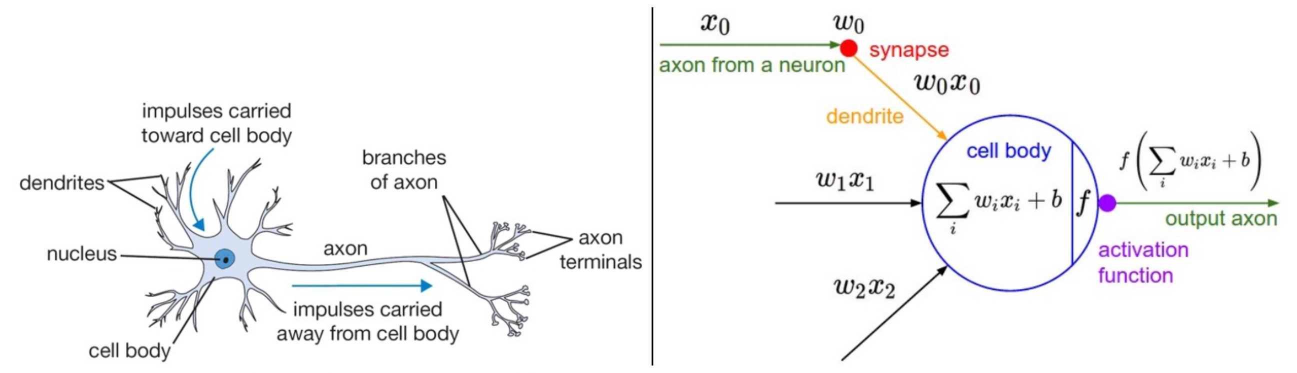
Activation Functions in Neural Networks

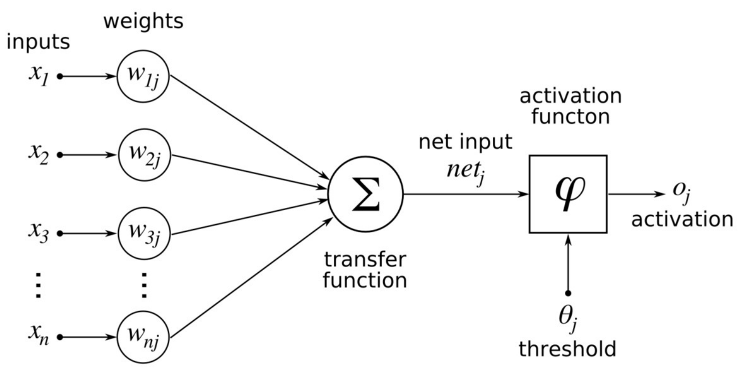
Mon, May 22, 2017 · 12 min read

By definition, activation function is a function used to transform the activation level of a unit (neuron) into an output signal. Typically, activation function has a "squashing" effect.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

An activation function serves as a threshold, alternatively called classification or a partition. Bengio et al. refers to this as "Space Folding". It essentially divides the original space into typically two partitions. Activation functions are usually introduced as requiring to be a non-linear function, that is, the role of activation function is made neural networks non-linear.



The purpose of an activation function in a Deep Learning context is to ensure that the representation in the input space is mapped to a different space in the output. In all cases a similarity function between the input and the weights are performed by a neural network. This can be an inner product, a correlation function or a convolution function. In all cases it is a measure of similarity between the learned weights and the input. This is then followed by a activation function that performs a threshold on the calculated similarity measure. In its most general sense, a neural network layer performs a projection that is followed by a selection. Both projection and selection are necessary for the dynamics learning. Without selection and only projection, a network will thus remain in the same space and be unable to create higher levels of abstraction between the layers. The projection operation may in fact be non-linear, but without the threshold function, there will be no mechanism to consolidate information. The selection operation is enforces information irreversibility, an necessary criteria for learning.

There have been many kinds of activation functions (over 640 different activation function proposals) that have been proposed over the years. However, best practice confines the use to only a limited kind of activation functions. Here I summarize several common-used activation functions, like Sigmoid, Tanh, ReLU, Softmax and so forth, as well as their merits and drawbacks.

