A Gentle Introduction to the Rectified Linear Unit (ReLU)

by **Jason Brownlee** on January 9, 2019 in **Deep Learning Performance**

Last Updated on August 6, 2019

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input.

The rectified linear activation function is a piecewise linear function that will output the input directly if is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

In this tutorial, you will discover the rectified linear activation function for deep learning neural networks.

After completing this tutorial, you will know:

- The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem.
- The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.
- The rectified linear activation is the default activation when developing multilayer Perceptron and convolutional neural networks.

Discover how to train faster, reduce overfitting, and make better predictions with deep learning models in my new book, with 26 step-by-step tutorials and full source code.

Let's get started.

- **Jun/2019**: Fixed error in the equation for He weight initialization (thanks Maltev).

A Gentle Introduction to the Rectified Linear Activation Function for Deep Learning Neural Networks
Photo by Bureau of Land Management, some rights reserved.

Tutorial Overview

This tutorial is divided into six parts; they are:

1. Limitations of Sigmoid and Tanh Activation Functions
2. Rectified Linear Activation Function
3. How to Implement the Rectified Linear Activation Function
4. Advantages of the Rectified Linear Activation
5. Tips for Using the Rectified Linear Activation
6. Extensions and Alternatives to ReLU

Limitations of Sigmoid and Tanh Activation Functions

A neural network is comprised of layers of nodes and learns to map examples of inputs to outputs.

For a given node, the inputs are multiplied by the weights in a node and summed together. This value is referred to as the summed activation of the node. The summed activation is then transformed via an activation function and defines the specific output or “activation” of the node.

The simplest activation function is referred to as the linear activation, where no transform is applied at all. A network comprised of only linear activation functions is very easy to train, but cannot learn complex mapping functions. Linear activation functions are still used in the output layer for networks that predict a quantity (e.g. regression problems).

Nonlinear activation functions are preferred as they allow the nodes to learn more complex structures in the data. Traditionally, two widely used nonlinear activation functions are the **sigmoid** and **hyperbolic tangent** activation functions.

The sigmoid activation function, also called the logistic function, is traditionally a very popular activation function for neural networks. The input to the function is transformed into a value between 0.0 and 1.0. Inputs that are much larger than 1.0 are transformed to the value 1.0, similarly, values much smaller than 0.0 are snapped to 0.0. The shape of the function for all possible inputs is an S-shape from zero up through 0.5 to 1.0. For a long time, through the early 1990s, it was the default activation used on neural networks.

The hyperbolic tangent function, or tanh for short, is a similar shaped nonlinear activation function that outputs values between -1.0 and 1.0. In the later 1990s and through the 2000s, the tanh function was preferred over the sigmoid activation function as models that used it were easier to train and often had better predictive performance.

> *… the hyperbolic tangent activation function typically performs better than the logistic sigmoid.*

— Page 195, Deep Learning, 2016.

A general problem with both the sigmoid and tanh functions is that they saturate. This means that large values snap to 1.0 and small values snap to -1 or 0 for tanh and sigmoid respectively. Further, the functions are only really sensitive to changes around their mid-point of their input, such as 0.5 for sigmoid and 0.0 for tanh.

The limited sensitivity and saturation of the function happen regardless of whether the summed activation from the node provided as input contains useful information or not. Once saturated, it becomes challenging for the learning algorithm to continue to adapt the weights to improve the performance of the model.

> *… sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0.*

— Page 195, Deep Learning, 2016.

Finally, as the capability of hardware increased through GPUs' very deep neural networks using sigmoid and tanh activation functions could not easily be trained.

Layers deep in large networks using these nonlinear activation functions fail to receive useful gradient information. Error is back propagated through the network and used to update the weights. The amount of error decreases dramatically with each additional layer through which it is propagated, given the derivative of the chosen activation function. This is called the vanishing gradient problem and prevents deep (multi-layered) networks from learning effectively.

> *Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function*

— Page 290, Deep Learning, 2016.

