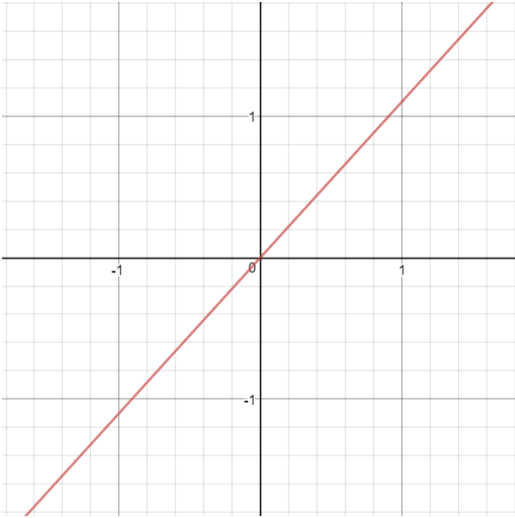
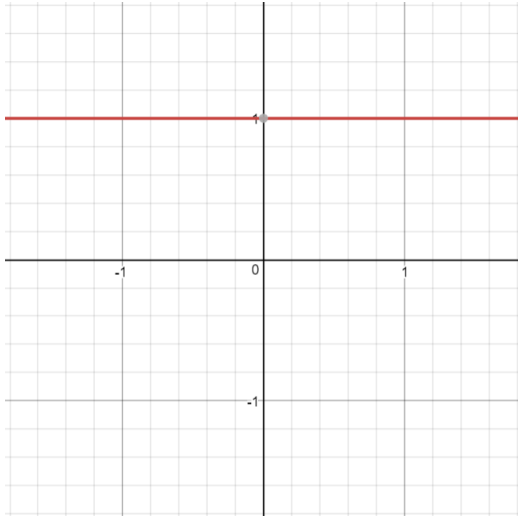


Activation Functions

- [Linear](#)
- [ELU](#)
- [ReLU](#)
- [LeakyReLU](#)
- [Sigmoid](#)
- [Tanh](#)
- [Softmax](#)

Linear

A straight line function where activation is proportional to input (which is the weighted sum from neuron).

Function	Derivative
$R(z, m) = \{ z * m \}$	$R'(z, m) = \{ m \}$
	
<pre>def linear(z,m): return m*z</pre>	<pre>def linear_prime(z,m): return m</pre>

Pros

- It gives a range of activations, so it is not binary activation.

- We can definitely connect a few neurons together and if more than 1 fires, we could take the max (or softmax) and decide based on that.

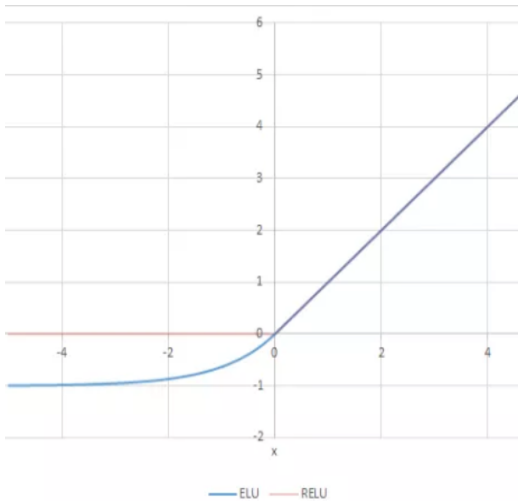
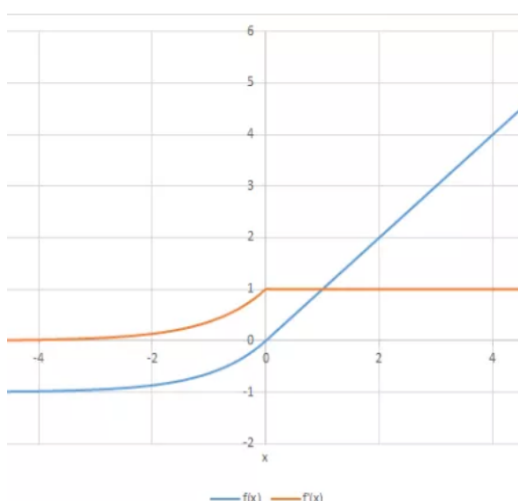
Cons

- For this function, derivative is a constant. That means, the gradient has no relationship with X.
- It is a constant gradient and the descent is going to be on constant gradient.
- If there is an error in prediction, the changes made by back propagation is constant and not depending on the change in input delta(x) !

