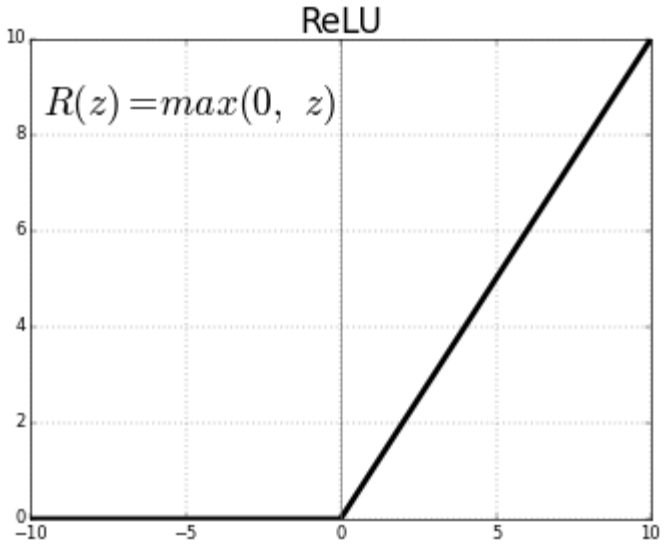
ReLU : Not a Differentiable Function: Why used in Gradient Based Optimization? and Other Generalizations of ReLU.

Kanchan Sarkar Follow

May 31, 2018 · 6 min read

ReLU (rectified linear unit) is one of the most popular function which is used as hidden layer activation function in deep neural network. ReLU activation function is defined as

g (z) = max { 0 , z }



ReLU Activation Function

The ReLU activation function **g(z) = max{0, z} is not differentiable at z = 0**. A function is differentiable at a particular point if there exist left derivatives and right derivatives and both the derivatives are equal at that point. ReLU is differentiable at all the point except 0. the left derivative at z = 0 is 0 and the right derivative is 1.

This may seem like g is not eligible for use in gradient based optimization algorithm. But in practice, gradient descent still performs

well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function. Hence it is acceptable for the minima of the cost function to correspond to points with undefined gradient. Hidden units that are not differentiable are usually non-differentiable at only a small number of points.

On the other hand, software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is not defined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0 . Instead, it was likely to be some small value that was rounded to 0 . In some contexts, more theoretically pleasing justifications are available, but these usually do not apply to neural network training. The important point is that in practice one can safely disregard the non-differentiability of the hidden unit activation functions.

Generalizations of Rectified Unit

1. Rectified linear units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active.

2. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.

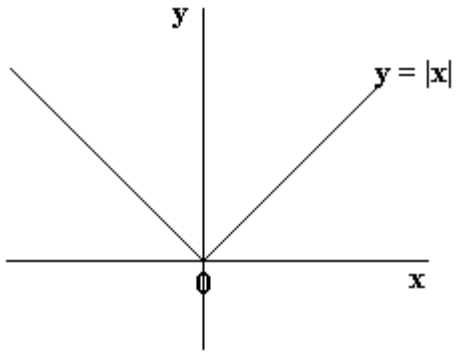
3. It can be a good practice to set all elements of b to a small, positive value, such as 0.1. This makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

4. One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is

zero. A variety of generalizations of rectified linear units guarantee that they receive gradient everywhere.

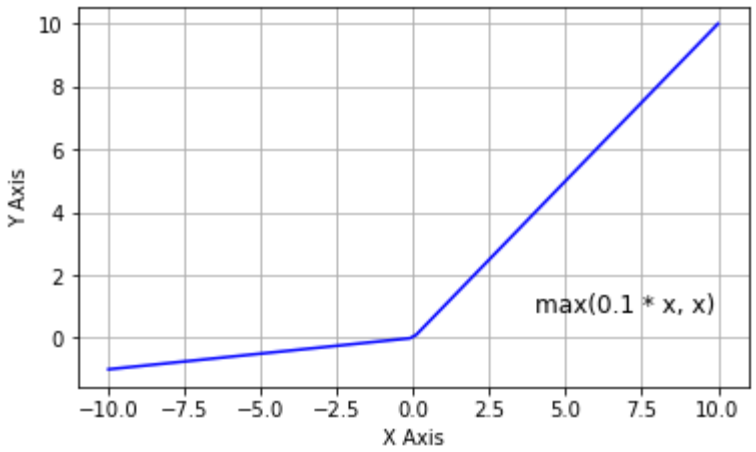
5. Several generalizations of rectified linear units exist. Most of these generalizations perform comparably to rectified linear units and occasionally perform better.

6. Three generalizations of rectified linear units are based on using a non-zero slope α_i when $\boldsymbol{z_i < 0}$: $\boldsymbol{h_i = g(z, \alpha)_i = \max(0, z_i) + \alpha_i \min(0, z_i)}$. **Absolute value rectification** fixes $\boldsymbol{\alpha_i = -1}$ **to obtain** $\boldsymbol{g(z) = |z|}$.



Absolute value rectification

It is used for object recognition from images, where it makes sense to seek features that are invariant under a polarity reversal of the input illumination. Other generalizations of rectified linear units are more broadly applicable. **leaky ReLU** uses $\boldsymbol{\alpha_i}$ to a small value like 0.01 while a **parametric ReLU or PReLU** treats α_i as a learnable parameter.



Leaky ReLU

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

Maxout units generalize rectified linear units further. Instead of applying an element-wise function g(z), maxout units divide z into groups of k values. Each maxout unit then outputs the maximum element of one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

where ***G(i)*** is the indices of the inputs for group **i, { (i − 1)k + 1, . . . , ik}**. This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.

A **maxout unit** can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as learning the activation function itself rather than just the relationship between units. With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, a maxout layer with two pieces can learn to implement the same function of the input x as a traditional layer using the rectified linear activation function, absolute value rectification function, or the leaky or parametric ReLU, or can learn to implement a totally different function altogether. The maxout layer will of course be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases

where maxout learns to implement the same function of x as one of the other layer types.

Each **maxout unit** is now parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low.

Maxout units have a few other benefits. In some cases, one can gain some sta-
tistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Rectified linear units and all of these generalizations of them are based on the
principle that models are easier to optimize if their behavior is closer to linear.
This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network architectures, the LSTM, propagates information through time via summation—a
particular straightforward kind of such linear activation.