For an example of how ReLU can fix the vanishing gradients problem, see the tutorial:

- How to Fix Vanishing Gradients Using the Rectified Linear Activation Function

Although the use of nonlinear activation functions allows neural networks to learn complex mapping functions, they effectively prevent the learning algorithm from working with deep networks.

Workarounds were found in the late 2000s and early 2010s using alternate network types such as Boltzmann machines and layer-wise training or unsupervised pre-training.

Rectified Linear Activation Function

In order to use stochastic gradient descent with backpropagation of errors to train deep neural networks, an activation function is needed that looks and acts like a linear function, but is, in fact, a nonlinear function allowing complex relationships in the data to be learned.

The function must also provide more sensitivity to the activation sum input and avoid easy saturation.

The solution had been bouncing around in the field for some time, although was not highlighted until papers in 2009 and 2011 shone a light on it.

The solution is to use the rectified linear activation function, or ReL for short.

A node or unit that implements this activation function is referred to as a **rectified linear activation unit**, or ReLU for short. Often, networks that use the rectifier function for the hidden layers are referred to as rectified networks.

Adoption of ReLU may easily be considered one of the few milestones in the deep learning revolution, e.g. the techniques that now permit the routine development of very deep neural networks.

> *[another] major algorithmic change that has greatly improved the performance of feedforward networks was the replacement of sigmoid hidden units with piecewise linear hidden units, such as rectified linear units.*

— Page 226, Deep Learning, 2016.

The rectified linear activation function is a simple calculation that returns the value provided as input directly, or the value 0.0 if the input is 0.0 or less.

We can describe this using a simple if-statement:

```
if input > 0:
    return input
else:
    return 0
```

We can describe this function *g()* mathematically using the *max()* function over the set of 0.0 and the input *z*; for example:

```
g(z) = max{0, z}
```

The function is linear for values greater than zero, meaning it has a lot of the desirable properties of a linear activation function when training a neural network using backpropagation. Yet, it is a nonlinear function as negative values are always output as zero.

> *Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well.*

— Page 175, Deep Learning, 2016.

Because the rectified function is linear for half of the input domain and nonlinear for the other half, it is referred to as a piecewise linear function or a hinge function.

> *However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces.*

— Page 175, Deep Learning, 2016.

Now that we are familiar with the rectified linear activation function, let's look at how we can implement it in Python.

How to Code the Rectified Linear Activation Function

We can implement the rectified linear activation function easily in Python.

Perhaps the simplest implementation is using the max() function; for example:

```
# rectified linear function
def rectified(x):
    return max(0.0, x)
```

We expect that any positive value will be returned unchanged whereas an input value of 0.0 or a negative value will be returned as the value 0.0.

Below are a few examples of inputs and outputs of the rectified linear activation function.

```
# demonstrate the rectified linear function

# rectified linear function
def rectified(x):
    return max(0.0, x)

# demonstrate with a positive input
x = 1.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
x = 1000.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
# demonstrate with a zero input
x = 0.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
# demonstrate with a negative input
x = -1.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
x = -1000.0
print('rectified(%.1f) is %.1f' % (x, rectified(x)))
```

Running the example, we can see that positive values are returned regardless of their size, whereas negative values are snapped to the value 0.0.

```
rectified(1.0) is 1.0
rectified(1000.0) is 1000.0
rectified(0.0) is 0.0
rectified(-1.0) is 0.0
rectified(-1000.0) is 0.0
```

We can get an idea of the relationship between inputs and outputs of the function by plotting a series of inputs and the calculated outputs.

