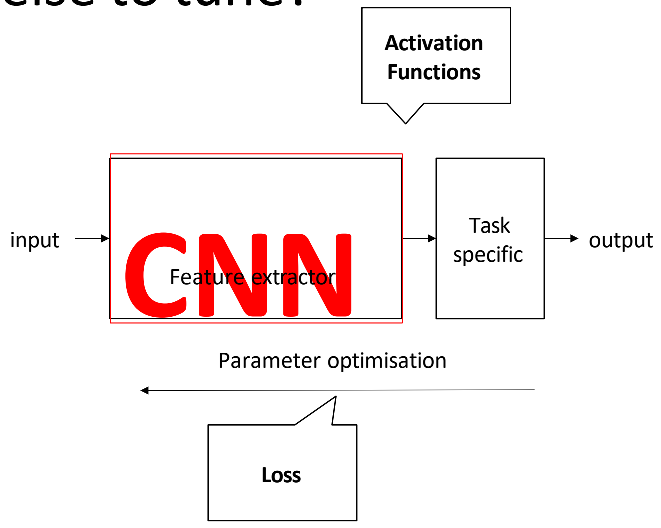
Deep Learning – Activation Functions

M.M Islam Imran

Let’s talk about activation functions

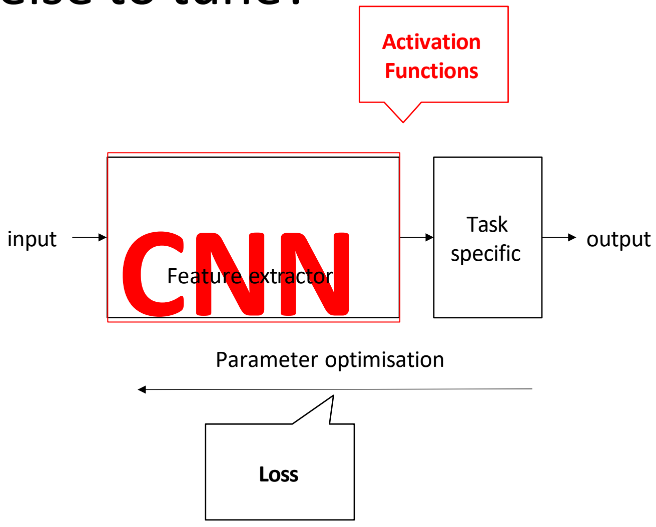
Where else to tune?



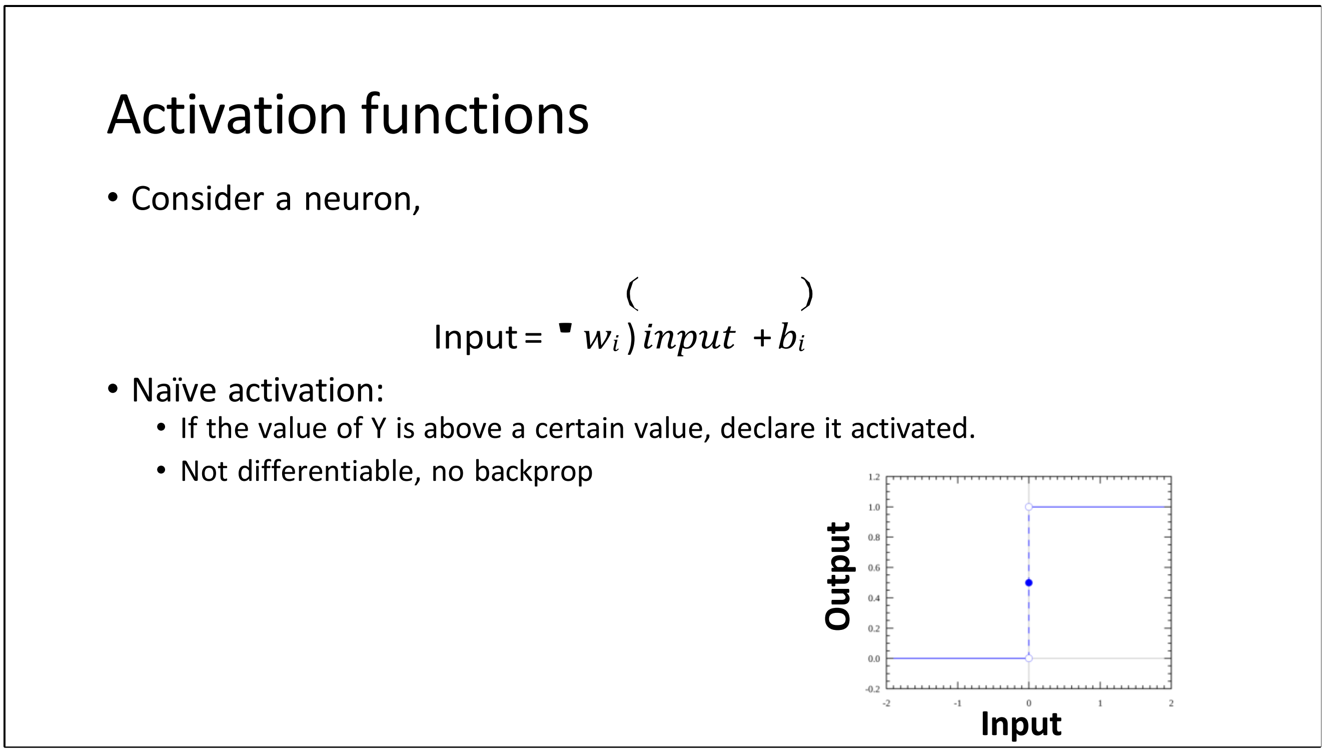
We will discuss another crucial aspect of Convolutional Neural Networks: Activation functions and loss functions.
These elements are essential as they can significantly impact the performance of your neural network model.

