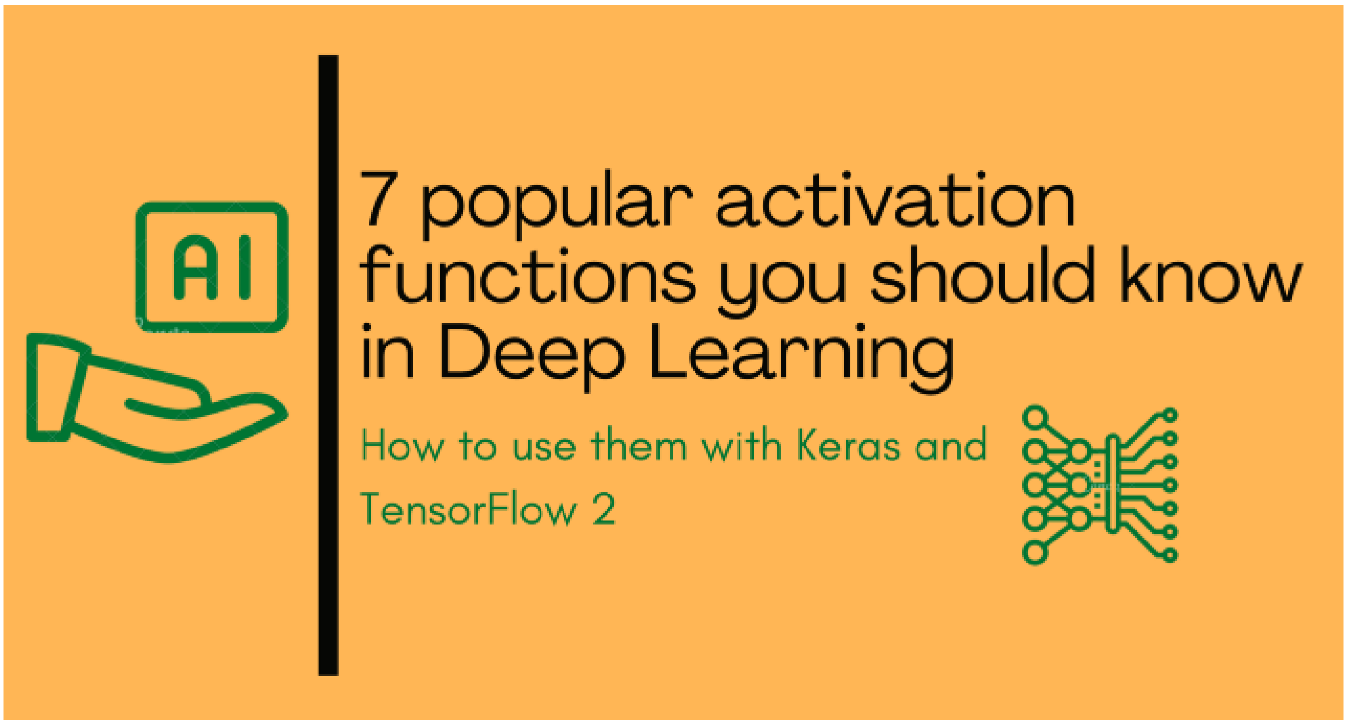
TENSORFLOW 2 TUTORIALS

7 popular activation functions you should know in Deep Learning and how to use them with Keras and TensorFlow 2

A practical introduction to Sigmoid, Tanh, ReLU, Leaky ReLU, PReLU, ELU, and SELU

B. Chen Jan 4 · 10 min read ★


AI
7 popular activation functions you should know in Deep Learning
How to use them with Keras and TensorFlow 2

7 popular activation functions in Deep Learning (Image by author using canva.com)

[1].

The activation functions are at the very core of Deep Learning. They determine the output of a model, its accuracy, and computational efficiency. In some cases, activation functions have a major effect on the model's ability to converge and the convergence speed.

In this article, you'll learn the following most popular activation functions in Deep Learning and how to use them with Keras and TensorFlow 2.