Sigmoid Units

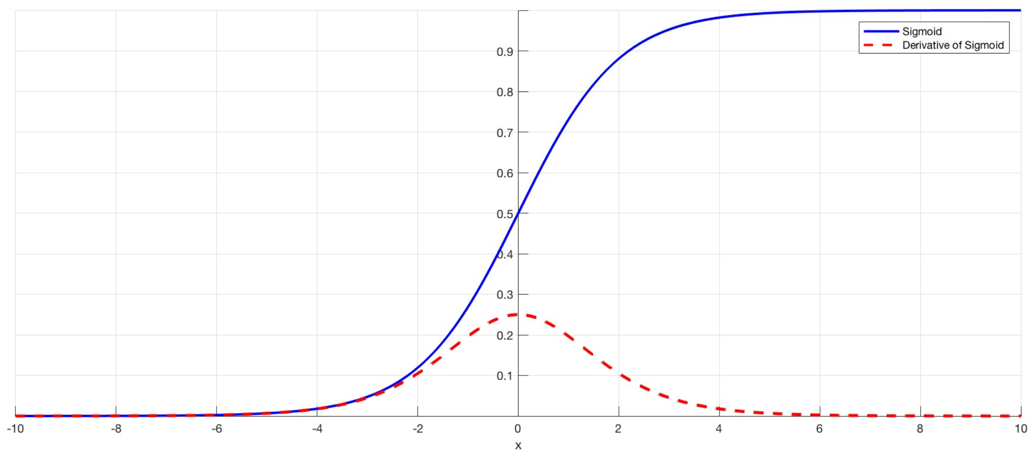
A Sigmoid function (https://en.wikipedia.org/wiki/Sigmoid_function) (used for hidden layer neuron output) is a special case of the logistic function (https://en.wikipedia.org/wiki/Logistic_function) having a characteristic "S"-shaped curve. The logistic function is defined by the formula

$$\sigma(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

where e is the natural logarithm (https://en.wikipedia.org/wiki/Natural_logarithm) base (also known as Euler's number), x_0 is the x-value of the Sigmoid's midpoint, L is the curve's maximum value, and k is the steepness of the curve. By setting $L = 1$, $k = 1$, $x_0 = 0$, we derive

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This is so called Sigmoid function and it is shown in the image below.



Sigmoid function takes a real-valued number and "squashes" it into range between 0 and 1, i.e., $\sigma(x) \in (0, 1)$. In particular, large negative numbers become 0 and large positive numbers become 1. Moreover, the sigmoid function has a nice interpretation as the firing rate of a neuron: from not firing at all (0) to fully-saturated firing at an assumed maximum frequency (1). But it has two major drawbacks:

1. Sigmoids saturate and kill gradients: The Sigmoid neuron has a property that when the neuron's activation saturates at either tail of 0 or 1, the gradient (where $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$, see the red dotted line above) at these regions is almost zero. During backpropagation, this 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. So, it is critically important to initialize the weights of sigmoid neurons to prevent saturation. For instance, if the initial weights are too large then most neurons would become saturated and the network will barely learn.

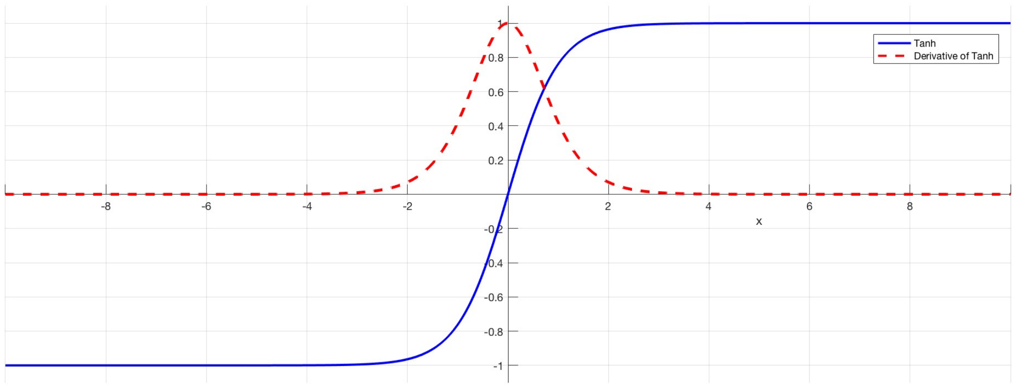
2. Sigmoid outputs are not zero-centered: It is undesirable since neurons in later layers of processing in a Neural Network would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f(x) = w^T x + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression 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.

