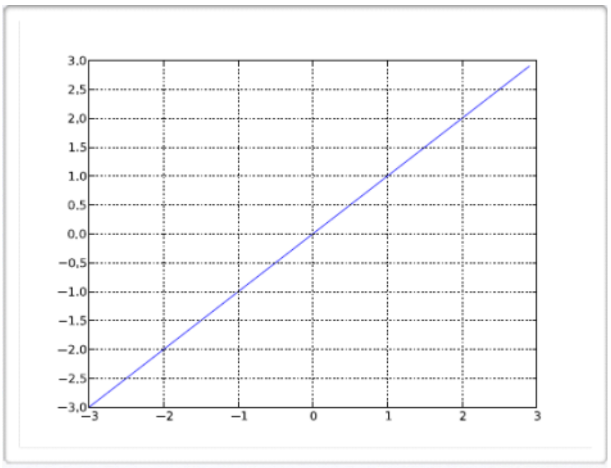
Neural Networks

Activation Functions:

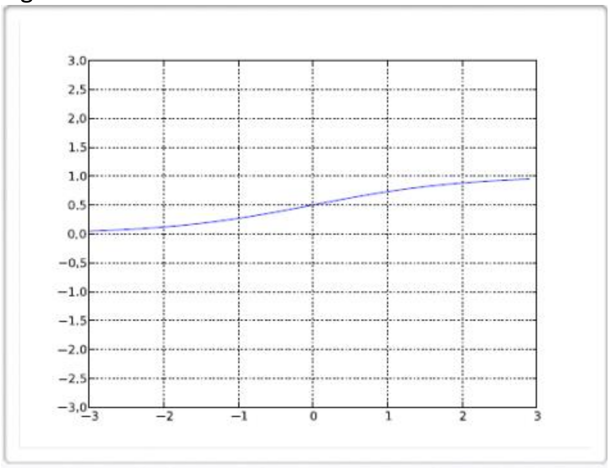
Linear



$$g(a) = a$$

Simple Activation Function.
Useful as the last layer activation for Regression.

Sigmoid

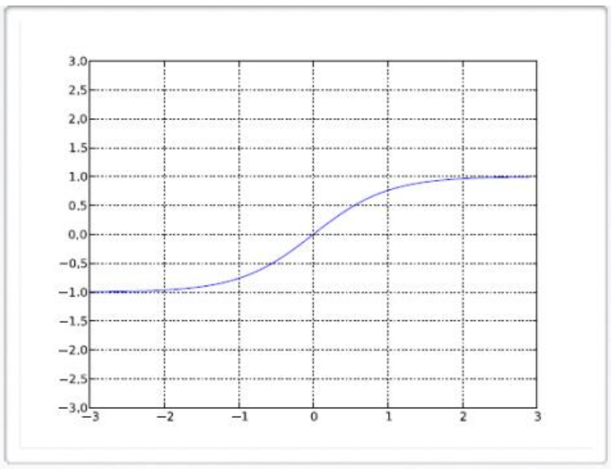


$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

Source: CS321n

- *Sigmoids saturate and kill gradients.* A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- *Sigmoid outputs are not zero-centered.* This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $\boldsymbol{x} > \boldsymbol{0}$ elementwise in $\boldsymbol{f} = \boldsymbol{w}^T\boldsymbol{x} + \boldsymbol{b}$)), then the gradient on the weights $\boldsymbol{w}$ will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression $\boldsymbol{f}$). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

TanH



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

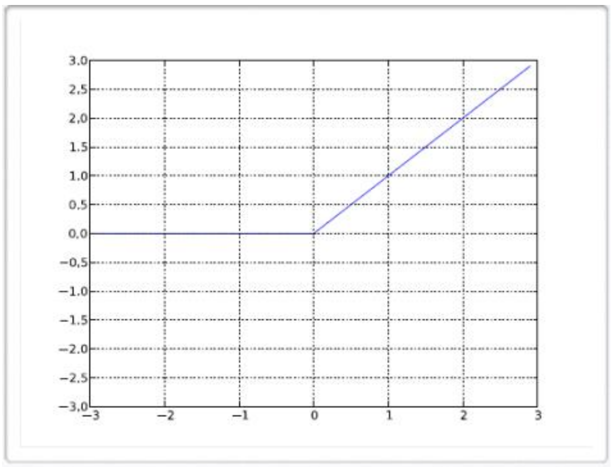
Source:CS231n

Alternate Form:

$\tanh(x) = 2\sigma(2x) - 1$.