1. Sigmoid (Logistic)
2. Hyperbolic Tangent (Tanh)
3. Rectified Linear Unit (ReLU)
4. Leaky ReLU
5. Parametric Leaky ReLU (PReLU)
6. Exponential Linear Units (ELU)
7. Scaled Exponential Linear Unit (SELU)

Please check out Notebook for the source code.

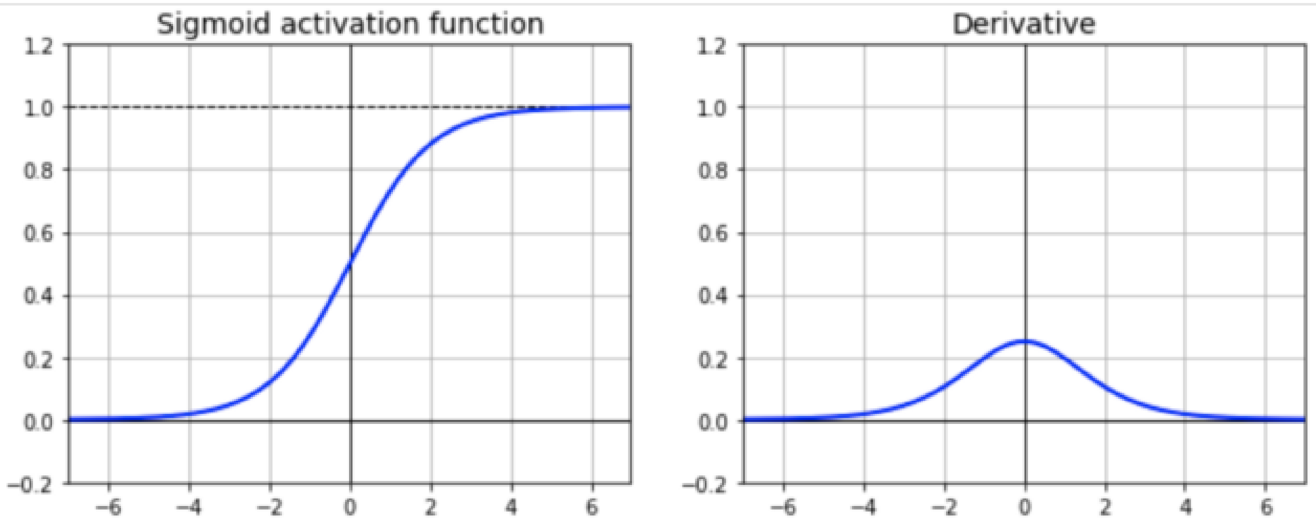
. . .

1. Sigmoid (Logistic)

The **Sigmoid function** (also known as the **Logistic function**) is one of the most widely used activation function. The function is defined as:

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

Sigmoid activation function (Image by author)



the plot of Sigmoid function and its derivative (Image by author)

As we can see in the plot above,

- The function is a common **S-shaped** curve.
- The output of the function is centered at **0.5** with a range from **0** to **1**.
- The function is **differentiable**. That means we can find the slope of the sigmoid curve at any two points.
- The function is **monotonic** but the function's derivative is not.

The **Sigmoid** function was introduced to Artificial Neural Networks (ANN) in the 1990s to replace the **Step** function [2]. It was a key change to ANN architecture because the **Step** function doesn't have any gradient to work with Gradient Descent, while the **Sigmoid** function has a well-defined nonzero derivative everywhere, allowing Gradient Descent to make some progress at every step during training.

Problems with Sigmoid activation function

The main problems with the Sigmoid function are:

1. **Vanishing gradient**: looking at the function plot, you can see that when inputs become small or large, the function saturates at 0 or 1, with a derivative extremely close to 0. Thus it has almost no gradient to propagate back through the network, so there is almost nothing left for lower layers [2].
2. **Computationally expensive:** the function has an exponential operation.