ELU

Exponential Linear Unit or its widely known name ELU is a function that tend to converge cost to zero faster and produce more accurate results. Different to other activation functions, ELU has a extra alpha constant which should be positive number.

ELU is very similiar to RELU except negative inputs. They are both in identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -alpha whereas RELU sharply smoothes.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha \cdot (e^z - 1) & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha \cdot e^z & z < 0 \end{cases}$
	
<pre>def elu(z,alpha): return z if z >= 0 else alpha* (e^z -1)</pre>	<pre>def elu_prime(z,alpha): return 1 if z > 0 else alpha*np.exp(z)</pre>

Pros

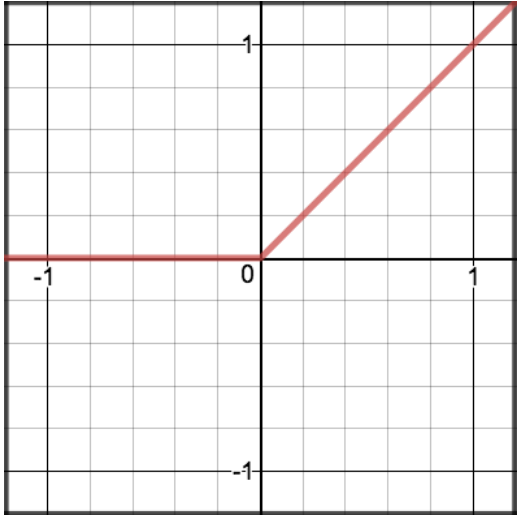
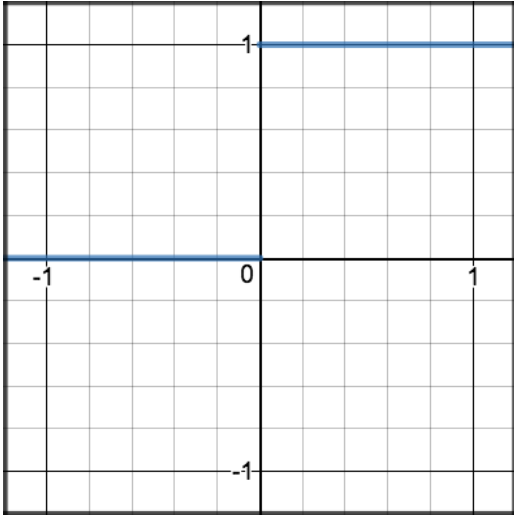
- ELU becomes smooth slowly until its output equal to $-\alpha$ whereas RELU sharply smooths.
- ELU is a strong alternative to ReLU.
- Unlike to ReLU, ELU can produce negative outputs.

Cons

- For $x > 0$, it can blow up the activation with the output range of $[0, \text{inf}]$.

ReLU

A recent invention which stands for Rectified Linear Units. The formula is deceptively simple: $\max(0, z)$. Despite its name and appearance, it's not linear and provides the same benefits as Sigmoid but with better performance.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$
	
<pre>def relu(z): return max(0, z)</pre>	<pre>def relu_prime(z): return 1 if z > 0 else 0</pre>

Pros

- It avoids and rectifies vanishing gradient problem.
- ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.

