在实际的测试数据上面效果差

梯度消亡(Gradient Vanishing)现象

神经网络靠输入端的网络层的系数逐渐不再随着训练而变化，或者变化非常缓慢

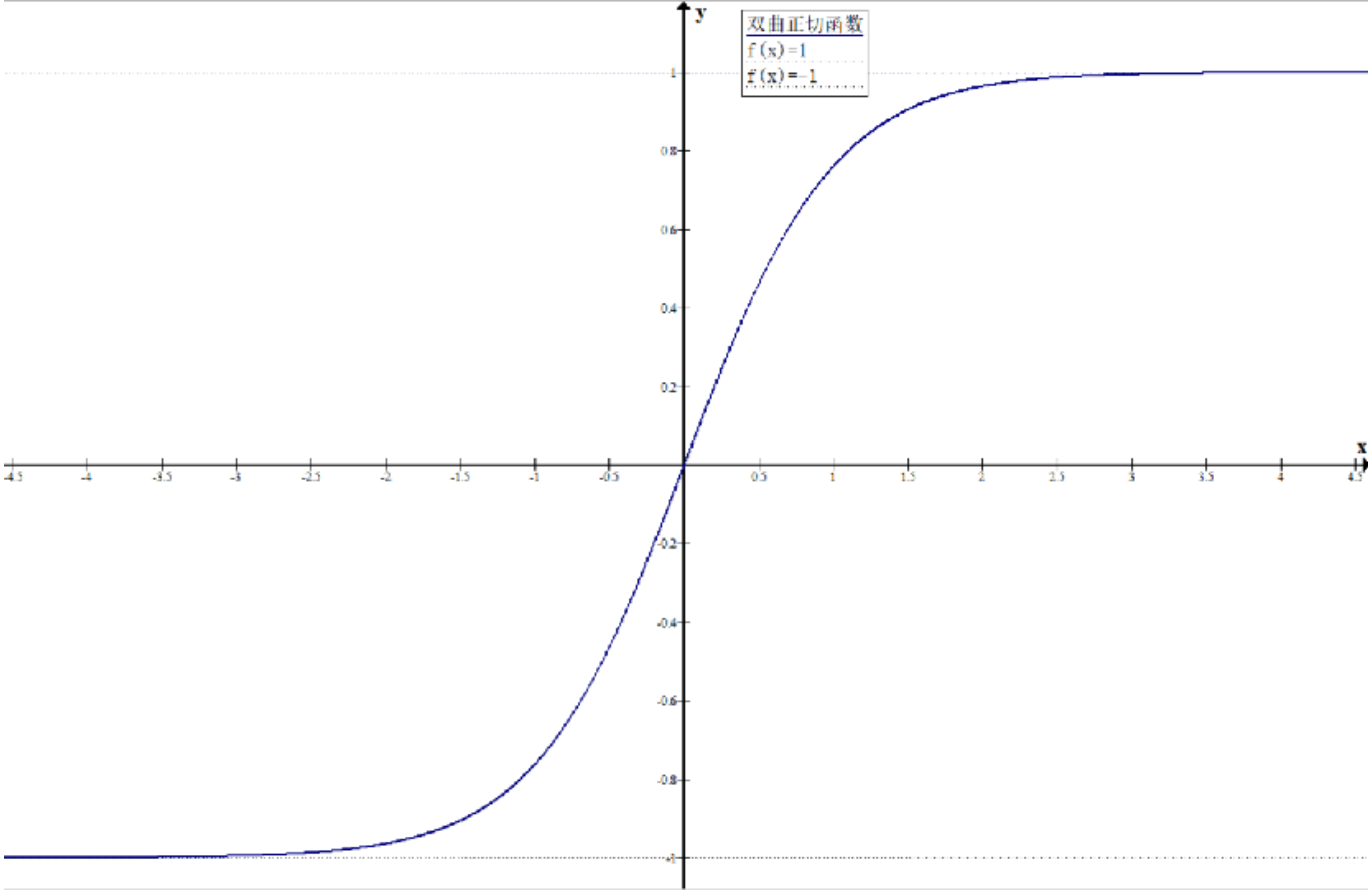
随着网络层数增加，这个现象越发明显

梯度消亡(Gradient Vanishing)

Sepp Hochreiter (Germany)1991年系统分析了梯度消亡的原因， 他也是LSTM的发明人

梯度消亡(Gradient Vanishing)前提

1. 使用基于梯度的训练方法（例如梯度下降法）
2. 使用的激活函数具有输出值范围大大小于输入值的范围，例如logistic（逻辑斯函数），tanh（双曲正切）



双曲正切函数
f(x)=1
f(x)=-1
y
x
1
0.8
0.6
0.4
0.2
-0.2
-0.4
-0.6
-0.8
-1
-4.5
-4
-3.5
-3
-2.5
-2
-1.5
-1
-0.5
0.5
1
1.5
2
2.5
3
3.5
4
4.5