Tanh Units

The hyperbolic tangent (tanh) function (used for hidden layer neuron output) is an alternative to Sigmoid function. It is defined by the formula

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

See the following image, tanh function is similar to Sigmoid function (Mathematically, $tanh(x) = 2\sigma(2x) - 1$), which is also sigmoidal ("S"-shaped). It squashes real-valued number to the range between -1 and 1, i.e., $tanh(x) \in (-1, 1)$.



Like the Sigmoid units, its activations saturate, but its output is zero-centered (means tanh solves the second drawback of Sigmoid). Therefore, in practice the tanh units is always preferred to the sigmoid units. The derivative of tanh function is defined as

$$tanh'(x) = 1 - tanh^2(x)$$

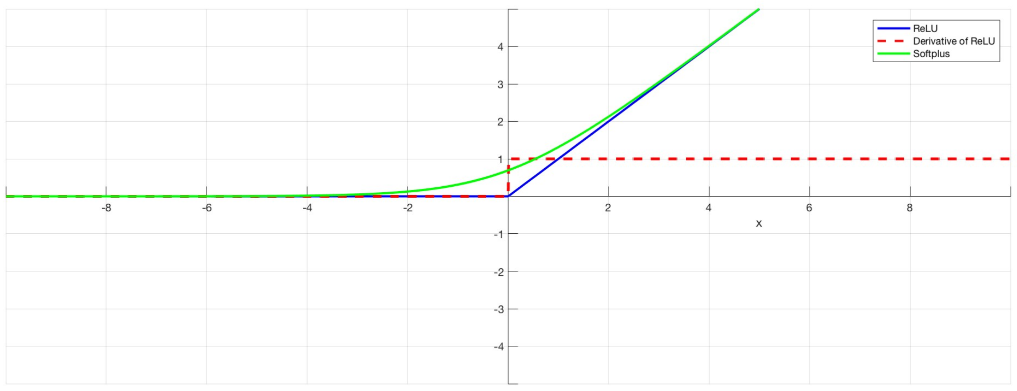
See the red dotted line in the above image, it interprets that tanh also saturate and kill gradient, since tanh's derivative has similar shape as compare to Sigmoid's derivative. What's more, tanh has stronger gradients, since data is centered around 0, the derivatives are higher, and tanh avoids bias in the gradients.

Rectified Linear Units (ReLU)

In the context of artificial neural networks, the ReLU (used for hidden layer neuron output) is defined as

$$f(x) = max(0, x)$$

where x is the input to a neuron. In other words, the activation is simply thresholded at zero. The range of ReLU is betweem 0 to ∞. See the image below (red dotted line is the derivative)



The ReLU function is more effectively than the widely used logistic sigmoid and its more practical counterpart, the hyperbolic tangent, since it efficaciously reduce the computation cost as well as some other merits:

1. It was found to greatly accelerate (Krizhevsky et al. (http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)) 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.
2. 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 also suffers several drawbacks, for instance, 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. 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.[2] Plus, here is a smooth approximation to the rectifier, which is called the softplus function (see the green line in the above image). It is defined as

$$f(x) = \ln(1 + e^x)$$

and its derivative is

$$f'(x) = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$

Interestingly, the derivative of Softplus is the logistic function. We can see that both the ReLU and Softplus are largely similar, except near 0 where the softplus is enticingly smooth and differentiable. But it is much easier and efficient to compute ReLU and its derivative than for the softplus function which has $log(\cdot)$ and $exp(\cdot)$ in its formulation. In deep learning, computing the activation function and its derivative is as frequent as addition and subtraction in arithmetic. By using ReLU, the forward and backward passes are much faster while retaining the non-linear nature of the activation function required for deep neural networks to be useful.

Leaky and Parametric ReLU

Leaky ReLU is 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 negative slope (of 0.01, or so). That is, the function computes

$$\begin{cases} f(x) & = 0.01x, \quad (x < 0) \\ f(x) & = x, \qquad (x \geq 0) \end{cases}$$

This form of activation function achieves some success, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, in this case, it is a Parametric ReLU (introduced in Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (http://arxiv.org/abs/1502.01852)), which take this idea further by making the coefficient of leakage into a parameter that is learned along with the other neural network parameters.