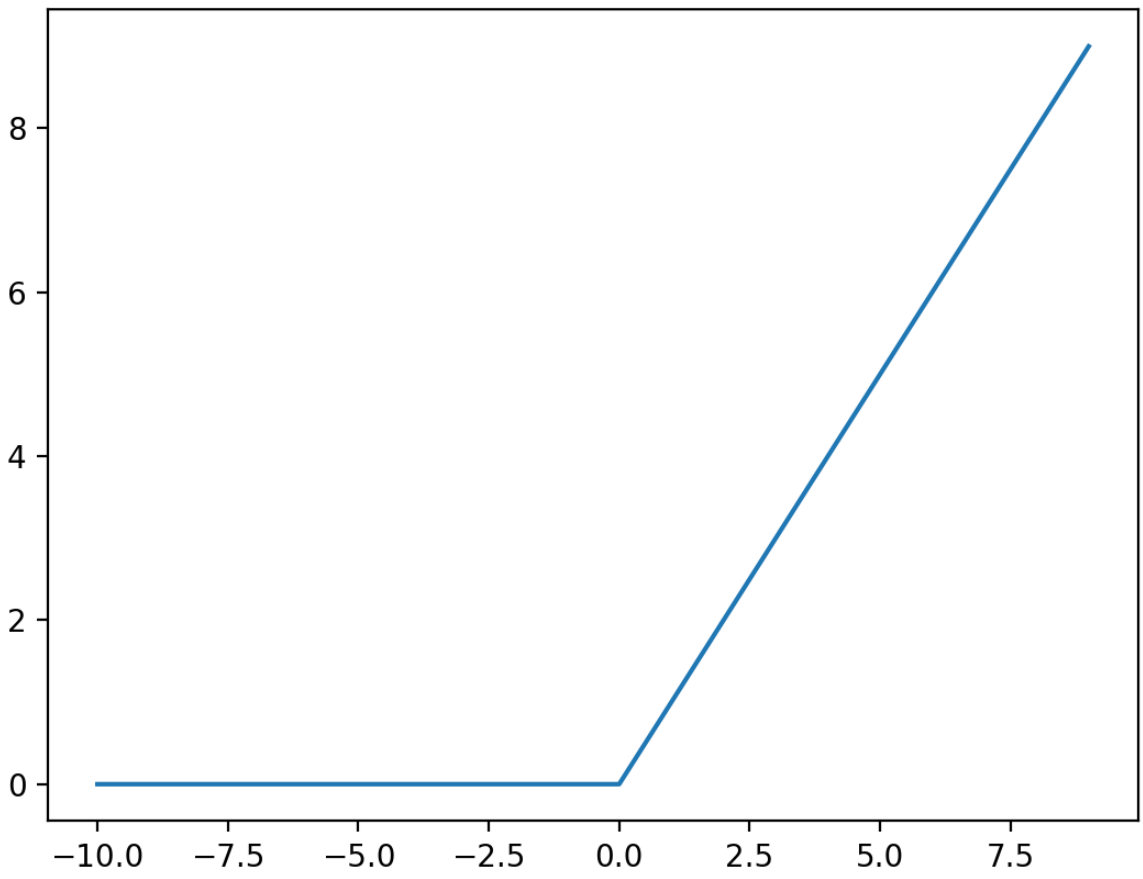
The example below generates a series of integers from -10 to 10 and calculates the rectified linear activation for each input, then plots the result.

```
# plot inputs and outputs
from matplotlib import pyplot

# rectified linear function
def rectified(x):
    return max(0.0, x)

# define a series of inputs
series_in = [x for x in range(-10, 11)]
# calculate outputs for our inputs
series_out = [rectified(x) for x in series_in]
# line plot of raw inputs to rectified outputs
pyplot.plot(series_in, series_out)
pyplot.show()
```

Running the example creates a line plot showing that all negative values and zero inputs are snapped to 0.0, whereas the positive outputs are returned as-is, resulting in a linearly increasing slope, given that we created a linearly increasing series of positive values (e.g. 1 to 10).



Line Plot of Rectified Linear Activation for Negative and Positive Inputs

The derivative of the rectified linear function is also easy to calculate. Recall that the derivative of the activation function is required when updating the weights of a node as part of the backpropagation of error.

The derivative of the function is the slope. The slope for negative values is 0.0 and the slope for positive values is 1.0.

Traditionally, the field of neural networks has avoided any activation function that was not completely differentiable, perhaps delaying the adoption of the rectified linear function and other piecewise-linear functions. Technically, we cannot calculate the derivative when the input is 0.0, therefore, we can assume it is zero. This is not a problem in practice.

> *For example, the rectified linear function g(z) = max{0, z} is not differentiable at z = 0. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks.*

— Page 192, Deep Learning, 2016.