To use the Sigmoid activation function with Keras and TensorFlow 2, we can simply pass `'sigmoid'` to the argument `activation` :

```
from tensorflow.keras.layers import Dense

Dense(10, activation='sigmoid')
```

To apply the function for some constant inputs:

```
import tensorflow as tf
from tensorflow.keras.activations import sigmoid

z = tf.constant([-20, -1, 0, 1.2], dtype=tf.float32)
output = sigmoid(z)
output.numpy()
```

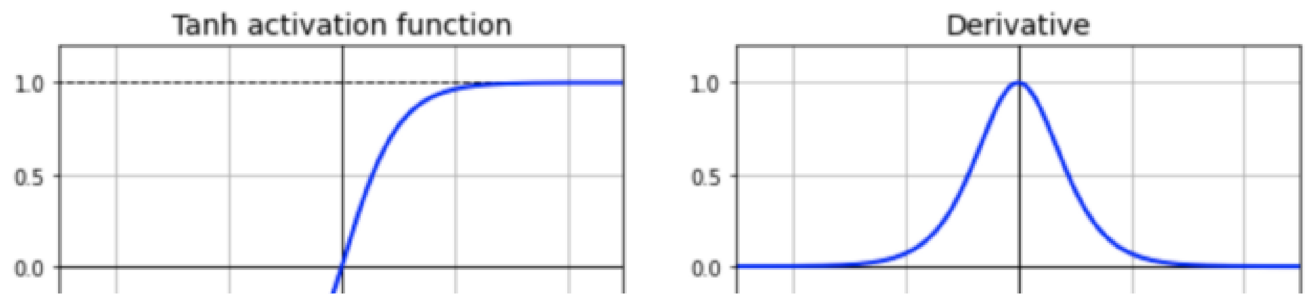
2. Hyperbolic Tangent (Tanh)

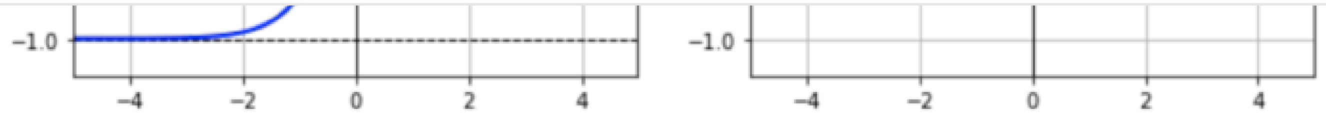
Another very popular and widely used activation function is the **Hyperbolic Tangent**, also known as **Tanh**. It is defined as:

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

tanh function (image by author)

The plot of the function and its derivative:





The plot of tanh and its derivative (image by author)

We can see that the function is very similar to the Sigmoid function.

- The function is a common **S-shaped** curve as well.
- The difference is that the output of **Tanh** is **zero centered** with a range from **-1** to **1** (instead of 0 to 1 in the case of the Sigmoid function)
- The same as the Sigmoid, this function is **differentiable**
- The same as the Sigmoid, the function is **monotonic**, but the function's derivative is not.

Tanh has characteristics similar to **Sigmoid** that can work with Gradient Descent. One important point to mention is that **Tanh** tends to make each layer's output more or less centered around 0 and this often helps speed up convergence [2].

Problems with Tanh activation function

Since **Tanh** has characteristics similar to **Sigmoid**, it also faces the following two problems:

1. **Vanishing gradient**: looking at the function plot, you can see that when inputs become small or large, the function saturates at -1 or 1, with a derivative extremely close to 0. Thus it has almost no gradient to propagate back through the network, so there is almost nothing left for lower layers.
2. **Computationally expensive:** the function has an exponential operation.

