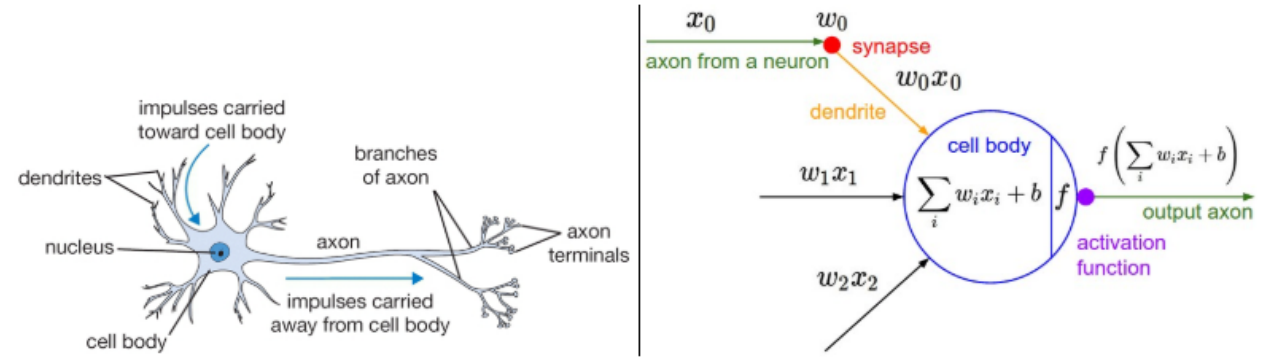
Different Types of Activation Units – Note

1. Introduction

In artificial neural networks, the *activation function* of a neuron defines the output of that node given the linear combination of its input. The idea of activation functions, together with its name, is inspired by the activation in the model of human brains, as is shown in Figure 1 [1]. Activation function is usually an element-wise mapping from $\mathbb{R}$ to $\mathbb{R}$. Because neural networks are trained using first order methods such as stochastic gradient descent, the activation functions need to be differentiable (almost everywhere).



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

Figure 1. A biological neuron versus a neuron in an artificial neuron network. This figure is from Stanford CS231n.

Activation functions can help an artificial neural network learn complex patterns in the data because it can add non-linearity to a neural network. If there is no activation function or the activation function is linear, then the mapping $\mathbf{x} \mapsto \mathbf{A}\mathbf{x} + b$ will be a linear mapping. Consequently, no matter how many layer does the neural network have, its expressiveness will be no better than a linear model. Instead, if there is any non-trivial activation function, the expressiveness of the neural network will be greatly improved. This fact is supported by the *universal approximation theorem* [2]:

Universal Approximation Theorem: Fix a continuous function $\sigma : \mathbb{R} \to \mathbb{R}$ (activation function) and positive integers d, D. The following two statements are equivalent:

(1) The function σ is not a polynomial
(2) for every continuous function $f : \mathbb{R}^d \to \mathbb{R}^D$ (target function), for every compact subset $K \subset \mathbb{R}^d$, for every $\epsilon > 0$, there exists a continuous function $f_\epsilon : \mathbb{R}^d \to \mathbb{R}^D$ (the function induced by the neural network) with representation $f_\epsilon : \mathbf{x} \mapsto \mathbf{W}_2\sigma(\mathbf{W}_1\mathbf{x})$, such that the approximation bound $\sup_{\mathbf{x}\in K}\|f(\mathbf{x}) - f_\epsilon(\mathbf{x})\| < \epsilon$

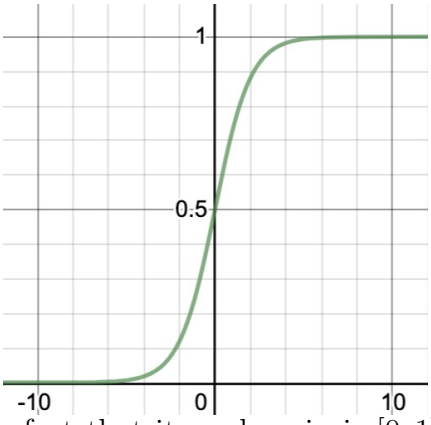
The universal approximation theorem implies that if an activation function is not a polynomial, an infinitely wide single-layer neural network with this activation function can approximate any continuous function. Compared with the case where the activation is linear and the neural network can only express linear mappings, the non-linear (and non-polynomial) activation function plays an important role in boosting the expressiveness of a neural network.