Cons

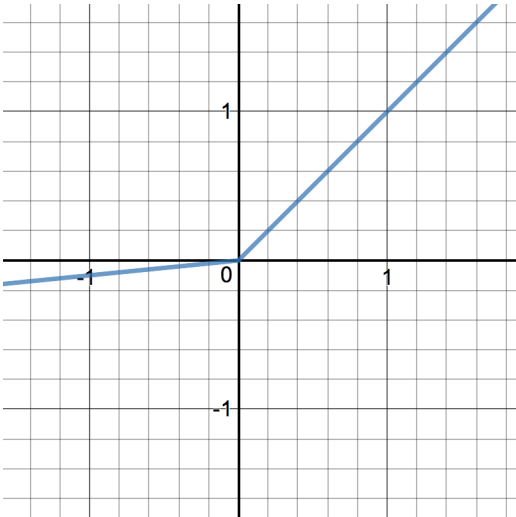
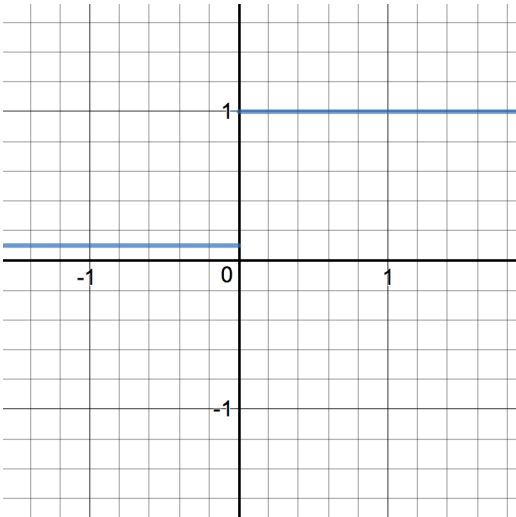
- One of its limitation is that it should only be used within Hidden layers of a Neural Network Model.
- Some gradients can be fragile during training and can die. It can cause a weight update which will makes it never activate on any data point again. Simply saying that ReLu could result in Dead Neurons.
- In another words, For activations in the region ($x < 0$) of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called dying ReLu problem.
- The range of ReLu is $[0, \infty)$. This means it can blow up the activation.

Further reading

- [Deep Sparse Rectifier Neural Networks](#) Glorot et al., (2011)
- [Yes You Should Understand Backprop](#), Karpathy (2016)

LeakyReLU

LeakyRelu is a variant of ReLU. Instead of being 0 when $z < 0$, a leaky ReLU allows a small, non-zero, constant gradient α (Normally, $\alpha = 0.01$). However, the consistency of the benefit across tasks is presently unclear. ^[1]

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$
	
<pre>def leakyrelu(z, alpha): return max(alpha * z, z)</pre>	<pre>def leakyrelu_prime(z, alpha): return 1 if z > 0 else alpha</pre>

Pros

- Leaky ReLUs are one attempt to fix the “dying ReLU” problem by having a small negative slope (of 0.01, or so).

Cons

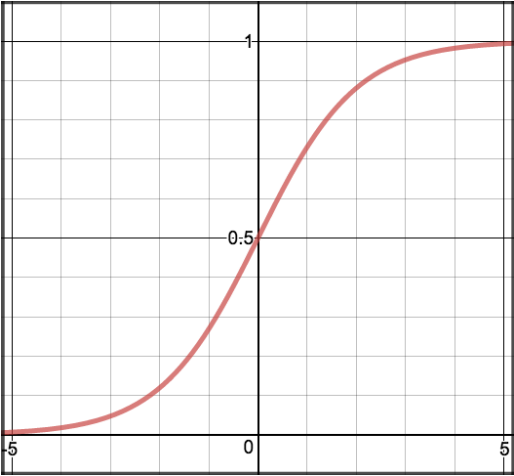
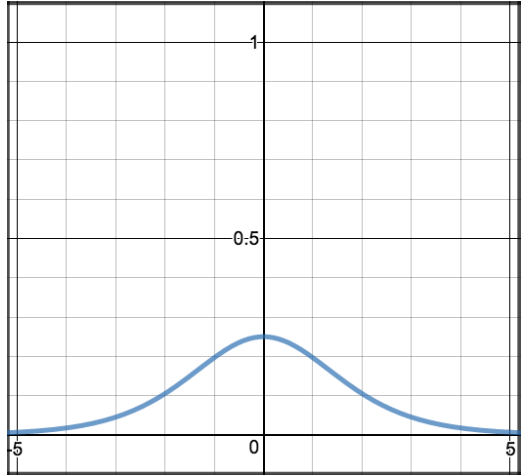
- As it possess linearity, it can't be used for the complex Classification. It lags behind the Sigmoid and Tanh for some of the use cases.