How to use Tanh with Keras and TensorFlow 2

To use the Tanh, we can simply pass `'tanh'` to the argument `activation`:

```
from tensorflow.keras.layers import Dense

Dense(10, activation='tanh')
```

```
import tensorflow as tf
from tensorflow.keras.activations import tanh

z = tf.constant([-20, -1, 0, 1.2], dtype=tf.float32)
output = tanh(z)
output.numpy()
```

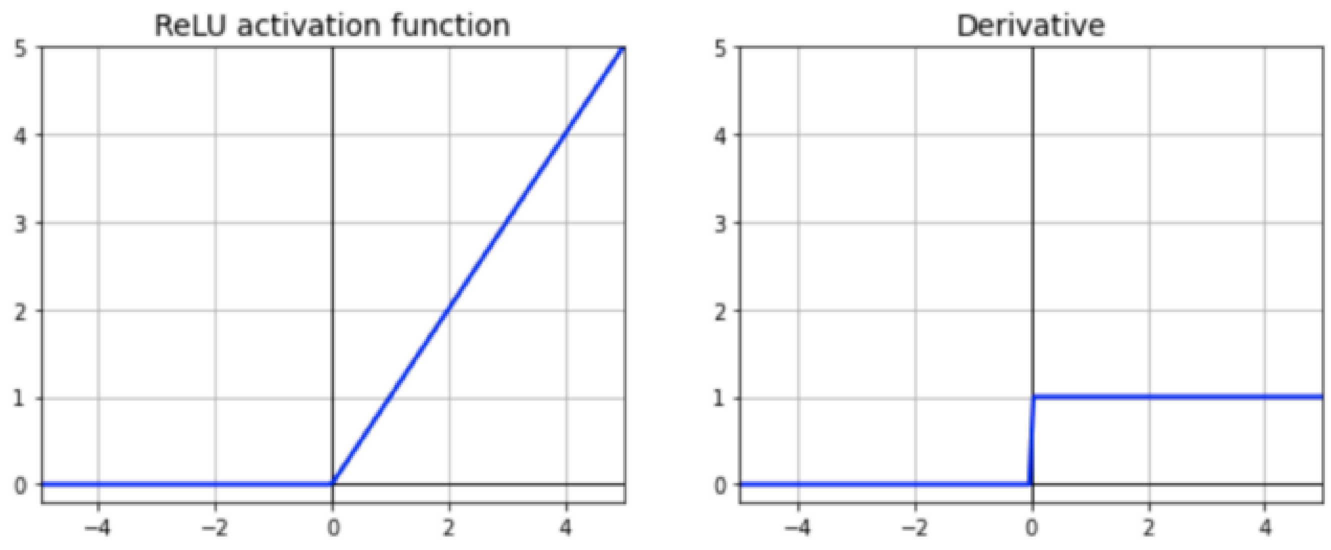
3. Rectified Linear Unit (ReLU)

The **Rectified Linear Unit (ReLU)** is the most commonly used activation function in deep learning. The function returns 0 if the input is negative, but for any positive input, it returns that value back. The function is defined as:

$$\begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$$

ReLU function (image by author)

The plot of the function and its derivative:



The plot of ReLU and its derivative (image by author)

As we can see that:

is non-linear around 0, but the slope is always either 0 (for negative inputs) or 1 (for positive inputs).

- The ReLU function is **continuous**, but it is **not differentiable** because its derivative is 0 for any negative input.
- The output of ReLU does not have a maximum value (It is **not saturated**) and this helps Gradient Descent
- The function is very fast to compute (Compare to Sigmoid and Tanh)

It's surprising that such a simple function works very well in deep neural networks.

Problem with ReLU

ReLU works great in most applications, but it is not perfect. It suffers from a problem known as the **dying ReLU**.

