

# Some Intuition about Activation Functions in Feed-Forward Neural Networks

June 13, 2014

## **Abstract**

Everyone thought it was great to use differentiable, symmetric, non-linear activation functions in feed-forward neural networks, until Alex Krizhevsky [8] found that Rectifier Linear Units, despite being not entirely differentiable, nor symmetric, and most of all, piece-wise linear, were computationally cheaper and worth the trade-off with their more sophisticated counterparts. Here are just a few thoughts on why we might care about the shape of the sigmoid and hyperbolic tangent functions, what the implications of using ReLUs are, and possible ways of applying these insights for better learning strategies.

# 1 Models of Neurons

The rectified linear unit is the building block for current state-of-the-art implementations of deep convolutional neural networks. In order to bring out its specific characteristics, we shall first consider two other compatible neuron models: the binary threshold neuron, which is the most intuitive, and the logistic sigmoid neuron, which is the most analytically appealing.

## Binary Threshold Neuron

$$y = \begin{cases} 1 & \text{if } M \leq b + \sum_{i=1}^k x_i \cdot w_i, \text{ where } M \text{ is a threshold parameter} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

Intuitively,  $y$  takes a hard decision, just like biological neurons: either a charge is sent, or it isn't.  $y$  can be seen as producing spikes,  $x_i$  as the indicator value of some feature, and  $w_i$  as a parameter of the function that indicates how important  $x_i$  is in determining  $y$ . Although this model is closer than most to reality, the function is not differentiable, which makes it impossible to use greedy local optimisation learning algorithms - such as gradient descent - which need to compute derivatives involving the activation functions.

## Logistic Sigmoid Neuron

$$y = \frac{1}{1 + e^{-z}}, \text{ where } z = b + \sum_{i=1}^k x_i \cdot w_i \quad (2)$$

Like the binary threshold neuron, the output domain of this neuron is bounded by 0 and 1. But this time, the function is fully differentiable. Moreover, it is non-linear, which helps to increase performance [6].

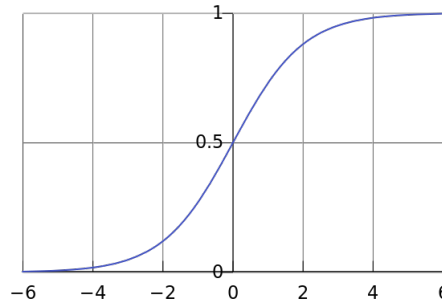


Figure 1: Single-Input Logistic Sigmoid Neuron

To see why, the graph plot below lends itself to the following intuition: consider the neuron as a feature detector, where the input  $x$  is the amount of evidence for a certain feature, and  $y$  is the probability that the feature is present.  $y$  is strictly increasing in  $x$ , because every extra amount of evidence for the feature increases the probability of its presence.  $y$  is bounded above by 1 and 0, because it is a probability. Interestingly, the slope of  $y$  is highest in absolute value around  $x = 0.5$ , and tends towards 0 as  $x$  departs from 0.5 in either direction. This is like saying: if I'm clueless as to whether the feature is present or not, any evidence in favour of presence or absence will push me easily in that direction; however, the more you want to convince me of it, the higher the required evidence gets.

A disadvantage of this neuron model is that it is computationally expensive to compute.

## Rectified Linear Neuron

$$y = \max\{0, b + \sum_{i=1}^k x_i \cdot w_i\} \quad (3)$$

As can be seen in the graph plot below, the rectified linear neuron is neither fully differentiable (not at 0), nor bounded above. Moreover, it only has two slopes, so its derivative with respect to  $x_i$  can only be one of two values: 0 or  $w_i$ . Although this may come as a strong downgrade in

sophistication compared to the logistic sigmoid neuron, it is so much more efficient to compute (both its value and its partial derivatives) that it enables much larger network implementations [8]. Until now, this has more than offset the per-neuron information loss - and saturation risks - of the rectifier versus the sigmoid unit [9].

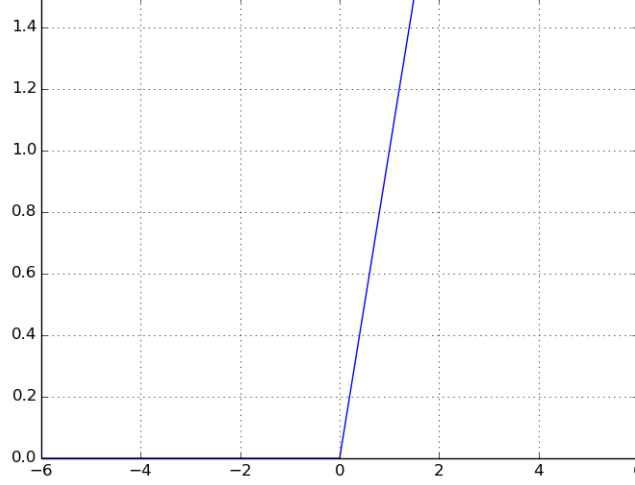


Figure 2: Single-Input Rectified Linear Neuron

### Softmax Neuron

$$y_j = \frac{e^{z_j}}{\sum_{i=1}^k e^{z_i}}, \text{ where } z_j = b_j + \sum_{i=1}^k x_i \cdot w_{i,j} \quad (4)$$