梯度消亡(Gradient Vanishing)问题分析

梯度下降法依靠理解系数的微小变化对输出的影响来学习网络的系数的值。

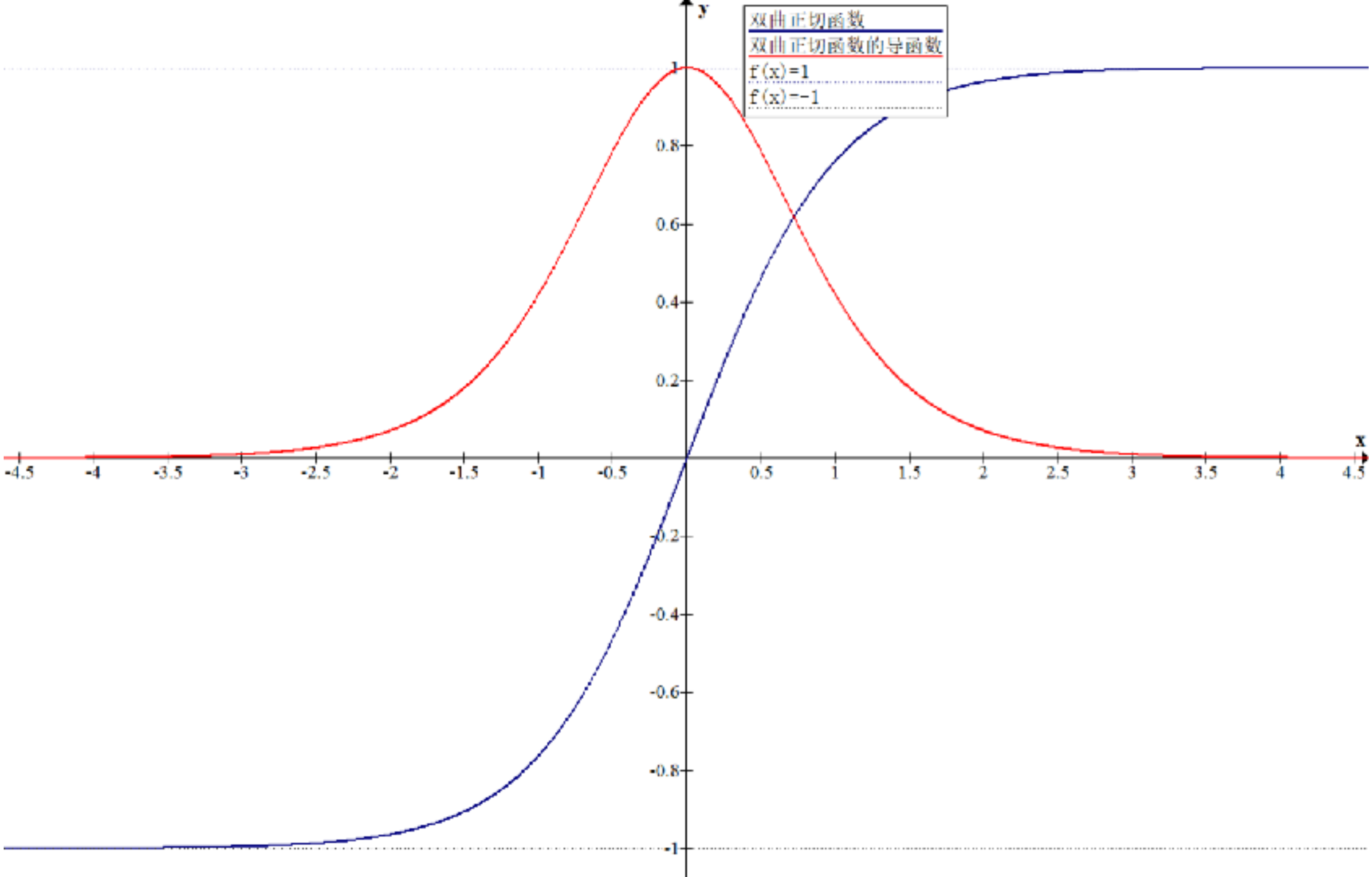
如果一个系数的微小变化对网络的输出没有影响或者影响极小，那么就无法知晓如何优化这个系数，或者优化特别慢。造成训练的困难。

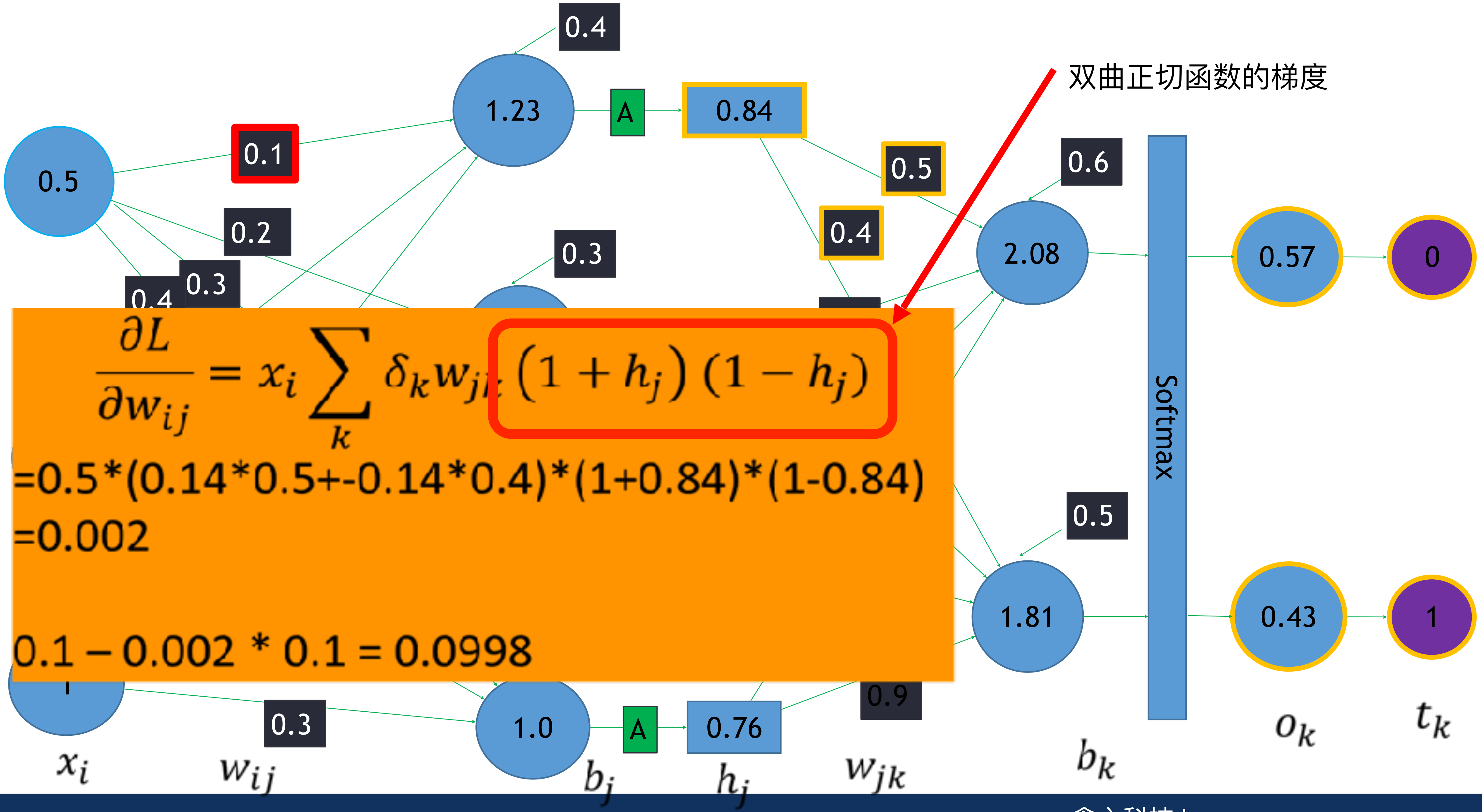
梯度消亡(Gradient Vanishing)原因

•

使用梯度下降法训练神经网络，如果激活函数具备将输出值的范围相对于输入的值大幅度压缩，那么就会出现梯度消亡。

例如，双曲正切函数（tanh）将−∞到∞的输入压缩到-1到+1之间。除开在输入为-6,+6之间的值，其它输入值对应的梯度都非常小，接近0.