> ***Dying ReLU***
>
> *During training, some neurons effectively die, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network's neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the weighted sum of its inputs are negative for all instances in the training set. When this happens, it just keeps outputting 0s, and gradient descent does not affect it anymore since the gradient of the ReLU function is 0 when its input is negative.*
>
> *Hands-on Machine Learning [2], page 329*

How to use it with Keras and TensorFlow 2

To use ReLU with Keras and TensorFlow 2, just set `activation='relu'`

```
from tensorflow.keras.layers import Dense

Dense(10, activation='relu')
```

To apply the function for some constant inputs:

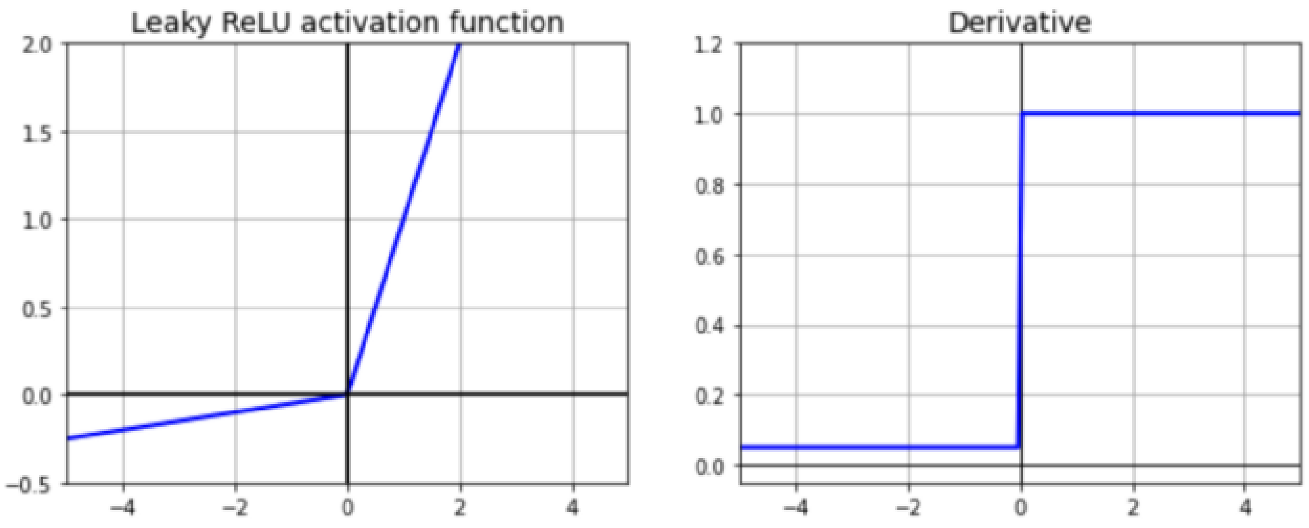
```
z = tf.constant([-20, -1, 0, 1.2], dtype=tf.float32)
output = relu(z)
output.numpy()
```

4. Leaky ReLU

Leaky ReLU is an improvement over the ReLU activation function. It has all properties of ReLU, plus it will never have **dying ReLU** problem. Leaky ReLU is defined as:

```
f(x) = max(αx, x)
```

The hyperparameter `α` defines how much the function leaks. It is the slope of the function for x < 0 and is typically set to `0.01`. The small slope ensures that Leaky ReLU never dies.



How to use Leaky ReLU with Keras and TensorFlow 2

To use the **Leaky ReLU** activation function, you must create a `LeakyReLU` instance like below:

```
from tensorflow.keras.layers import LeakyReLU, Dense

leaky_relu = LeakyReLU(alpha=0.01)
Dense(10, activation=leaky_relu)
```

to be learned during training (instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameters). This was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set [2].

How to use PReLU with Keras and TensorFlow 2

To use Parametric leaky ReLU, you must create a `PReLU` instance like below:

```
from tensorflow.keras.layers import PReLU, Dense

para_relu = PReLU()
Dense(10, activation=para_relu)
```