2. Common Activation Units

There are many different types of activation units. In the following, we introduce some of the most commonly used:

Sigmoid:

$$f(z) = \frac{1}{1 + e^{-z}}$$



The sigmoid function is used due heavily to the fact that its codomain is $[0, 1]$, and that it is differentiable everywhere. However, it has the disadvantage of **vanishing gradient**.
Recall that in doing back-propagation, we need to calculate the derivative of the loss function with respect to weights. The gradient with respect to weights W_ℓ in layer ℓ depends on the error term $\delta^{\ell+1}$ in the next layer $\ell + 1$ and the derivative of the activation function.

$$\begin{aligned} \delta^\ell &= \left((W^\ell)^T \delta^{\ell+1}\right) \cdot f'(z^\ell) \\ \frac{\partial \mathcal{L}}{\partial W_\ell} &= X^{\ell-1} \delta^\ell. \end{aligned}$$

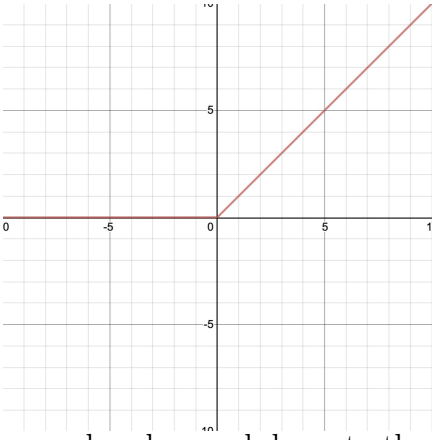
where $\mathcal{L}$ denotes the loss function, z the weighted sum at the neurons, and $X^{\ell-1}$ the input at layer $\ell - 1$. Hence, the update of weights is proportional to the multiplication of derivatives of activation functions in the following layers. That is.

$$\frac{\partial \mathcal{L}}{\partial W_\ell} \propto f'(z^\ell) f'(z^{\ell+1}) \cdots f'(z^{\ell+n}).$$

If the number of layers n is large, and that $f'(z) \leq 1$, the gradient will become very small and vanish. This can happen quite frequently when sigmoid is used as the activation function in a neural network of moderate size because when the input values are too large or too small, the gradient of sigmoid gets close to zero. This can be easily seen in the graph above. In practice, the vanishing of gradients can occur when the network is poorly initialized or when the learning rate is set too high. Fortunately, rectified linear unit, ReLU, can be used in place of sigmoid to alleviate this problem.

ReLU:

$$f(z) = \begin{cases} z & if \quad z \geq 0, \\ 0 & else. \end{cases}$$



When z is greater than or equal to zero, it is preserved and passed down to the next layer, otherwise, the value dies to zero. Thus, it can also be written as $f(x) = \max(x, 0)$. Also, ReLU is said to bear the biological resemblance to how the human brain works.