y
x
双曲正切函数
双曲正切函数的导函数
f(x)=1
f(x)=-1
1
0.8
0.6
0.4
0.2
-0.2
-0.4
-0.6
-0.8
-1
-4.5
-4
-3.5
-3
-2.5
-2
-1.5
-1
-0.5
0.5
1
1.5
2
2.5
3
3.5
4
4.5

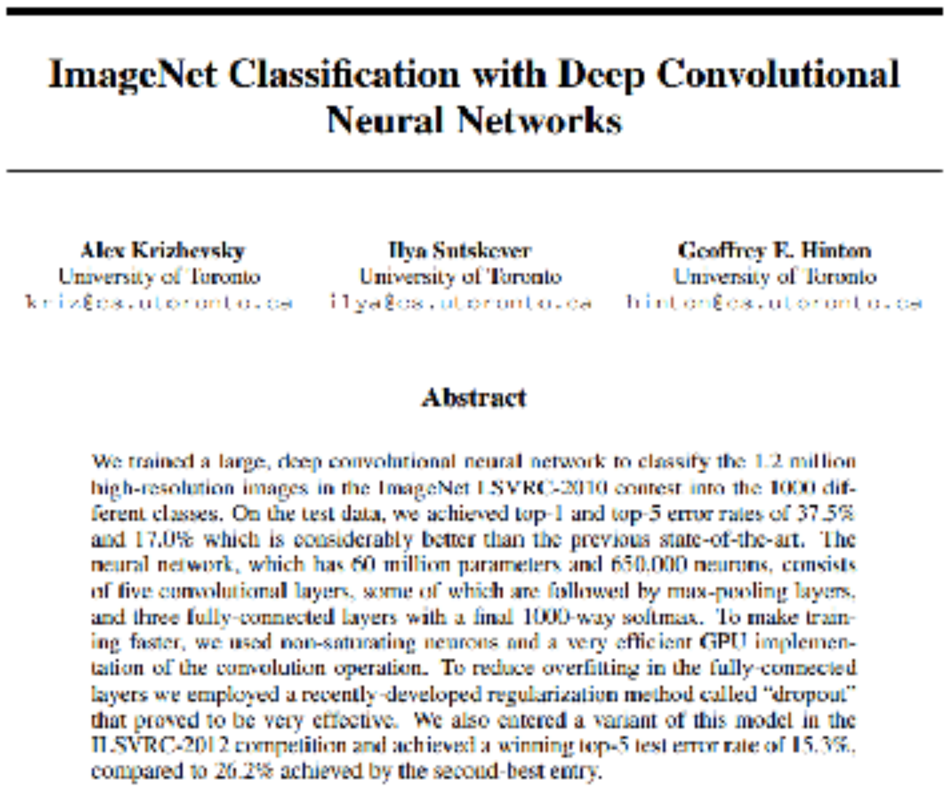


$$\frac{\partial L}{\partial w_{ij}} = x_i \sum_k \delta_k w_{jk} \boxed{(1 + h_j)(1 - h_j)}$$

=0.5*(0.14*0.5+-0.14*0.4)*(1+0.84)*(1-0.84)

=0.002

0.1 – 0.002 * 0.1 = 0.0998



ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky
University of Toronto
kriz@cs.utoronto.ca

Ilya Sutskever
University of Toronto
ilya@cs.utoronto.ca

Geoffrey E. Hinton
University of Toronto
hinton@cs.utoronto.ca

Abstract

We trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. On the test data, we achieved top-1 and top-5 error rates of 37.5% and 17.0% which is considerably better than the previous state-of-the-art. The neural network, which has 60 million parameters and 650,000 neurons, consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To make training faster, we used non-saturating neurons and a very efficient GPU implementation of the convolution operation. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%, compared to 26.2% achieved by the second-best entry.

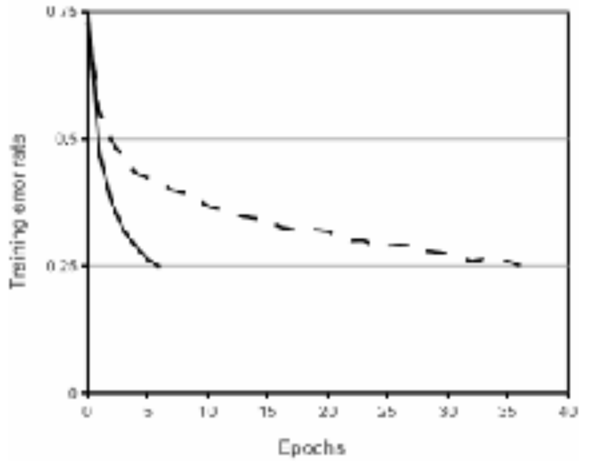


Figure 1: A four-layer convolutional neural network with ReLUs **(solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with tanh neurons **(dashed line)**. The learning rates for each network were chosen independently to make training as fast as possible. No regularization of any kind was employed. The magnitude of the effect demonstrated here varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

AlexNet is the name of a convolutional neural network, originally written with CUDA to run with GPU support, which competed in the ImageNet Large Scale Visual Recognition Challenge[1] in 2012. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points ahead of the runner up. AlexNet was designed by the SuperVision group, consisting of **Alex Krizhevsky**, Geoffrey Hinton, and Ilya Sutskever