6. Exponential Linear Unit (ELU)

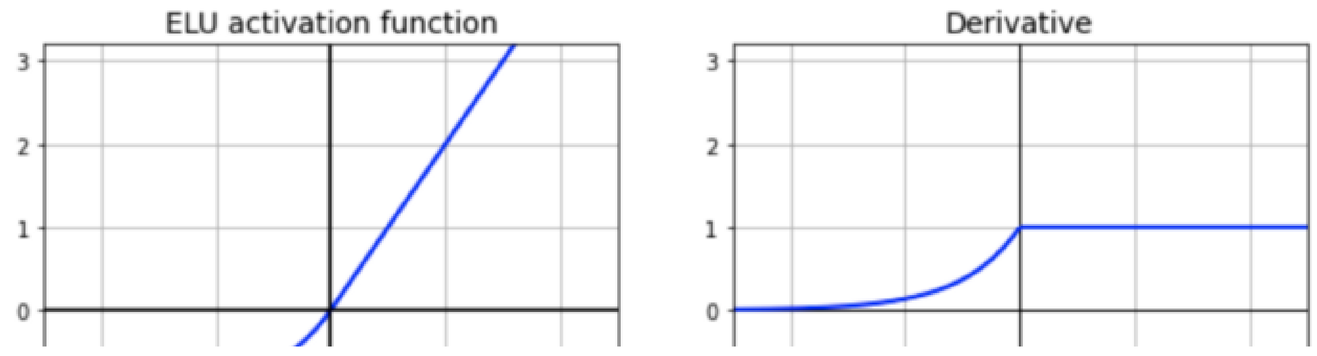
Exponential Linear Unit (**ELU**) is a variation of **ReLU** with a better output for z < 0. The function is defined as:

$$\begin{cases} \alpha\left(e^{x}-1\right) & \text{if } x \le 0 \\ x & \text{if } x > 0 \end{cases}$$

ELU function (image by author)

The hyperparameter α controls the value to which an **ELU** saturates for negative net inputs.

The plot of the function and its derivative:



The plot of ELU and its derivative (image by author)

We can see in the plot above,

- **ELU** modified the slope of the negative part of the function.
- Unlike the **Leaky ReLU** and **PReLU** functions, instead of a straight line, **ELU** uses a log curve for the negative values.

According to the authors, ELU outperformed all the ReLU variants in their experiments [3].

Problem with ELU

According to [2, 3], the main drawback of the ELU activation is that it is slower to compute than the ReLU and its variants (due to the use of the exponential function), but during training this is compensated by the faster convergence rate. However, at test time, an ELU network will be slower than a ReLU network.

How to use it with Keras and TensorFlow 2

Implementing ELU in TensorFlow 2 is trivial, just specify the activation function when building each layer:

```
Dense(10, activation='elu')
```

To apply the function for some constant inputs:

```
import tensorflow as tf
from tensorflow.keras.activations import elu

z = tf.constant([-20, -1, 0, 1.2], dtype=tf.float32)
output = elu(z, alpha=1)
output.numpy()
```

7. Scaled Exponential Linear Unit (SELU)

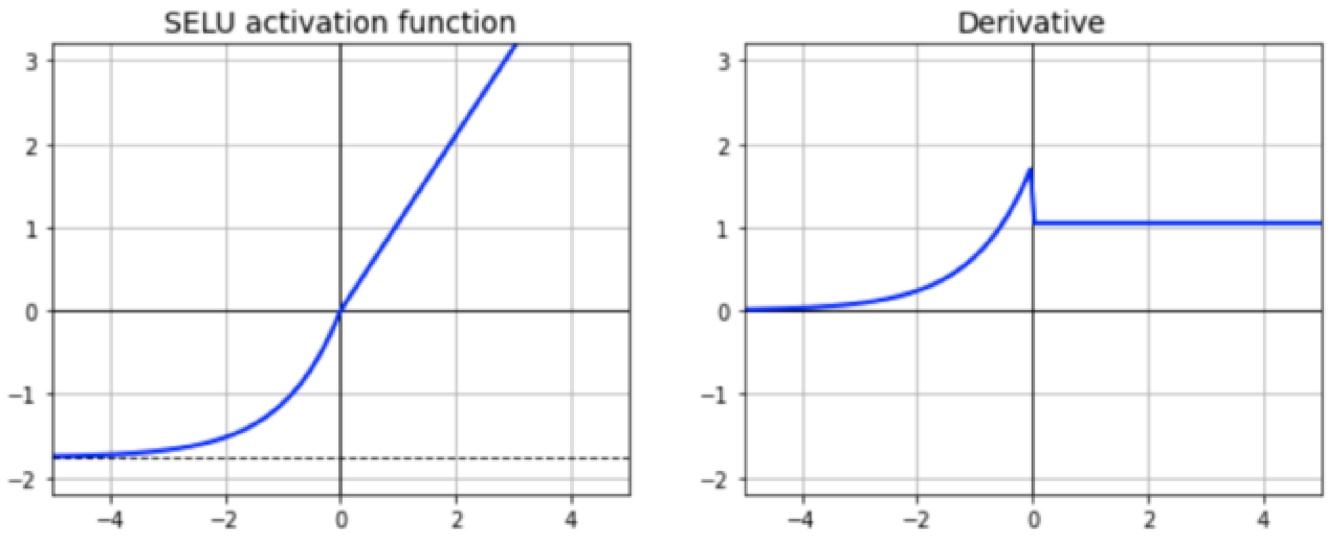
a neural network composed exclusively of a stack of dense layers, and if all hidden layers use the **SELU** activation function, then the network will self-normalize (the output of each layer will tend to preserve mean 0 and standard deviation 1 during training, which resolves the vanishing/exploding gradients problem). This activation function often outperforms other activation functions very significantly.