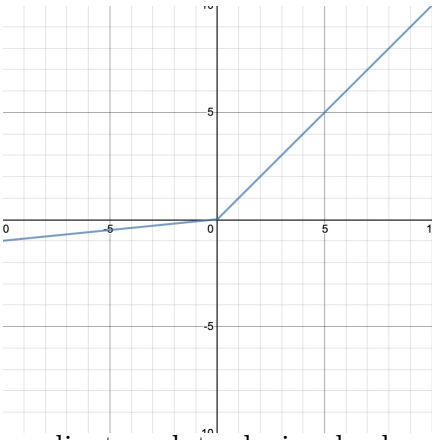
Let's calculate the derivative of the ReLU function.

$$f'(z) = \begin{cases} 1 & if \quad z \geq 0, \\ 0 & else. \end{cases}$$

The derivative of the ReLU function is undefined at $z = 0$ but it can be assigned as either 0 or 1, and we assign it to be 1 here. We can see that the gradient vanishing problem disappears for $z \geq 0$ because it is constant 1. However, when $z < 0$, the problem persists, which is also called the **dying ReLU problem**. If all inputs to ReLU are stuck on the negative side, there is no salvaging since the gradient is zero. We can introduce another parameter to make the gradient of ReLU function nonzero when $z < 0$. There comes Parametric ReLU.

Parametric ReLU:

$$f(z) = \begin{cases} z & if \quad z \geq 0, \\ az & else. \end{cases}$$



To solve the problem in ReLU that there's no gradient update during backpropagation when $z < 0$, parametric ReLU introduces a variable a as the slope when $z < 0$. Same as how other model variables such as the weights and biases are learned, the slope a is learned through backpropagation. Every layer ℓ has a different a^ℓ to learn. The update rule can be written as

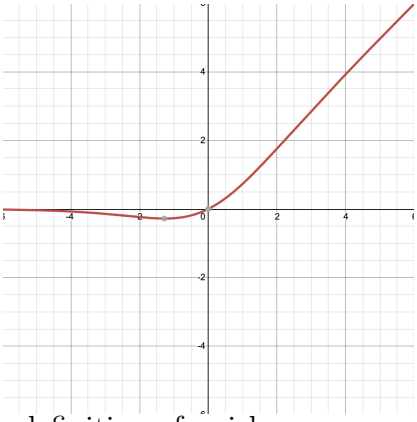
$$\begin{aligned} \frac{\partial f(z^\ell)}{\partial a^\ell} &= \begin{cases} 0 & if \quad z^\ell \geq 0, \\ z^\ell & else. \end{cases} \\ \frac{\partial \mathcal{L}}{\partial a^\ell} &= \sum_{z_i^\ell} \frac{\partial \mathcal{L}}{\partial f(z_i^\ell)} \frac{\partial f(z_i^\ell)}{\partial a^\ell}, \end{aligned}$$

where the term $\frac{\partial \mathcal{L}}{\partial f(z_i^\ell)}$ is the gradient propagated from the deeper layer and the summation $\sum_{z_i^\ell}$ runs over all positions of the feature map.
This method introduces more variables to the model but the number of additional variables is the number of layers, which is way smaller than the weights and biases. This doesn't burden the training too much and excludes extra possibilities of overfitting. Reader is encouraged to read more in this paper [3]. Leaky ReLU is a variation when the variable a is hardcoded 0.01.

In the following, we introduce two latest state-of-the-art activation functions, swish and mish. You will also see them in action in the corresponding coding assignment.

Swish:

$$f(z) = \frac{z}{1 + e^{-\beta z}}$$



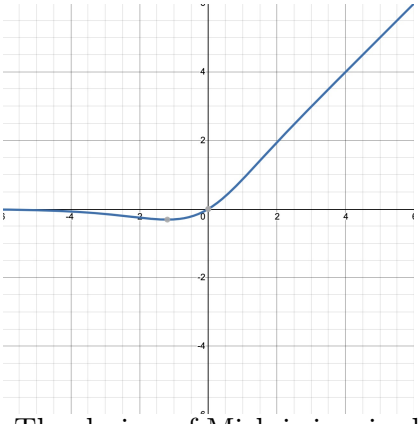
Swish is empirically superior to ReLU. From the definition of swish, we can see that it can also be written as $f(z) = z\sigma(\beta z)$, where σ is sigmoid. It is said to be a self-gated activation function because z itself is multiplied by a sigmoid that is parametrized by z. Sigmoid plays the role of controlling how much of z should be passed to the next layer since sigmoid $\in [0, 1]$.
The parameter $\beta \geq 0$ can be a hardcoded constant or a learnable variable as a in parametric ReLU. When $\beta = 0$, swish becomes a linear function, $f(z) = z$. When $\beta \to \infty$, swish becomes ReLU.
From the graph above, we can see that swish shares two common properties with ReLU.