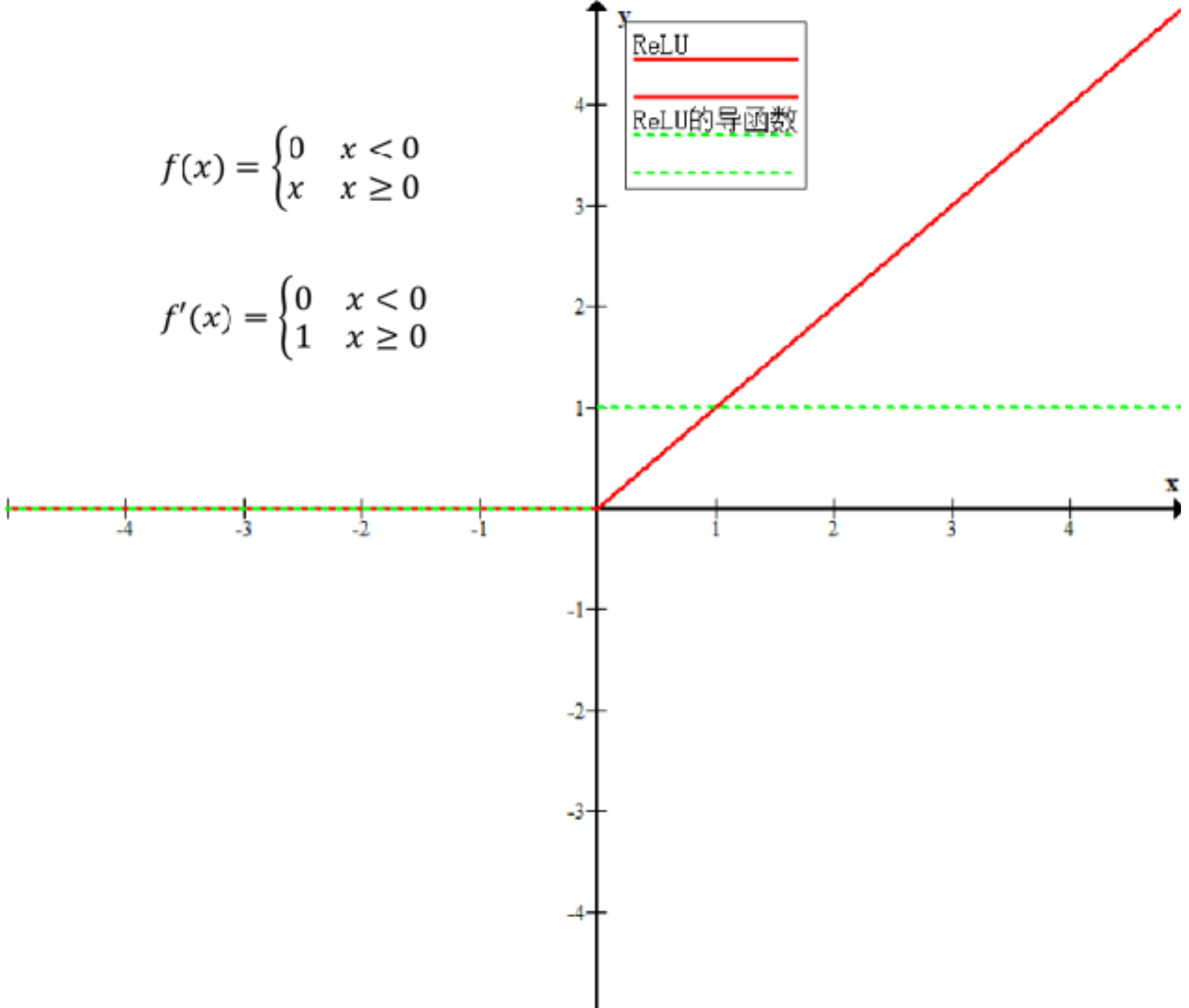
梯度消亡(Gradient Vanishing) 解决方案

- 1. 激活函数 ReLu： $f(x) = \max(0, x)$
  输入大于0， 梯度为1， 否则0.

  2. 激活函数 LeakyReLu： $f(x) = \max(ax, x)$
  输入大于等于0， 梯度为1， 否则为a

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

$$f'(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$



梯度消亡(Gradient Vanishing)思考题

- 为什么不选择 激活函数： $f(x) = x$?

梯度消亡(Gradient Vanishing)解决方案

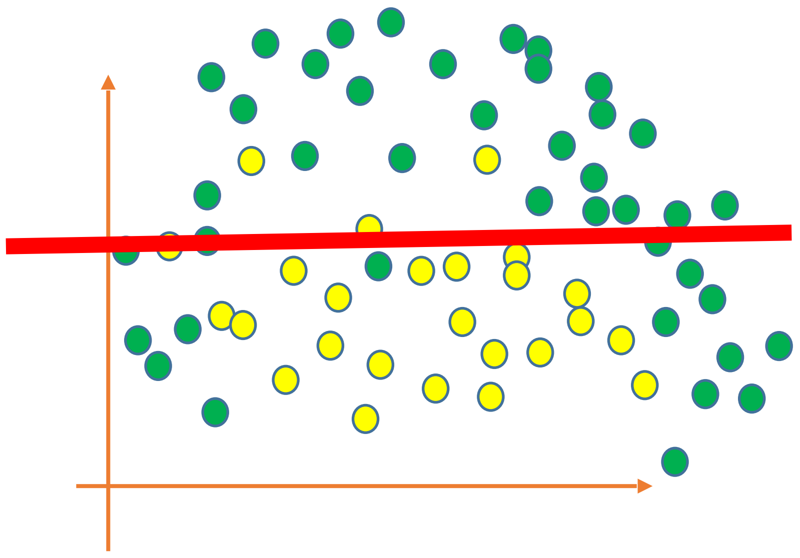
采用不使用用梯度的网络训练方法：https://link.springer.com/article/10.1007/s10898-012-9951-y(Derivative-free optimization: a review of algorithms and comparison of software implementations)