Tanh. The tanh non-linearity is shown on the image above on the right. It squashes a real-valued number to the range [-1, 1]. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the *tanh non-linearity is always preferred to the sigmoid nonlinearity.* Also note that the tanh neuron is simply a scaled sigmoid neuron, in particular the following holds: $\tanh(x) = 2\sigma(2x) - 1$.

ReLU

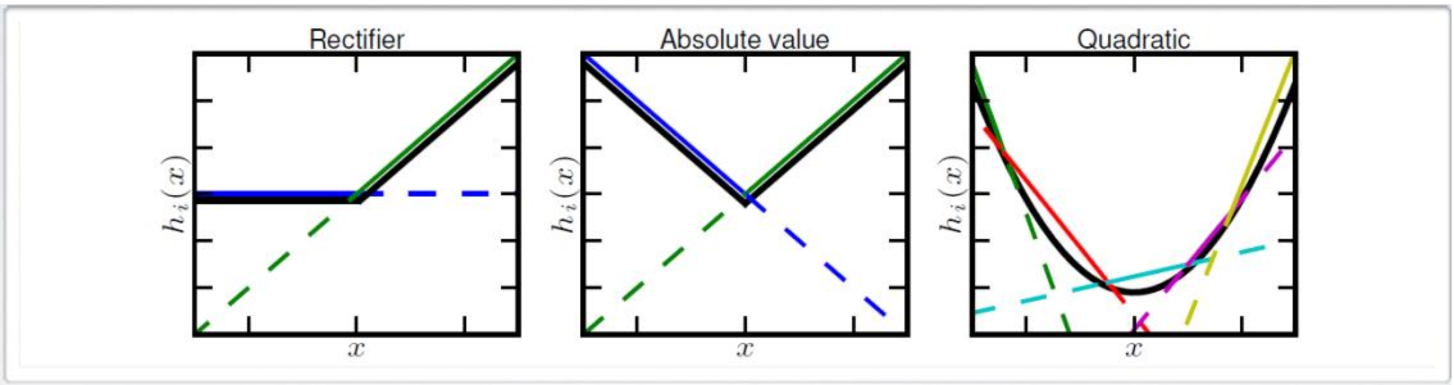


$$g(a) = \text{reclin}(a) = \max(0, a)$$

Source:CS231n

- (+) It was found to greatly accelerate (e.g. a factor of 6 in Krizhevsky et al.) the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- (+) Compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- (-) Unfortunately, ReLU units can be fragile during training and can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. That is, the ReLU units can irreversibly die during training since they can get knocked off the data manifold. For example, you may find that as much as 40% of your network can be "dead" (i.e. neurons that never activate across the entire training dataset) if the learning rate is set too high. With a proper setting of the learning rate this is less frequently an issue.

Maxout



Not lower / upper bounded

Does not give neurons with sparse activities

But gradients are sparse

$$g(x) = \max_{j \in [1,k]} a_j$$

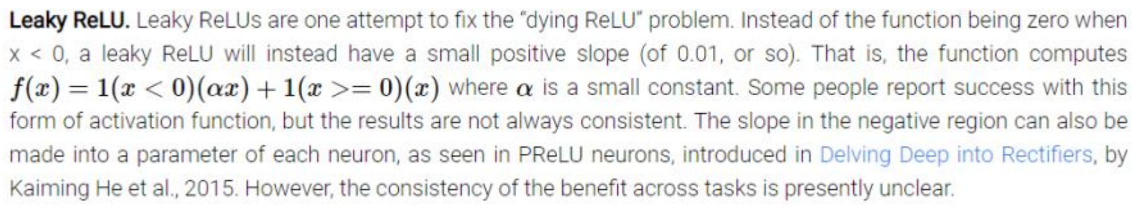
$$a_j = \boldsymbol{b}_j + \boldsymbol{w}_j^T \boldsymbol{x}$$

Source:CS231n

Maxout. Other types of units have been proposed that do not have the functional form $f(w^T x + b)$ where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (introduced recently by Goodfellow et al.) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function $\max(w_1^T x + b_1, w_2^T x + b_2)$. Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