Using the rectified linear activation function offers many advantages; let's take a look at a few in the next section.

Advantages of the Rectified Linear Activation Function

The rectified linear activation function has rapidly become the default activation function when developing most types of neural networks.

As such, it is important to take a moment to review some of the benefits of the approach, first highlighted by Xavier Glorot, et al. in their milestone 2012 paper on using ReLU titled "Deep Sparse Rectifier Neural Networks".

1. Computational Simplicity.

The rectifier function is trivial to implement, requiring a *max()* function.

This is unlike the tanh and sigmoid activation function that require the use of an exponential calculation.

> *Computations are also cheaper: there is no need for computing the exponential function in activations*

— Deep Sparse Rectifier Neural Networks, 2011.

2. Representational Sparsity

An important benefit of the rectifier function is that it is capable of outputting a true zero value.

This is unlike the tanh and sigmoid activation functions that learn to approximate a zero output, e.g. a value very close to zero, but not a true zero value.

This means that negative inputs can output true zero values allowing the activation of hidden layers in neural networks to contain one or more true zero values. This is called a sparse representation and is a desirable property in representational learning as it can accelerate learning and simplify the model.

An area where efficient representations such as sparsity are studied and sought is in autoencoders, where a network learns a compact representation of an input (called the code layer), such as an image or series, before it is reconstructed from the compact representation.

> *One way to achieve actual zeros in h for sparse (and denoising) autoencoders […] The idea is to use rectified linear units to produce the code layer. With a prior that actually pushes the representations to zero (like the absolute value penalty), one can thus indirectly control the average number of zeros in the representation.*

— Page 507, Deep Learning, 2016.

3. Linear Behavior

The rectifier function mostly looks and acts like a linear activation function.

In general, a neural network is easier to optimize when its behavior is linear or close to linear.

> *Rectified linear units […] are based on the principle that models are easier to optimize if their behavior is closer to linear.*

— Page 194, Deep Learning, 2016.

Key to this property is that networks trained with this activation function almost completely avoid the problem of vanishing gradients, as the gradients remain proportional to the node activations.

> *Because of this linearity, gradients flow well on the active paths of neurons (there is no gradient vanishing effect due to activation non-linearities of sigmoid or tanh units).*

— Deep Sparse Rectifier Neural Networks, 2011.

4. Train Deep Networks

Importantly, the (re-)discovery and adoption of the rectified linear activation function meant that it became possible to exploit improvements in hardware and successfully train deep multi-layered networks with a nonlinear activation function using backpropagation.

In turn, cumbersome networks such as Boltzmann machines could be left behind as well as cumbersome training schemes such as layer-wise training and unlabeled pre-training.

> *… deep rectifier networks can reach their best performance without requiring any unsupervised pre-training on purely supervised tasks with large labeled datasets. Hence, these results can be seen as a new milestone in the attempts at understanding the difficulty in training deep but purely supervised neural networks, and closing the performance gap between neural networks learnt with and without unsupervised pre-training.*

— Deep Sparse Rectifier Neural Networks, 2011.

Tips for Using the Rectified Linear Activation

In this section, we'll take a look at some tips when using the rectified linear activation function in your own deep learning neural networks.

Use ReLU as the Default Activation Function

For a long time, the default activation to use was the sigmoid activation function. Later, it was the tanh activation function.

For modern deep learning neural networks, the default activation function is the rectified linear activation function.

> *Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function or the hyperbolic tangent activation function.*

— Page 195, Deep Learning, 2016.

Most papers that achieve state-of-the-art results will describe a network using ReLU. For example, in the milestone 2012 paper by Alex Krizhevsky, et al. titled "ImageNet Classification with Deep Convolutional Neural Networks," the authors developed a deep convolutional neural network with ReLU activations that achieved state-of-the-art results on the ImageNet photo classification dataset.

> *… we refer to neurons with this nonlinearity as Rectified Linear Units (ReLUs). Deep convolutional neural networks with ReLUs train several times faster than their equivalents with tanh units.*

If in doubt, start with ReLU in your neural network, then perhaps try other piecewise linear activation functions to see how their performance compares.