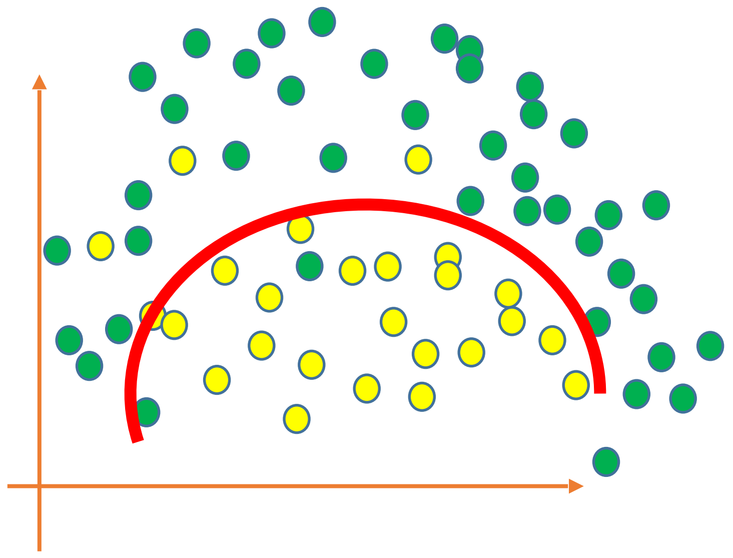
1. 基于遗传、进化算法

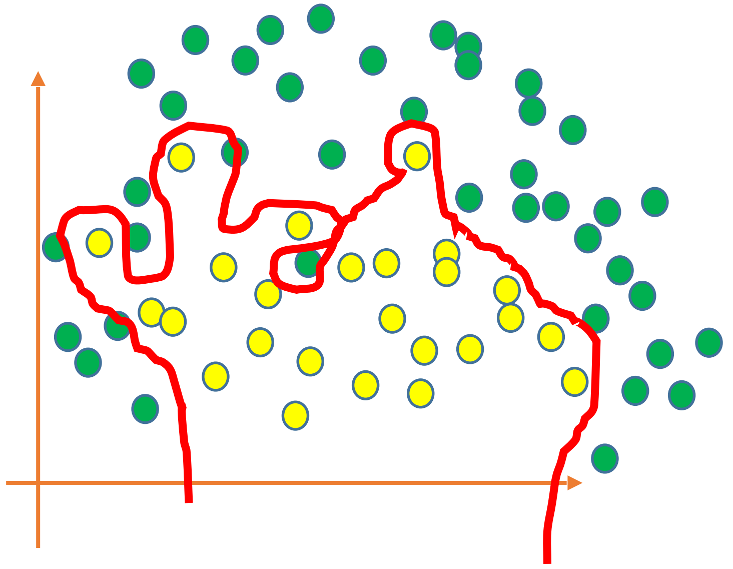
   https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf
   https://blog.coast.ai/lets-evolve-a-neural-network-with-a-genetic-algorithm-code-included-8809bece164

2. 粒子群优化（Particle Swarm Optimization, PSO）
   https://visualstudiomagazine.com/articles/2013/12/01/neural-network-training-using-particle-swarm-optimization.aspx
   https://ieeexplore.ieee.org/document/1202255/?reload=true

过拟合 （Overfitting)