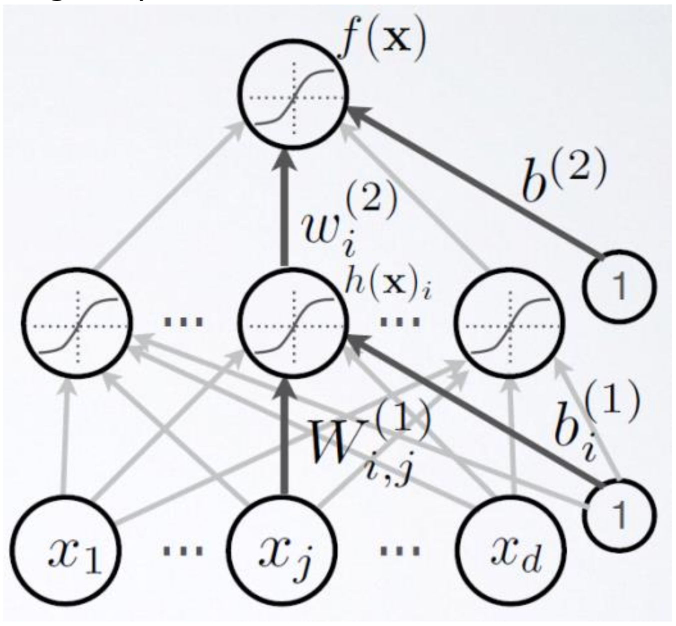
Leaky ReLU

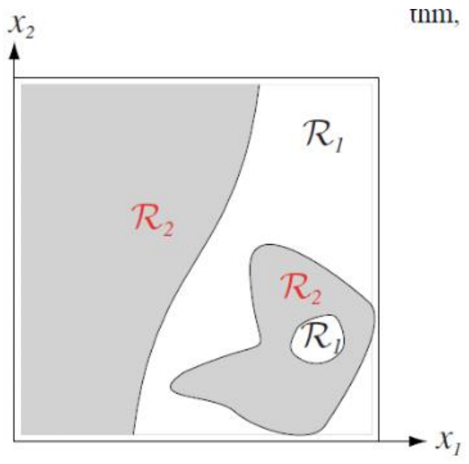
Source:CS231n



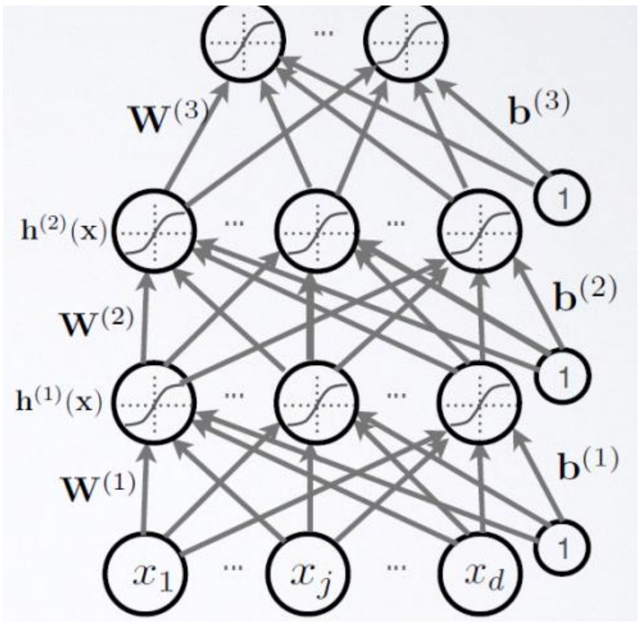
Leaky ReLU. Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small positive slope (of 0.01, or so). That is, the function computes $f(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x >= 0)(x)$ where α is a small constant. Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons, introduced in Delving Deep into Rectifiers, by Kaiming He et al., 2015. However, the consistency of the benefit across tasks is presently unclear.

Single Layer NN





Multi-Layer NN

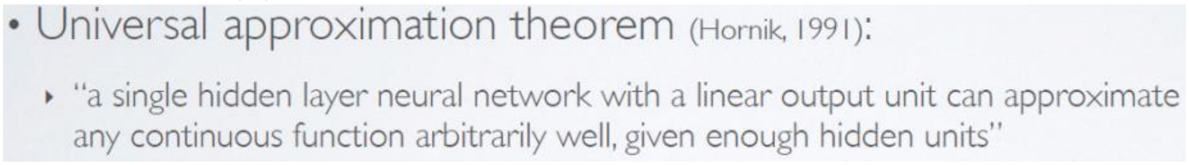


Question: For the hidden layer, we wanted to add nonlinearity to the model so we had to add it, but what I don't understand is the motivation behind stacking multiple layers. Why should we add multiple layers?

Answer: Single layer has a capacity limit(based on number of hidden units). So stacking multiple hidden number of layers help. Other method is the Universal Approximation Theorem for Single NN.

QUESTION : Is there a specific classification problem that has been proven that it can not to be addressed using a 1-layer NN ?
Answer: No it is possible as per the Universal Approx. thm,

Universal Approximation Thm.



- Universal approximation theorem (Hornik, 1991):
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"

Original Paper: https://cognitivemedium.com/magic_paper/assets/Hornik.pdf