> *In modern neural networks, the default recommendation is to use the rectified linear unit or ReLU*

— Page 174, Deep Learning, 2016.

Use ReLU with MLPs, CNNs, but Probably Not RNNs

The ReLU can be used with most types of neural networks.

It is recommended as the default for both Multilayer Perceptron (MLP) and Convolutional Neural Networks (CNNs).

The use of ReLU with CNNs has been investigated thoroughly, and almost universally results in an improvement in results, initially, surprisingly so.

> *… how do the non-linearities that follow the filter banks influence the recognition accuracy. The surprising answer is that using a rectifying non-linearity is the single most important factor in improving the performance of a recognition system.*

— What is the best multi-stage architecture for object recognition?, 2009

Work investigating ReLU with CNNs is what provoked their use with other network types.

> *[others] have explored various rectified nonlinearities […] in the context of convolutional networks and have found them to improve discriminative performance.*

— Rectified Linear Units Improve Restricted Boltzmann Machines, 2010.

When using ReLU with CNNs, they can be used as the activation function on the filter maps themselves, followed then by a pooling layer.

> *A typical layer of a convolutional network consists of three stages […] In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage.*

— Page 339, Deep Learning, 2016.

Traditionally, LSTMs use the tanh activation function for the activation of the cell state and the sigmoid activation function for the node output. Given their careful design, ReLU were thought to not be appropriate for Recurrent Neural Networks (RNNs) such as the Long Short-Term Memory Network (LSTM) by default.

> *At first sight, ReLUs seem inappropriate for RNNs because they can have very large outputs so they might be expected to be far more likely to explode than units that have bounded values.*

— A Simple Way to Initialize Recurrent Networks of Rectified Linear Units, 2015.

Nevertheless, there has been some work on investigating the use of ReLU as the output activation in LSTMs, the result of which is a careful initialization of network weights to ensure that the network is stable prior to training. This is outlined in the 2015 paper titled "A Simple Way to Initialize Recurrent Networks of Rectified Linear Units."

Try a Smaller Bias Input Value

The bias is the input on the node that has a fixed value.

The bias has the effect of shifting the activation function and it is traditional to set the bias input value to 1.0.

When using ReLU in your network, consider setting the bias to a small value, such as 0.1.

> *… it can be a good practice to set all elements of [the bias] 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.*

— Page 193, Deep Learning, 2016.

There are some conflicting reports as to whether this is required, so compare performance to a model with a 1.0 bias input.

Use "He Weight Initialization"

Before training a neural network,the weights of the network must be initialized to small random values.

When using ReLU in your network and initializing weights to small random values centered on zero, then by default half of the units in the network will output a zero value.

> *For example, after uniform initialization of the weights, around 50% of hidden units continuous output values are real zeros*

— Deep Sparse Rectifier Neural Networks, 2011.

There are many heuristic methods to initialize the weights for a neural network, yet there is no best weight initialization scheme and little relationship beyond general guidelines for mapping weight initialization schemes to the choice of activation function.

Prior to the wide adoption of ReLU, Xavier Glorot and Yoshua Bengio proposed an initialization scheme in their 2010 paper titled "Understanding the difficulty of training deep feedforward neural networks" that quickly became the default when using sigmoid and tanh activation functions, generally referred to as "*Xavier initialization*". Weights are set at random values sampled uniformly from a range proportional to the size of the number of nodes in the previous layer (specifically +/- *1/sqrt(n)* where *n* is the number of nodes in the prior layer).

Kaiming He, et al. in their 2015 paper titled "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification" suggested that Xavier initialization and other schemes were not appropriate for ReLU and extensions.

> *Glorot and Bengio proposed to adopt a properly scaled uniform distribution for initialization. This is called "Xavier" initialization […]. Its derivation is based on the assumption that the activations are linear. This assumption is invalid for ReLU*

— Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.

They proposed a small modification of Xavier initialization to make it suitable for use with ReLU, now commonly referred to as "*He initialization*" (specifically +/- *sqrt(2/n)* where *n* is the number of nodes in the prior layer known as the fan-in). In practice, both Gaussian and uniform versions of the scheme can be used.