The equation of a softmax neuron needs to be understood in the context of a layer of  $k$  such neurons within a neural network: therefore, the notation  $y_j$  corresponds to the output of the  $j^{th}$  softmax neuron, and  $w_{i,j}$  corresponds to the weight of  $x_i$  as in input for the  $j^{th}$  softmax neuron. A layer of softmax neurons distinguishes itself from others in that neighbouring neurons interact with each other: as can be seen from the equation, the input vectors of all the softmax neurons  $z_1, z_2, \dots, z_k$  serve to enforce  $\sum_{i=1}^k y_i = 1$ . In other words, the vector  $(y_1, y_2, \dots, y_k)$  defines a probability mass function. This makes the softmax layer ideal for classification: neuron  $j$  can be made to represent the probability that the input is an instance of class  $j$ . Another attractive aspect of the softmax neuron is that its derivative is quick to compute: it is given by  $\frac{dy}{dz} = \frac{y}{1-y}$ .

## 2 Training: Backpropagation

Now that the architecture of a deep CNN has been explained, the question remains of how to train it. Mathematically: now that the function space has been explained, the question remains of how this space is searched. In the case of feed-forward neural networks and supervised learning, this is done with gradient descent, a local (therefore greedy) optimisation algorithm. Gradient descent relies on the partial derivatives of the error (a.k.a cost) function with respect to each parameter of the network; the backpropagation algorithm is an implementation of gradient descent which efficiently computes these values.

### 2.0.1 Compute Error-Weight Partial Derivatives

Let  $t$  be the target output (with classification, this is the label) and let  $y = (y_1, y_2, \dots, y_P)$  be actual value of the output layer on a training case. (Note that classification is assumed here: there are multiple output neurons, one for each class).

The error is given by

$$E = \mathcal{C}(\mathbf{t} - \mathbf{y}) \quad (5)$$

where  $\mathcal{C}$  is the chosen cost function. The error-weight partial derivatives are given by

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial net} \cdot \frac{\partial net}{\partial w_{ij}} \quad (6)$$

Since in general, a derivative  $\frac{\partial f}{\partial x}$  is numerically obtained by perturbing  $x$  and taking the change in  $f(x)$ , the advantage with this formula is that instead of individually perturbing each weight  $w_{ij}$ , only the unit outputs  $y_i$  are perturbed. In a neural network with  $k$  fully connected layers and  $n$  units per layer, this amounts to  $\Theta(k \cdot n)$  unit perturbations instead of  $\Theta(k \cdot n^2)$  weight perturbations (note that the bound on weight perturbations is no longer tight if we drop the assumption of fully connected layers).

**How Gradient Propagates** It is important for intuition to realise how the gradient propagates through a network - especially since a single piece of information - the error - is supposed to adjust up to millions of parameters, which could be considered as a bit of a stretch. Let us note that in the case of classification, the error is not a scalar, but a vector of size equal to the number of classes, so it carries more than "one piece" of information.

With the following notation:

- $y_j$ , the output of unit (a.k.a neuron)  $j$ , but also used to refer to the unit  $j$  itself
- $w_{ij}$ , the weight of the edge connecting lower-layer neuron  $y_i$  to upper-layer neuron  $y_j$
- $z_j := b + \langle x, w \rangle = b + \sum_{i=1}^k x_i \cdot w_{ij}$ , the input vector for  $y_j$  - therefore  $y_j = \psi(z_j)$

The rules for propagating the gradient backwards through a network are:

- to **initialise**:  $grad \leftarrow \mathcal{C}'(y_L)$ , where  $y_L$  is the output unit
- to **propagate through a unit**  $y_j$ :  $grad \leftarrow grad \cdot \psi'(z_j)$
- to **propagate through an edge**  $w_{ij}$ :  $grad \leftarrow grad \cdot w_{ij}$
- to **stop at an edge**  $w_{ij}$ :  $grad \leftarrow grad \cdot y_i$

Note that

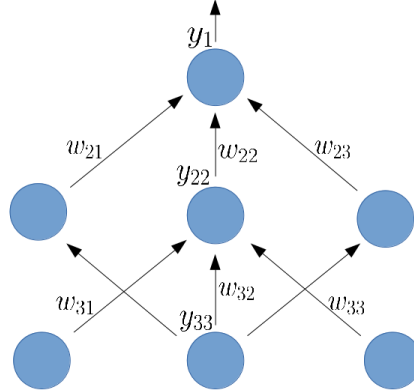


Figure 3: (CHANGE NOTATION to reflect one above with  $\mathbf{y}_L$  for top layer) Feed-Forward Neural Network with one Hidden Layer

So for example, given the figure above:

- for  $\frac{\partial E}{\partial w_{57}}$ : initialise, propagate through  $y_7$ , then stop at  $w_{57}$ :  $\mathcal{C}'(y_7) \cdot \psi'(z_7) \cdot y_5$
- for  $\frac{\partial E}{\partial w_{25}}$ : initialise, propagate through  $y_7$ ,  $w_{57}$ ,  $y_5$  then stop at  $w_{25}$ :  $\mathcal{C}'(y_7) \cdot \psi'(z_7) \cdot w_{57} \cdot \psi'(z_5) \cdot y_2$

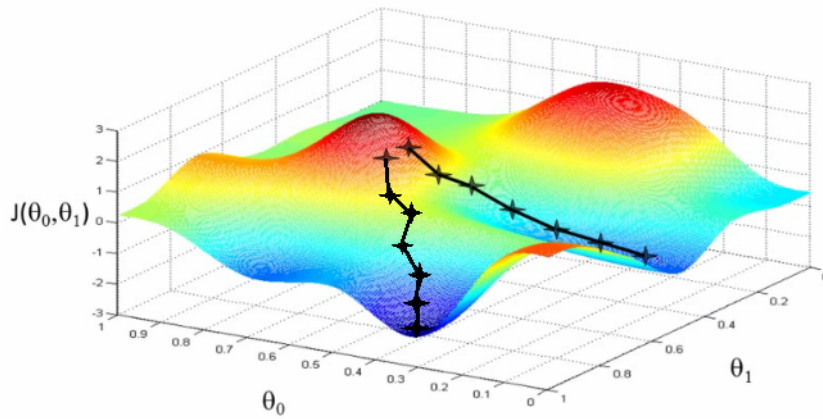


Figure 4: an error surface with poor local minima

## 2.0.2 Update Weight Values (with Stochastic Gradient Descent)

The learning rule is given by  $w_{i,t+1} = w_{i,t} + \tau \cdot \frac{\partial E}{\partial w_{i,t}}$

Visually, this means that weight values move in the direction they will reduce the error quickest, i.e. the direction of steepest descent on the error surface is taken. Notice that given the learning rule, gradient descent converges (i.e.  $w_{i,t+1}$  equals  $w_{i,t}$ ) when the partial derivative reaches zero. This corresponds to a local minimum on the error surface. In the figure below, two potential training sessions are illustrated. The minima attained in each case are not the same. This illustrates a strong shortcoming with backpropagation: parameter values can get stuck in poor local minima.

## References

- [1] Donahue, Jeff; Jia, Yangqing; Vinyals, Oriol; Hoffman, Judy; Zhang, Ning; Tzeng, Eric; Darrell, Trevor; *DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition* arXiv preprint arXiv:1310.1531, 2013
- [2] Zhou, Zhi-Hua; Zhang, Min-Ling; *Multi-Instance Multi-Label Learning with Application to Scene Classification* Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006
- [3] Pastor-Pellicer, Joan; Zamora-Martinez, Francisco; Espana-Boquera, Salvador; Castro-Bleda, Maria Jose; *F-Measure as the Error Function to Train Neural Networks*
- [4] Fusion Group - ControlPoint LLP, *Company Description* URL: <http://www.fusionprovida.com/companies/control-point>, last accessed 5th June 2014.
- [5] Barron, Andrew R., *Universal Approximation Bounds for Superpositions of a Sigmoidal Function* IEEE Transactions on Information Theory, Vol. 39, No. 3 May 1993
- [6] Bengio, Yoshua; *Learning Deep Architectures for AI* Foundations and Trends in Machine Learning, Vol. 2, No. 1 (2009) 1-127 2009
- [7] Russell, Stuart J; Norvig, Peter; *Artificial Intelligence: A Modern Approach* 2003
- [8] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E.; *ImageNet Classification with Deep Convolutional Neural Networks* 2012
- [9] Glorot, Xavier; Bordes, Antoine; Bengio, Yoshua; *Deep Sparse Rectifier Neural Networks* 2013

- [10] Hornik, Kur; Stinchcombe, Maxwell; White, Halber; *Multilayer Feed-Forward Networks are Universal Approximators*  
1989
- [11] Saenko, K., Kulis, B., Fritz, M., and Darrell, T.; *Adapting visual category models to new domains*  
ECCV, 2010
- [12] Bay, H., Tuytelaars, T., and Gool, L. Van; *SURF: Speeded up robust features*  
ECCV, 2006
- [13] Sermanet, Pierre; Eigen, David; Zhang, Xiang; Mathieu, Michael; Fergus, Rob; LeCun, Yann; *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*  
arXiv:1312.6229
- [14] Joan Pastor-Pellicer, Francisco Zamora-Martinez, Salvador Espaa-Boquera, Mara Jos Castro-Bleda; *F-Measure as the Error Function to Train Neural Networks*  
Advances in Computational Intelligence Volume 7902, 2013, pp 376-384
- [15] URL: <https://code.google.com/p/cuda-convnet/>, last accessed 6th June 2014.