Further reading

- [Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification](#), Kaiming He et al. (2015)

Sigmoid

Sigmoid takes a real value as input and outputs another value between 0 and 1. It's easy to work with and has all the nice properties of activation functions: it's non-linear, continuously differentiable, monotonic, and has a fixed output range.

Function	Derivative
$S(z) = \frac{1}{1 + e^{-z}}$	$S'(z) = S(z) \cdot (1 - S(z))$
	
<pre>def sigmoid(z): return 1.0 / (1 + np.exp(-z))</pre>	<pre>def sigmoid_prime(z): return sigmoid(z) * (1-sigmoid(z))</pre>

Pros

- It is nonlinear in nature. Combinations of this function are also nonlinear!

- It will give an analog activation unlike step function.
- It has a smooth gradient too.
- It's good for a classifier.
- The output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. Nice, it won't blow up the activations then.

Cons

- Towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.
- It gives rise to a problem of “vanishing gradients”.
- Its output isn't zero centered. It makes the gradient updates go too far in different directions. $0 < \text{output} < 1$, and it makes optimization harder.
- Sigmoids saturate and kill gradients.
- The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

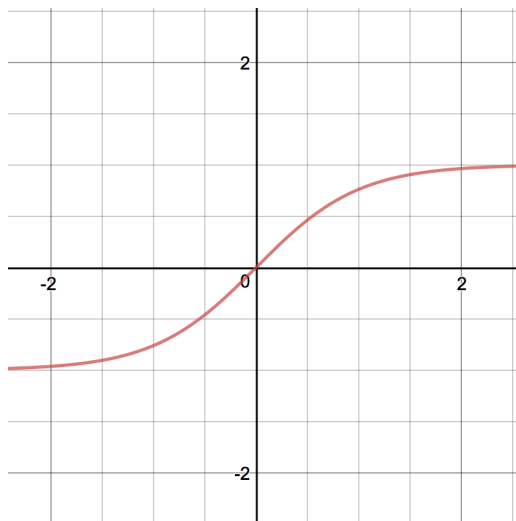
Further reading

- [Yes You Should Understand Backprop](#), Karpathy (2016)

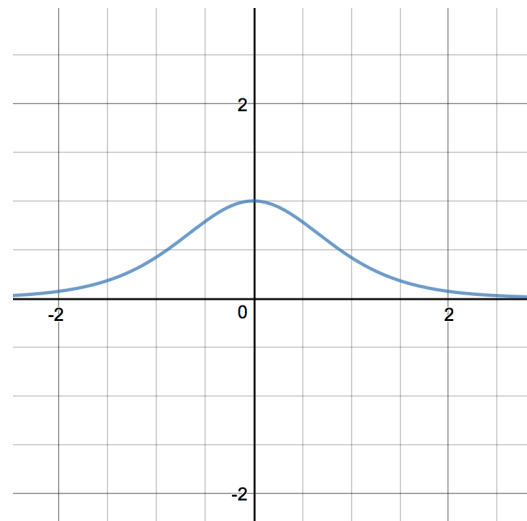
Tanh

Tanh squashes a real-valued number to the range $[-1, 1]$. It's non-linear. But unlike Sigmoid, its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity. [\[1\]](#)

Function	Derivative
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$\tanh'(z) = 1 - \tanh(z)^2$



```
def tanh(z):
    return (np.exp(z) - np.exp(-
z)) / (np.exp(z) + np.exp(-z))
```



```
def tanh_prime(z):
    return 1 - np.power(tanh(z),
2)
```

Pros

- The gradient is stronger for tanh than sigmoid (derivatives are steeper).

Cons

- Tanh also has the vanishing gradient problem.

Softmax 🔗

Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

References

- [1] (1, 2) <http://cs231n.github.io/neural-networks-1/>