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

where α is a small constant (smaller than 1). However, the consistency of the benefit across tasks is presently unclear. Moreover, the formula of Parametric ReLU is equivalent to

$$f(x) = max(x, \alpha x)$$

which has a relation to "maxout" networks.

Randomized ReLU

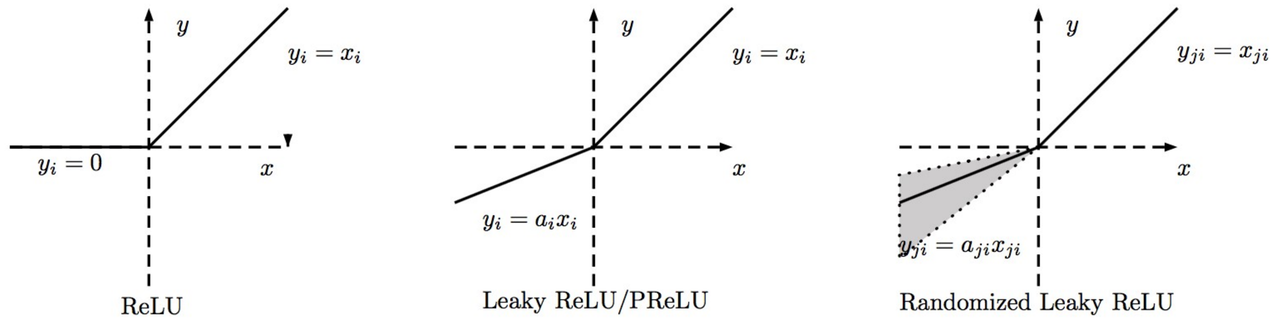
Randomized ReLU (RReLU) is a randomized version of Leaky ReLU, where the α is a random number. In RReLU, the slopes of negative parts are randomized in a given range in the training, and then fixed in the testing. It is reported that RReLU could reduce overfitting due to its randomized nature in the Kaggle (https://www.kaggle.com/) National Data Science Bowl (NDSB (https://www.kaggle.com/c/datasciencebowl)) competition. Mathematically, RReLU computes

$$\begin{cases} f(x_{ji}) & = x_{ji}, & \text{if } x_{ji} \geq 0 \\ f(x_{ji}) & = \alpha_{ji} x_{ji}, & \text{if } x_{ji} < 0 \end{cases}$$

where x_{ji} denotes the input of ith channel in jth example, $f(x_{ji})$ denotes the corresponding output after passing the activation function, $\alpha_{ji} \sim U(l, u), l < u$ and $l, u \in [0, 1)$. The highlight of RReLU is that in training process, α_{ji} is a random number sampled from a uniform distribution $U(l, u)$, while in the test phase, we take average of all the α_{ji} in training as in the method of dropout (https://www.cs.toronto.edu/-hinton /absps/JMLRdropout.pdf), and thus set α_{ji} to $\frac{l+u}{2}$ to get a deterministic result (In NDSB, α_{ji} is sampled from $U(3, 8)$). So in the test time, we have

$$f(x_{ji}) = \frac{x_{ji}}{\frac{l+u}{2}}$$

Here gives the comparing graph of different ReLUs



For Parametric ReLU, α_i is learned and for Leaky ReLU α_i is fixed. For RReLU, α_{ji} is a random variable keeps sampling in a given range, and remains fixed in testing. Generally, we summarize the advantages and potential problems of ReLUs:

- (+) Biological plausibility: One-sided, compared to the antisymmetry of tanh.
- (+) Sparse activation: For example, in a randomly initialized network, only about 50% of hidden units are activated (having a non-zero output).
- (+) Efficient gradient propagation: No vanishing or exploding gradient problems.
- (+) Efficient computation: Only comparison, addition and multiplication.
- (+) Scale-invariant: $max(0, \alpha x) = \alpha \cdot max(0, x)$.
- (-) Non-differentiable at zero: however it is differentiable anywhere else, including points arbitrarily close to (but not equal to) zero.
- (-) Non-zero centered.
- (-) Unbounded: Could potentially blow up.
- (-) Dying Relu problem: Relu neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. In this state, no gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state and "dies." In some cases, large numbers of neurons in a network can become stuck in dead states, effectively decreasing the model capacity. This problem typically arises when the learning rate is set too high.