SELU is defined as:

```
f(x) = scale * x                    , z > 0
     = scale * α * (exp(x) - 1)     , z <= 0
```

where `α` and `scale` are pre-defined constants (`α=1.67326324` and `scale=1.05070098`).

The plot of SELU and its derivative:



The plot of SELU and its derivative (Image by author)

Problem with SELU

The main problem with SELU is that there are a few conditions for SELU to work:

- SELU works only for a neural network composed exclusively of a stack of dense layers. It might not work for convolutional neural networks.
- Every hidden layer's weights must also be initialized using LeCun normal initialization.

To use SELU with Keras and TensorFlow 2, just set `activation='selu'` and `kernel_initializer='lecun_normal'`:

```
from tensorflow.keras.layers import Dense

Dense(10, activation='relu', kernel_initializer='lecun_normal')
```

How to choose an activation function?

We have gone through 7 different activation functions in deep learning. When building a model, the selection of activation functions is critical. So which activation function should you use? Here is a general suggestion from the book Hands-on ML

> *Although your mileage will vary, in general SELU > ELU > leaky ReLU (and its variants) > ReLU > tanh > logistic. If the network's architecture prevents it from self-normalizing, then ELU may perform better than SELU (since SELU is not smooth at z = 0). If you care a lot about runtime latency, then you may prefer leaky ReLU. If you don't want to tweak yet another hyperparameter, you may just use the default α values used by Keras (e.g., 0.3 for the leaky ReLU). If you have spare time and computing power, you can use cross-validation to evaluate other activation functions, in particular, RReLU if your network is over-fitting, or PReLU if you have a huge training set.*
>
> *Hands-on ML, page 332*

Conclusion

In this article, we have gone through 7 different activation functions in Deep Learning and how to use them with Keras and TensorFlow.

I hope this article will help you to save time in building and tuning your own Deep Learning model. I recommend you to check out the Keras documentation for the activation functions and to know about other things you can do.

Thanks for reading. Please check out the notebook for the source code and stay tuned if you are interested in the practical aspect of machine learning.

You may be interested in some of my other TensorFlow articles:

- Learning Rate Schedules in practice
- The Google's 7 steps of Machine Learning in practice: a TensorFlow example for structured data
- 3 ways to create a Machine Learning Model with Keras and TensorFlow 2.0
- Batch normalization in practice: an example with Keras and TensorFlow 2.0
- Early stopping in Practice: an example with Keras and TensorFlow

More can be found from my Github

References

- [1] 7 Types of Neural Network activation function
- [2] Hands-on Machine Learning
- [3] Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)
- [4] Self-Normalizing Neural Networks

Deep Learning Machine Learning Keras Activation Functions Neurons