Scale Input Data

It is good practice to scale input data prior to using a neural network.

This may involve standardizing variables to have a zero mean and unit variance or normalizing each value to the scale 0-to-1.

Without data scaling on many problems, the weights of the neural network can grow large, making the network unstable and increasing the generalization error.

This good practice of scaling inputs applies whether using ReLU for your network or not.

Use Weight Penalty

By design, the output from ReLU is unbounded in the positive domain.

This means that in some cases, the output can continue to grow in size. As such, it may be a good idea to use a form of weight regularization, such as an L1 or L2 vector norm.

> *Another problem could arise due to the unbounded behavior of the activations; one may thus want to use a regularizer to prevent potential numerical problems. Therefore, we use the L1 penalty on the activation values, which also promotes additional sparsity*

— Deep Sparse Rectifier Neural Networks, 2011.

This can be a good practice to both promote sparse representations (e.g. with L1 regularization) and reduced generalization error of the model.

Extensions and Alternatives to ReLU

The ReLU does have some limitations.

Key among the limitations of ReLU is the case where large weight updates can mean that the summed input to the activation function is always negative, regardless of the input to the network.

This means that a node with this problem will forever output an activation value of 0.0. This is referred to as a "*dying ReLU*".

> *the gradient is 0 whenever the unit is not active. This could lead to cases where a unit never activates as a gradient-based optimization algorithm will not adjust the weights of a unit that never activates initially. Further, like the vanishing gradients problem, we might expect learning to be slow when training ReL networks with constant 0 gradients.*

— Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013.

Some popular extensions to the ReLU relax the non-linear output of the function to allow small negative values in some way.

The Leaky ReLU (LReLU or LReL) modifies the function to allow small negative values when the input is less than zero.

> *The leaky rectifier allows for a small, non-zero gradient when the unit is saturated and not active*

— Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013.

The Exponential Linear Unit, or ELU, is a generalization of the ReLU that uses a parameterized exponential function to transition from the positive to small negative values.

> *ELUs have negative values which pushes the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient*

— Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016.

The Parametric ReLU, or PReLU, learns parameters that control the shape and leaky-ness of the function.

> *… we propose a new generalization of ReLU, which we call Parametric Rectified Linear Unit (PReLU). This activation function adaptively learns the parameters of the rectifiers*

— Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.

Maxout is an alternative piecewise linear function that returns the maximum of the inputs, designed to be used in conjunction with the dropout regularization technique.

> *We define a simple new model called maxout (so named because its output is the max of a set of inputs, and because it is a natural companion to dropout) designed to both facilitate optimization by dropout and improve the accuracy of dropout's fast approximate model averaging technique.*

— Maxout Networks, 2013.

Further Reading

This section provides more resources on the topic if you are looking to go deeper.

Posts

- How to Fix Vanishing Gradients Using the Rectified Linear Activation Function

Books

- Section 6.3.1 Rectified Linear Units and Their Generalizations, Deep Learning, 2016.

Papers

- What is the best multi-stage architecture for object recognition?, 2009
- Rectified Linear Units Improve Restricted Boltzmann Machines, 2010.
- Deep Sparse Rectifier Neural Networks, 2011.
- Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013.
- Understanding the difficulty of training deep feedforward neural networks, 2010.
- Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, 2015.
- Maxout Networks, 2013.

API

- max API

Articles

- Neural Network FAQ
- Activation function, Wikipedia.
- Vanishing gradient problem, Wikipedia.
- Rectifier (neural networks), Wikipedia.
- Piecewise Linear Function, Wikipedia.

Summary

In this tutorial, you discovered the rectified linear activation function for deep learning neural networks.

Specifically, you learned:

- The sigmoid and hyperbolic tangent activation functions cannot be used in networks with many layers due to the vanishing gradient problem.
- The rectified linear activation function overcomes the vanishing gradient problem, allowing models to learn faster and perform better.
- The rectified linear activation is the default activation when developing multilayer Perceptron and convolutional neural networks.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.