Maxout

Some other types of units that do not have the functional form $f(w^Tx + b)$ where a non-linearity is applied on the dot product between the weights and the data. One relatively popular choice is the Maxout neuron (http://www-etud.iro.umontreal.ca/~goodfeli/maxout.html) that generalizes the ReLU and its leaky version. The Maxout neuron computes the function

$$max(w_1^T + b_1, w_2^T + b_2)$$

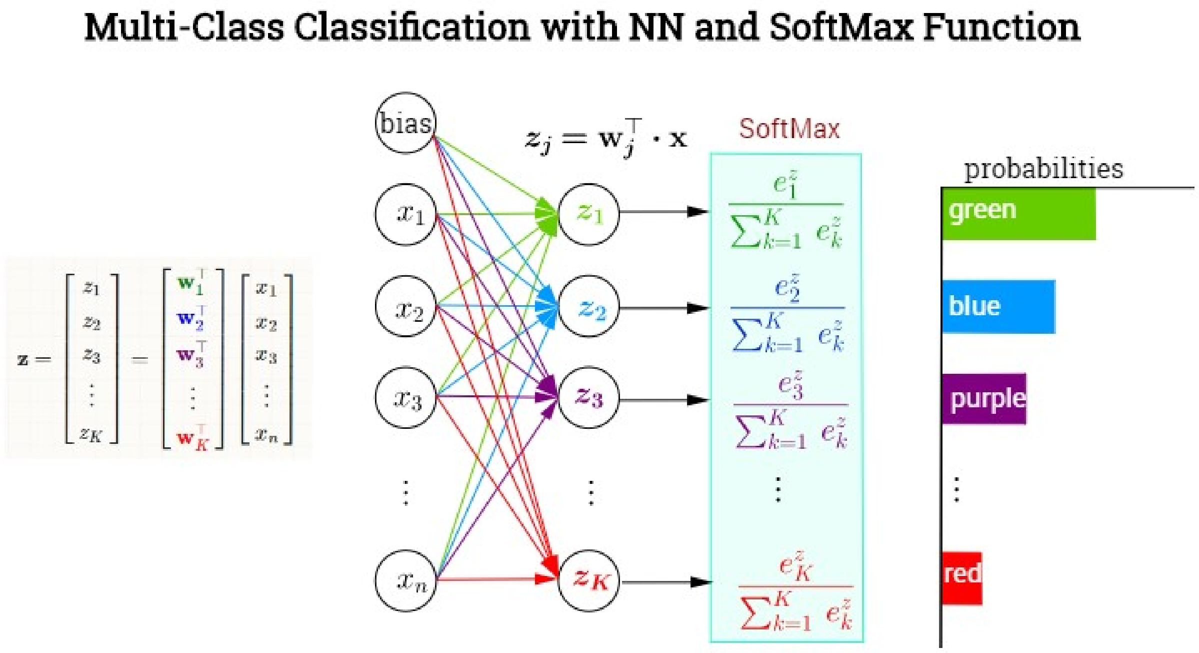
Notice that both ReLU and Leaky ReLU are a special case of this form (for example, for ReLU we have $w_1, b_1 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and *does not have its drawbacks (dying ReLU).* However, unlike the ReLU neurons it *doubles* the number of *parameters* for every single neuron, leading to a high total number of parameters.