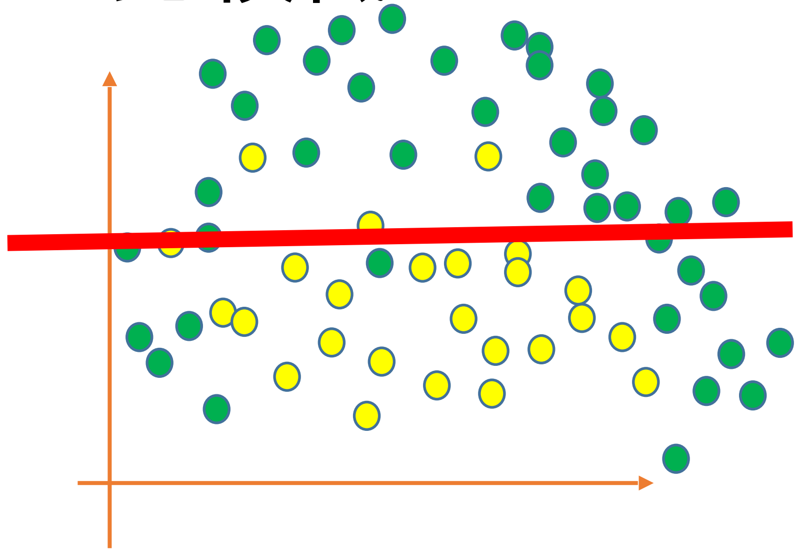

欠拟合

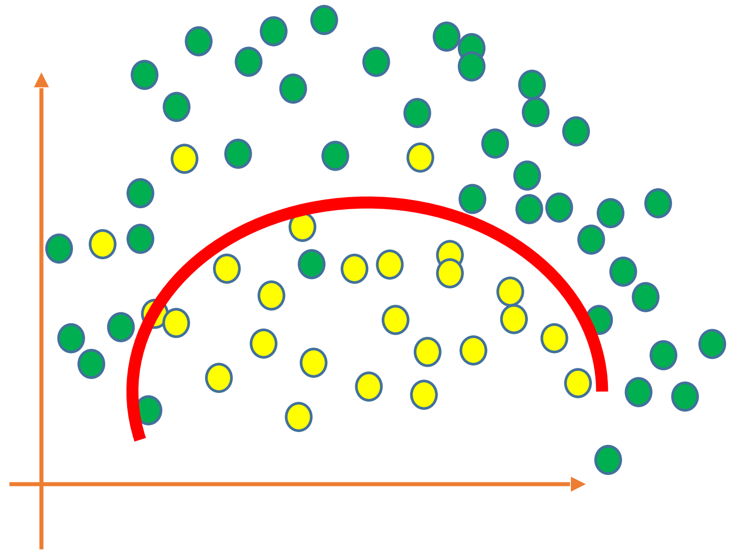

恰好

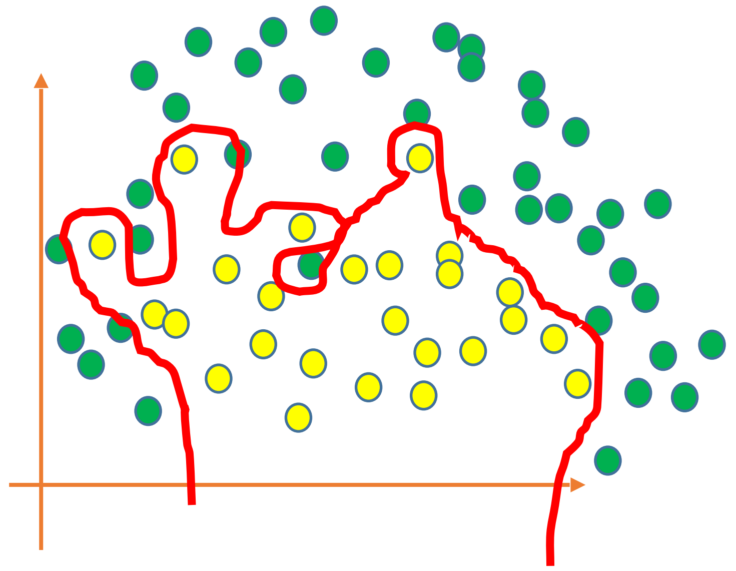

过拟合

过拟合 （Overfitting)