(1) First, it is unbounded above (i.e. $\lim_{z\to\infty} f(z) = \infty$).
(2) Second, it is bounded below (i.e. $\lim_{z\to-\infty} f(z) = 0$).

Activation functions that are not bounded above have an edge over those that are, such as sigmoid, and can avoid the saturation problem in sigmoid that causes gradient vanishing. On the other hand, the boundedness below is desirable because of strong regularization effects [4]. When z are large negative values, activation functions such as ReLU and swish will *forget* about them and produce an output of zero. However, swish has an advantage over ReLU in that it is non-monotonic; it still produces small negative outputs when z is of small negative values. The preservation of small negative weights increases the expressivity of the model, whereas ReLU regularizes them to zero altogether.
Additionally, swish is smooth everywhere. Smoothness is a favorable property because it produces smooth outputs and in turn smooth loss landscape, which makes it easier to optimize. Reader is encouraged to read more in this paper [4].

Mish:

$$f(z) = z \tanh\left(\ln\left(1 + e^{z}\right)\right)$$



Mish is said to empirically outperform swish [5]. The design of Mish is inspired by investigating the advantageous properties that swish possesses. From the graph above, we can see that Mish holds all the good properties that makes swish successful, such as (1) unbounded above (2) bounded below (3) non-monotonicity (4) differentiable everywhere. We have to look into the first derivative of mish to understand the one possible reason that mish performs better than swish. However, note that it is a mere speculation even in the original paper [5].

$$\begin{aligned} f'(z) &= sech^2(\ln(1+e^z))\frac{z}{1+e^{-z}} + \frac{f(z)}{z} \\ &= \Delta(z) \times swish(z) + \frac{f(z)}{z} \end{aligned}$$

We can see that $f'(z)$ is closely related to swish. The $\Delta(z)$ plays a role of preconditioner that makes the gradient smoother. Preconditioning is used to modify the geometry of objective function to increase the rate of convergence. Recall while doing gradient descent, when the objective function is contracted in one direction, it causes the optimization path jagged and circuitous. Besides momentum, preconditioning can also be used to smoothen the objective function in this situation. The inverse of a symmetric positive definite matrix is often used as a preconditioner to quote unquote "*normalize*" the objective function so that it is proportional in all directions, and $\Delta(z)$ analogously plays the role of a precoditioner in mish. Reader is encouraged to read more in this paper [5].

3. Numerical Stability: Gradient Vanishing and Exploding

Although these activation units are powerful tools and give our neural networks impressive capability of representing various functions, we have to realize that there is no free lunch. The non-linearity can introduce quite complex numerical behaviors that are very hard to understand. In this section, we will talk about the most famous topic: gradient vanishing and exploding. We will analyze it as a numerical stability problem.

First, let us recall how we represent real numbers in computer programs. Usually, you cannot store each number with unlimited precision! We use floating-point numbers to do math in computer programs.

As a workaround to balance limited computation ability and ideal mathematics, such representations will definitely have many drawbacks. For example, extremely small/large values cannot be well-represented and can even lead to errors. Moreover, some operations like addition and multiplication between floating-point numbers cannot reflect the true math well in some cases.