Softmax

The Softmax function (Used for multi-classification neural network output), or normalized exponential function, in mathematics, is a generalization of the logistic function that “squashes” a K-dimensional vector $\mathbf{z}$ from arbitrary real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values in the range $[0, 1]$ that add up to 1. The function is given by

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, j = 1, 2, \ldots, K$$

In probability theory, the output of the Softmax function can be used to represent a categorical distribution, that is, a probability distribution over K different possible outcomes. In fact, it is the gradient-log-normalizer of the categorical probability distribution. Here is an example of Softmax application


Multi-Class Classification with NN and SoftMax Function
$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \\ \vdots \\ \mathbf{w}_K^T \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$
bias
x_1
x_2
x_3
x_n
$z_j = \mathbf{w}_j^T \cdot \mathbf{x}$
z_1
z_2
z_3
z_K
SoftMax
$\frac{e_1^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_2^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_3^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_K^z}{\sum_{k=1}^K e_k^z}$
probabilities
green
blue
purple
red

The softmax function is used in various multiclass classification methods, such as multinomial logistic regression, multiclass linear discriminant analysis, naive Bayes classifiers, and artificial neural networks. Specifically, in multinomial logistic regression and linear discriminant analysis, the input to the function is the result of K distinct linear functions, and the predicted probability for the jth class given a sample vector $\mathbf{x}$ and a weighting vector $\mathbf{w}$ is

$$P(y = j|\mathbf{x}) = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}}$$

This can be seen as the composition of K linear functions $\mathbf{x} \mapsto x^T w_1, \ldots, \mathbf{x} \mapsto x^T w_K$ and the softmax function (where $x^T w$ denotes the inner product of $\mathbf{x}$ and $\mathbf{w}$). The operation is equivalent to applying a linear operator defined by $\mathbf{w}$ to vectors $\mathbf{x}$, thus transforming the original, probably highly-dimensional, input to vectors in a K-dimensional space R^K. More details, see the link: Softmax Function (https://en.wikipedia.org/wiki/Softmax_function).

Other Activation Functions

Like, Identity (https://en.wikipedia.org/wiki/Identity_function), Binary Step (https://en.wikipedia.org/wiki/Heaviside_step_function), ArcTan (https://en.wikipedia.org/wiki/Inverse_trigonometric_functions), SoftSign, Exponential linear unit (ELU), S-shaped rectified linear activation unit (SReLU), Adaptive piecewise linear (APL), Bent identity, SoftExponential, Sinusoid (https://en.wikipedia.org/wiki/Sine_wave), Sinc (https://en.wikipedia.org/wiki/Sinc_function), Gaussian (https://en.wikipedia.org/wiki/Gaussian_function) and so forth. See the Wikipedia link: Activation Function (https://en.wikipedia.org/wiki/Activation_function).

Choose Activation Functions

Finally, to choose an activation function, we can follow a simple rule that: "Use the ReLU non-linearity, be careful with your learning rates and possibly monitor the fraction of 'dead' units in a network. If this concerns you, give Leaky ReLU or Maxout a try. Never use sigmoid. Try tanh, but expect it to work worse than ReLU/Maxout."

Reference

1. What is the role of the activation function in a neural network? – Quora (https://www.quora.com/What-is-the-role-of-the-activation-function-in-a-neural-network)
2. CS231n Convolutional Neural Networks for Visual Recognition (http://cs231n.github.io/neural-networks-1/)
3. Derivation: Derivatives for Common Neural Network Activation Functions (https://theclevermachine.wordpress.com/tag/tanh-function/)
4. Why use activation functions? (http://www.faqs.org/faqs/ai-faq/neural-nets/part2/section-10.html)
5. What are the benefits of using ReLU over softplus as activation functions? – Quora (https://www.quora.com/What-are-the-benefits-of-using-ReLU-over-softplus-as-activation-functions)
6. Empirical Evaluation of Rectified Activations in Convolution Network (https://arxiv.org/pdf/1505.00853.pdf)
7. Rectifier (neural networks) (https://en.wikipedia.org/wiki/Rectifier_%28neural_networks%29)
8. Maxout Networks (https://arxiv.org/pdf/1302.4389.pdf)
9. Softmax Function (https://en.wikipedia.org/wiki/Softmax_function)

deep-learning (/tags/deep-learning) machine-learning (/tags/machine-learning)

Related

- Machine Learning Note (4): Ensemble Learning (/post/ml_zzh_note_4/)
- Word2Vecf -- Dependency-Based Word Embeddings and Lexical Substitute (/post

/word2vecf/)

- Machine Learning Note (3): Support Vector Machine (/post/ml_zzh_note_3/)
- Word2Vec -- Mathematical Principles and Java Implementation (/post/word2vec/)
- Machine Learning Note (2): Linear Regression (/post/ml_zzh_note_2/)