在训练数据集上的准确率很高，但是在测试集上的准确率比较低


欠拟合

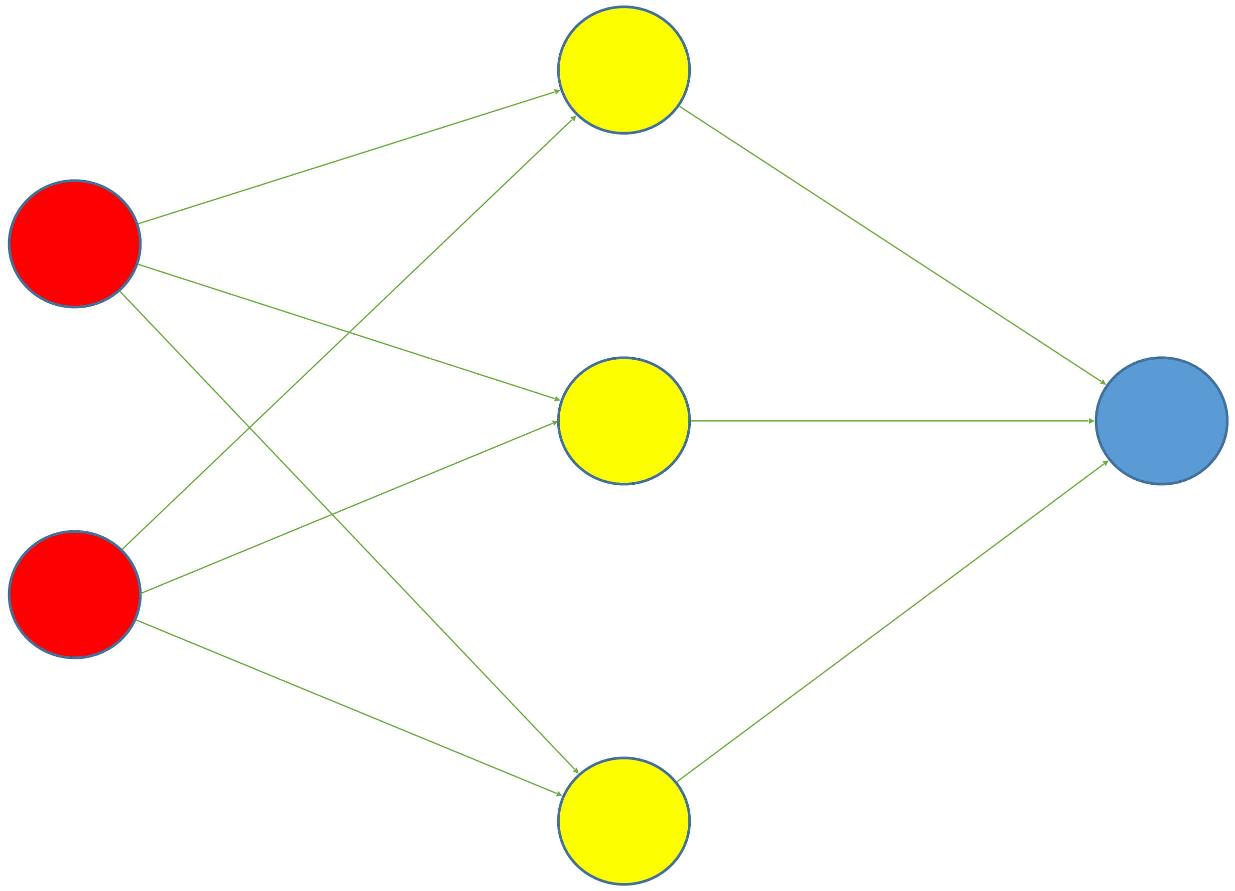

恰好


过拟合

过拟合的解决方案

- DropOut
- L2 正则化
- L1 正则化
- MaxNorm

过拟合　(Overfitting) 的网络

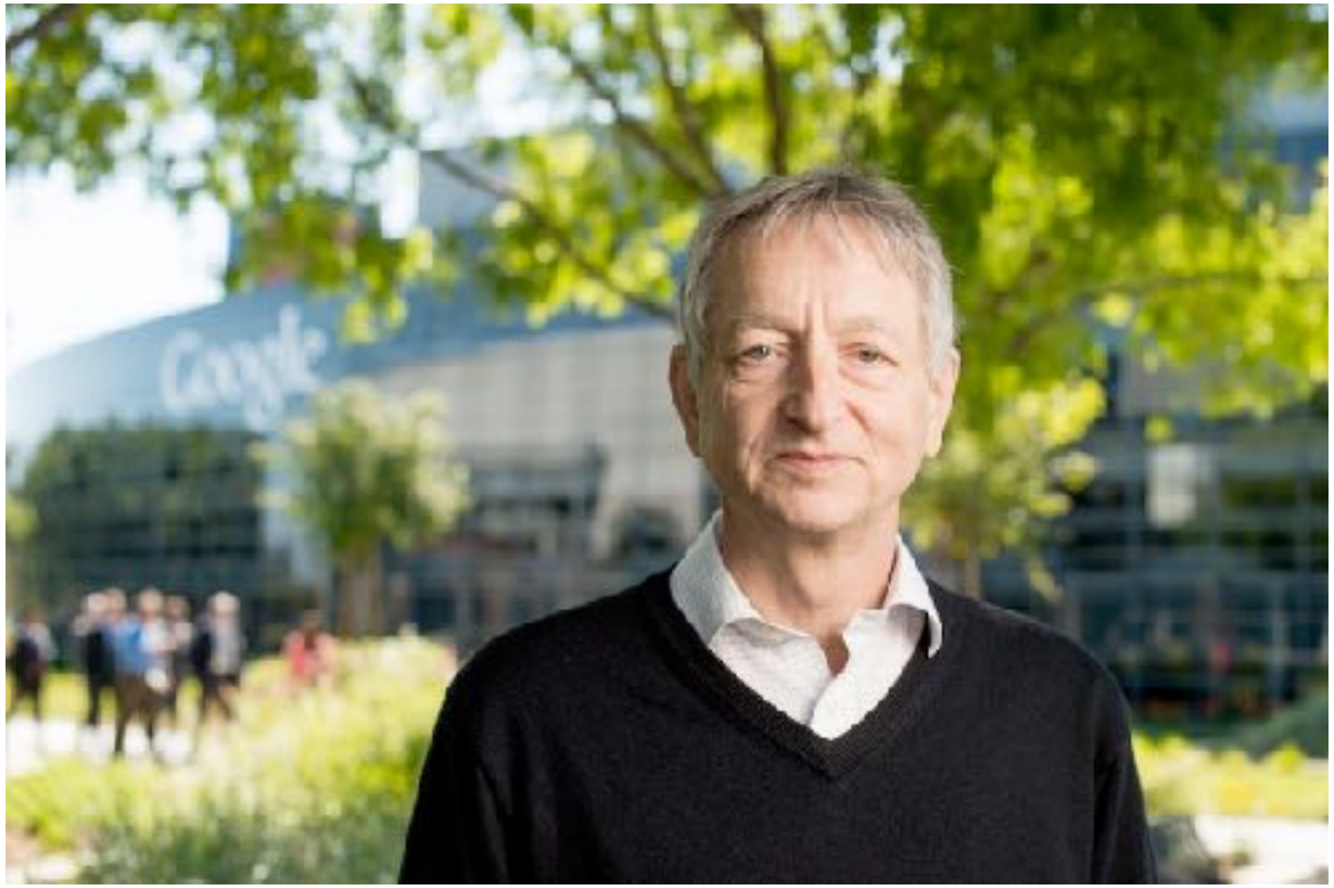


Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

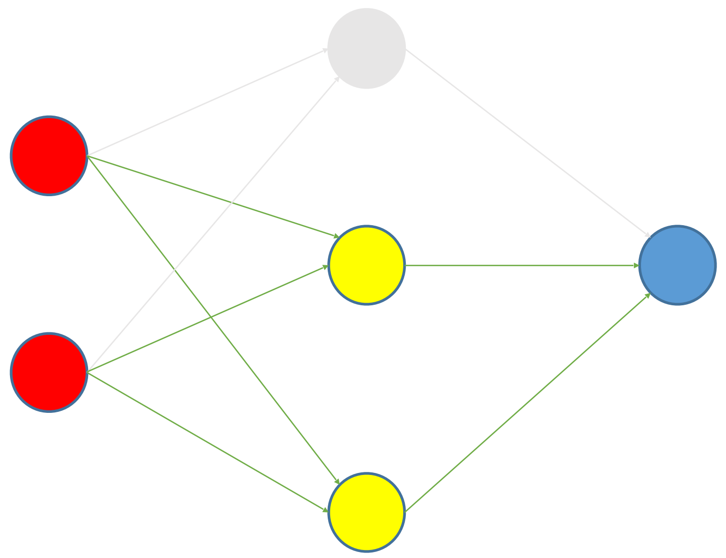
Dropout: A Simple Way to Prevent Neural Networks from Overfitting

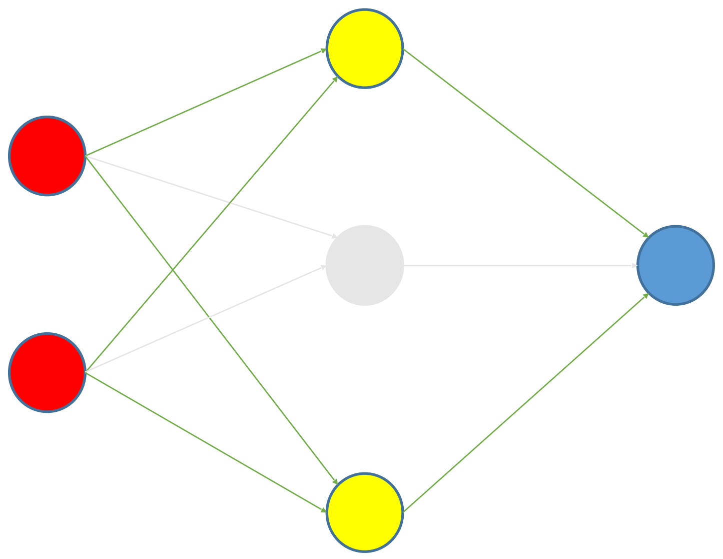
Nitish Srivastava
Geoffrey Hinton
Alex Krizhevsky
Ilya Sutskever
Ruslan Salakhutdinov
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