Consider a neural network (MLP) with L layers in total, the input is $\mathbf{x} \in \mathbb{R}^{n^{(0)}}$, the output is $\mathbf{y} \in \mathbb{R}^{n^{(L)}}$, the activation function is $\sigma : \mathbb{R} \to \mathbb{R}$ (we use the same activation function for all hidden layers), and the weight for layer l is $\mathbf{W}^{(l)}$, where $l \in \{1, 2, \ldots, L\}$.

For forward propagation, we have:

$$\mathbf{y} = \sigma(\mathbf{W}^{(L)}\ \sigma(\mathbf{W}^{(L-1)}\ ...\ \sigma(\mathbf{W}^{(1)}\ \mathbf{x})...))$$

$$\mathbf{W}^{(l)} \in \mathbb{R}^{n^{(l)} \times n^{(l-1)}}$$

For convenience, we introduce some notations for intermediate variables. Let

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)},\ l > 0$$

$$\mathbf{a}^{(l)} = \begin{cases} \sigma(\mathbf{z}^{(l)}),\ l > 0 \\ \mathbf{x},\ l = 0 \end{cases}$$

Assume that the final loss is $\mathcal{L}$ and that we know the derivative of loss w.r.t the output $\mathbf{y}$, i.e. $\dfrac{\partial \mathcal{L}}{\partial \mathbf{y}^T}$. Then the back-propagation algorithm can be written as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)T}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}^T}$$

for $l \leftarrow L$ downto 1 do

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}} = \sigma'(\mathbf{z}^{(l)})\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l)T}},\ \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(l-1)T}} = \mathbf{W}^{(l)T}\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}},\ \nabla_{\mathbf{W}^{(l)}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}}\mathbf{a}^{(l-1)T}$$

Let

$$\begin{aligned}\delta^{(l)} &= \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)T}} \\ &= \text{diag}(\sigma'(\mathbf{z}^{(l)}))\mathbf{W}^{(l+1)T}\text{diag}(\sigma'(\mathbf{z}^{(l+1)}))\cdots\mathbf{W}^{(L)T}\text{diag}(\sigma'(\mathbf{z}^{(L)}))\frac{\partial \mathcal{L}}{\partial \mathbf{y}^T}\end{aligned}$$

Note that $\text{diag}(\sigma'(\mathbf{z}^{(l)}))$ is a diagonal matrix consists of the derivative of $\mathbf{a}^{(l)}$ w.r.t $\mathbf{z}^{(l)}$ (i.e., $\frac{\partial \sigma(\mathbf{a}^{(l)})}{\partial \mathbf{z}^{(l)}}$).
We can write the gradient of loss w.r.t the parameter $\mathbf{W}^{(l)}$, $\nabla_{\mathbf{W}^{(l)}}\mathcal{L}$ as:

$$\nabla_{\mathbf{W}^{(l)}}\mathcal{L} = \delta^{(l)}\mathbf{a}^{(l-1)T}$$

In summary, the gradient $\nabla_{\mathbf{W}^{(l)}}\mathcal{L}$ is the product of many matrices/vectors!!! For scalars, if you multiply many extremely small numbers, you will get 0 and if you multiply many extremely large numbers, you will get an inf, nan or even error. Such process is not invertible. So once you got trapped you can never get out. For these matrices and vectors, the same thing would happen!!!
Recall the gradient plots of common non-linear activation units. To keep away from gradient vanishing/exploding, you might want to keep the inputs of activation units is a suitable region. How to make it?

- Change $\mathbf{z}^{(l)}$ or equivalently $\mathbf{a}^{(l)}$
- Change $\mathbf{W}^{(l)}$
- Change σ'

References

[1] "Cs231n convolutional neural networks for visual recognition." [Online]. Available: https://cs231n.github.io/neural-networks-1/

[2] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, p. 303–314, Dec 1989.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[4] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: a self-gated activation function," *arXiv preprint arXiv:1710.05941*, vol. 7, 2017.

[5] D. Misra, "Mish: A self regularized non-monotonic neural activation function," *arXiv preprint arXiv:1908.08681*, 2019.