Leaky ReLU: improving traditional ReLU

<https://www.machinecurve.com/index.php/2019/10/15/leaky-relu-improving-traditional-relu/>

The **Leaky [ReLU](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/)** is a type of [activation function](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/) which comes across many machine learning blogs every now and then. It is suggested that it is an improvement of traditional ReLU and that it should be used more often.

[Rectified Linear Unit](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/), or ReLU, is one of the most common [activation functions](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/) used in neural networks today. It is added to layers in neural networks to add nonlinearity, which is required to handle today’s ever more complex and nonlinear datasets.

Each neuron computes a [dot product and adds a bias value](https://www.machinecurve.com/index.php/2019/07/23/linking-maths-and-intuition-rosenblatts-perceptron-in-python/) before the value is output to the neurons in the subsequent layer. These mathematical operations are linear in nature. This is not bad if we were training the model against a dataset that is linearly separable (in the case of classification) or where a line needs to be estimated (when regressing).

However, if data is nonlinear, we face problems. Linear neuron outputs ensure that the system as a whole, thus the entire neural network, behaves linearly. By consequence, it cannot handle such data, which is very common today: the MNIST dataset, which we used for showing how to build [classifiers in Keras](https://www.machinecurve.com/index.php/2019/09/17/how-to-create-a-cnn-classifier-with-keras/), is nonlinear – and it is one of the simpler ones!

[Activation functions](https://www.machinecurve.com/index.php/2019/09/09/implementing-relu-sigmoid-and-tanh-in-keras/) come to the rescue by adding nonlinearity. They’re placed directly after the neural outputs and do nothing else but converting some input to some output. Because the mathematical functions used are nonlinear, the output is nonlinear – which is exactly what we want, since now the system behaves nonlinearly and nonlinear data is supported!

Note that although activation functions are pretty much nonlinear all the time, it’s of course also possible to use the identity function 𝑓(𝑥)=𝑥 as an [activation function](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/). It would be pointless, but it can be done.

It’s grown very popular and may be the most popular activation used today – it is more popular than the older [Sigmoid and Tanh](https://www.machinecurve.com/index.php/2019/09/04/relu-sigmoid-and-tanh-todays-most-used-activation-functions/) activation functions – for the reason that it can be computed relatively inexpensively. Computing ReLU is equal to computing 𝑅𝑒𝐿𝑈(𝑥)=𝑚𝑎𝑥(0,𝑥), which is much less expensive than the exponents or trigonometric operations necessary otherwise.

## **Problems with ReLU**

Firstly, ReLU is not continuously differentiable. At 𝑥=0, the breaking point between 𝑥and 0, the gradient cannot be computed. This is not too problematic, but can very lightly impact training performance.

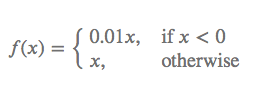
Secondly, and more gravely, ReLU sets all values < 0 to zero. This is beneficial in terms of sparsity, as the network will adapt to ensure that the most important neurons have values of > 0. However, this is a problem as well, since the gradient of 0 is 0 and hence neurons arriving at large negative values cannot recover from being stuck at 0. The neuron effectively dies and hence the problem is known as the *dying ReLU problem*. You’re especially vulnerable to it when your neurons are not initialized properly or when your data is not normalized very well, causing significant weight swings during the first phases of optimizing your model. The impact of this problem may be that your network essentially stops learning and underperforms.

## **Introducing Leaky ReLU**

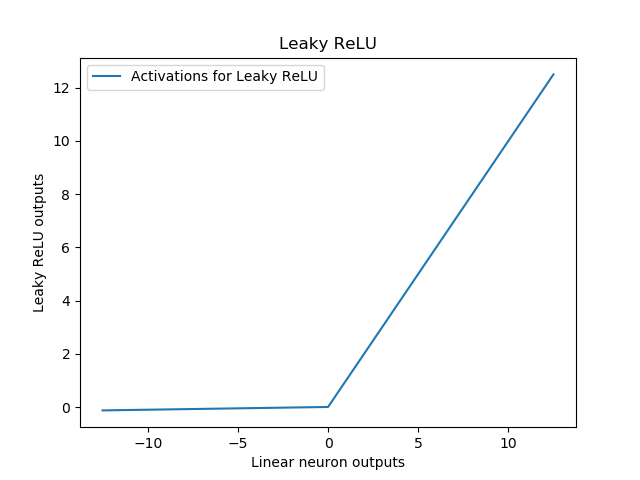
What if you *caused a slight but significant information leak* in the left part of ReLU, i.e. the part where the output is always 0?

This is the premise behind **Leaky ReLU**, one of the possible newer activation functions that attempts to minimize one’s sensitivity to the *dying ReLU problem*.

Mathematically, it is defined as follows (Maas et al., 2013):



Leaky ReLU can be visualized as follows:



If you compare this with the image for traditional ReLU above, you’ll see that for all 𝑖𝑛𝑝𝑢𝑡𝑠<0, the outputs are slightly descending. The thesis is that these small numbers reduce the death of ReLU activated neurons. This way, you’ll have to worry less about the initialization of your neural network and the normalization of your data. Although these topics remain important, they are slightly less critical.

In a 2018 study, Pedamonti argues that Leaky ReLU and ReLU performance on the MNIST dataset is similar. Even though the problem of dying neural networks may now be solved theoretically, it can be the case that it simply doesn’t happen very often – and that in those cases, normal ReLU works as well. “It’s simple, it’s fast, it’s standard” – someone argued. And I tend to agree.