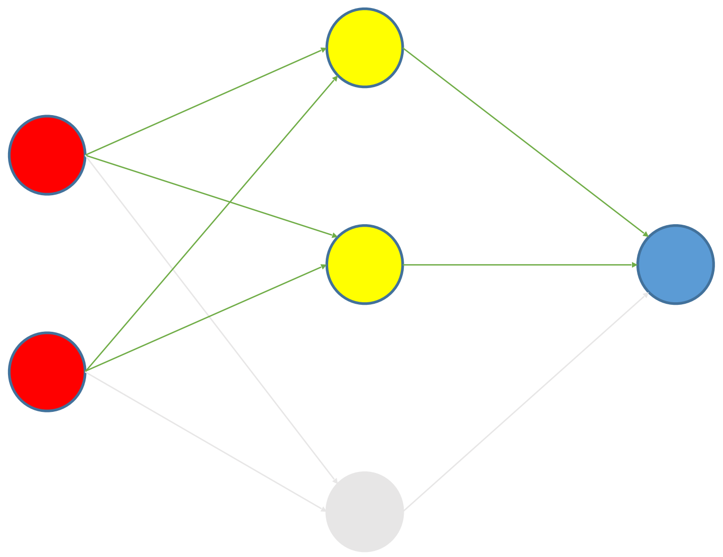
Editor: Yoshua Bengio



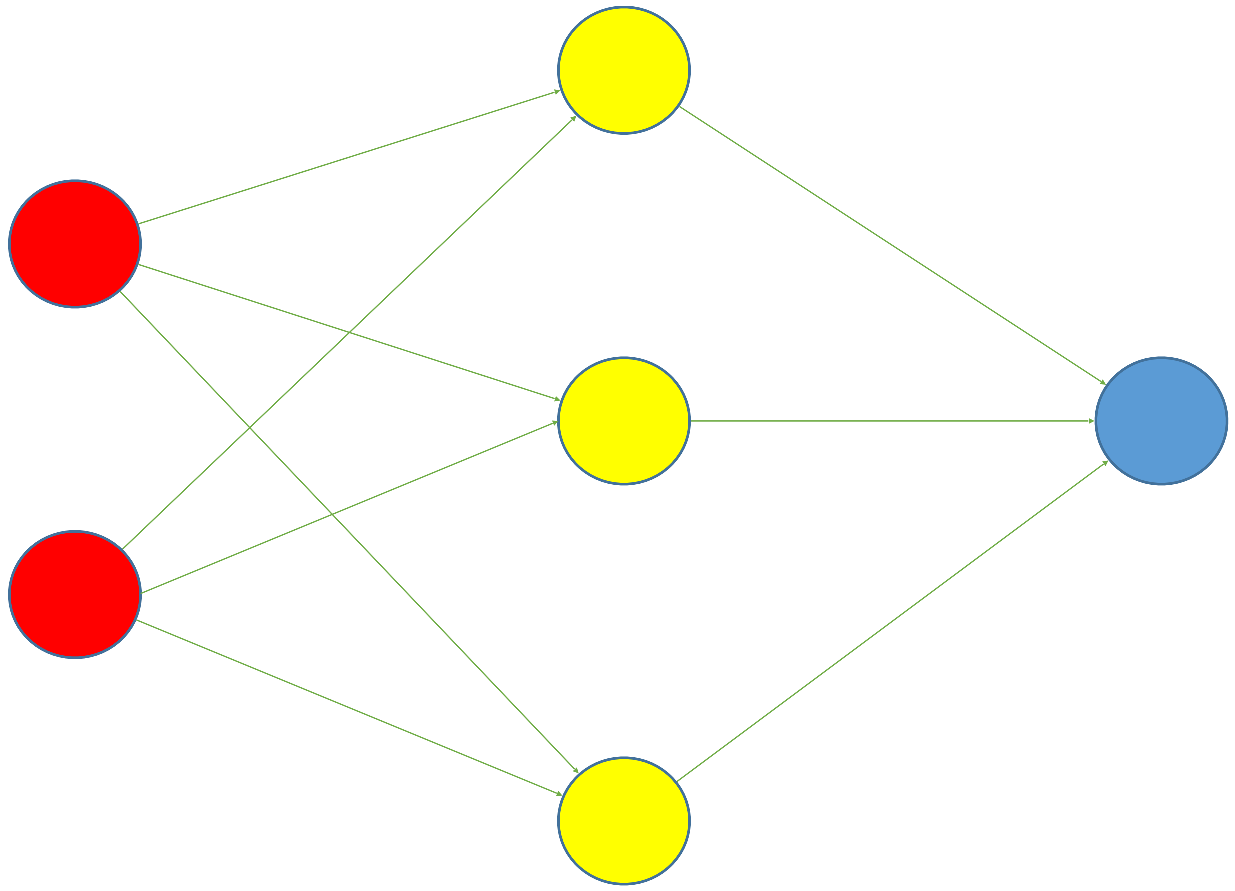
DropOut 训练 （DropOut rate 1/3)







DropOut 使用 （系数需要乘以1-DropOut rate）



L2 正则化

- 对损失函数（loss function）f(θ) 中的每一个系数θ_i, 都对损失函数加上$\frac{1}{2}\lambda\theta_i^2$， 其中$\lambda$是正则化的强度。
- 在训练的每一次更新系数的时候都额外加上这一步:
- $\theta_i = \theta_i - \lambda\theta_i$

- 正则化的目的是使系数的绝对值减小，对绝对值越大的系数，减小的程度越强。
- 正则化使得大多数系数的值都不为零，但是绝对值都比较小。

L1 正则化

- 对损失函数（loss function）f(θ) 中的每一个系数θ_i, 都对损失函数加上$\lambda|\theta_i|$，其中λ是正则化的强度。
- 在训练的每一次更新系数的时候都额外加上这一步：

$$\theta_i = \begin{cases} \theta_i - \lambda & \theta_i > 0 \\ \theta_i + \lambda & \theta_i \leq 0 \end{cases}$$

- L1正则化的目的是使得许多系数的绝对值接近0，其它那些系数不接近于0的系数对应的特征就是对输出有影响的特征。所以L1甚至可以用于作为特征选择的工具。

最大范数约束 (Max Norm)

- 对每一个神经元对应的系数向量，设置一个最大第二范数值 c， 这个值通常设为3。如果一个一神经元的第二范数值大于 c， 那么就将每一个系数值按比例缩小，使得第二范式值等于 c 。
- 在训练的每一次更新系数的时候都额外加上这一步：

$$\theta_i = \theta_i * \frac{c}{\|\theta\|}, \|\theta\| > c$$

- 由于最大范数的约束，可以防止由于训练步长较大引发的过拟合。

神经元系数的初始化

- bias（偏置系数）： 初始化为0
- 普通系数：初始化为 $\theta_i = np.random.randn(n) * sqrt(\frac{2.0}{n})$, 其中$n$为神经元的输入向量的元素个数