First, let's talk about activation functions.
They determine how neurons in the network respond -- or "activate” -- when they receive a set of inputs.
Activation functions introduce non-linear properties into the system, allowing the network to learn from complex data.
Next, we have the concept of Error or Loss functions.
These functions measure how well your network is doing, quantifying the difference between the predicted outputs and the actual ground truth.
Backpropagation uses this error measurement to update the model parameters, aiming to minimize this error.
Research in this area has been dynamic and extensive.
While activation functions were a primary focus in the earlier days of neural networks, more recent research has shifted towards optimizing loss functions.
This change in focus highlights the ongoing quest to fine-tune every aspect of these complex systems for better performance and efficiency.
Apart from tweaking the depth of the network, fine-tuning activation functions and loss functions can also yield significant improvements in your model.

Where else to tune?



Let's dive into the role of activation functions in neural networks.
Activation functions are mathematical formulas that dictate the output of a neuron given a certain input.
In essence, they act as the "gatekeepers" of each node, deciding how much signal should pass through to the next layer.
A key point to remember is that activation functions introduce non-linearity into the network.
This non-linearity is crucial because it allows the neural network to learn from complex and varied data.
Without non-linear activation functions, your neural network would essentially become a simple linear regression model, incapable of learning complex functions.
So, when designing or fine-tuning a neural network, choosing the right activation function can significantly impact the model's performance.
Different activation functions, like ReLU, Sigmoid, or Tanh, have their own advantages and disadvantages, which we will discuss in subsequent slides.
Remember, the choice of activation function can make or break your network's ability to learn effectively.

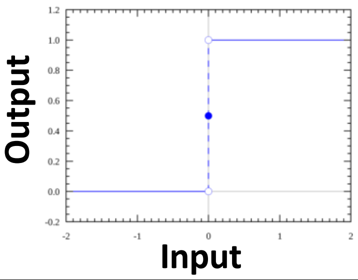


Activation functions

- Consider a neuron,

$$\text{Input} = \sum (w_i)\, input + b_i$$

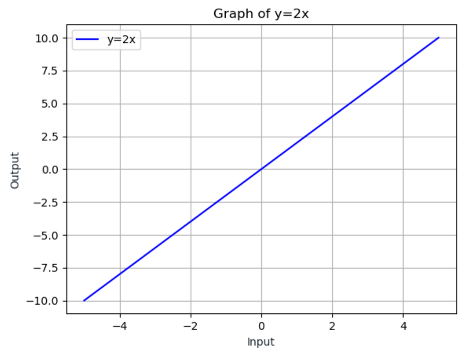
- Naïve activation:
  - If the value of Y is above a certain value, declare it activated.
  - Not differentiable, no backprop



First, let's remember the basic functionality of an artificial neuron.
Simply put, a neuron takes multiple inputs, applies weights to them, sums them up, adds a bias, and then decides whether to "activate" or not.
This operation is represented by a "weighted sum" equation, which is often denoted by Y = sum(weight_i * input_i) + bias.
Now, the resulting value of Y can range anywhere from negative infinity to positive infinity.
Given this wide range, how can we determine if a neuron should fire or not?
This is precisely where activation functions come into play.
Activation functions serve as a filter to decide the neuron's output.
A naive approach to activation might be to simply set a threshold: if Y is above a certain value, we declare the neuron activated.
However, this naive approach is not differentiable, which means we can't use backpropagation to adjust the weights and biases during the learning phase.
Therefore, we need smooth, differentiable activation functions like ReLU, Sigmoid, or Tanh, which not only decide the output but also make it possible to train the network effectively using gradient-based methods like backpropagation.

Linear activation

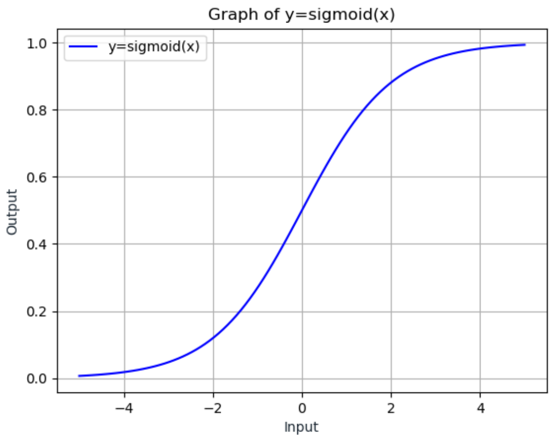
- $Output = c\ )\ x$
- Constant gradient, no relationship to x during backprop



Firstly, let's discuss what we seek to achieve with an activation function. We aim to move beyond binary "activated" or "not activated" outputs, and instead seek a more nuanced, continuous range of outputs. Linear activation functions offer one such solution, by providing activation values that are directly proportional to the input.
In the case of linear activations, the function can be represented simply as Output=c×x, where c is a constant. This produces a constant gradient, meaning that during backpropagation, the updates applied to weights are constant and independent of the change in input, denoted by Δx.
However, linear activation functions have limitations, especially when it comes to stacked, or multi-layered, neural networks. If each layer in a multi-layered network employs a linear activation function, the output of one layer becomes the input to the next, perpetuating linearity throughout the network. Ultimately, no matter how many layers you have, the entire network behaves like a single-layer linear model.
This means you could replace all N linear layers with just a single linear layer and achieve the same output. It renders the "depth" of the network irrelevant because it doesn't allow for the complexity needed to learn from more intricate forms of data. Therefore, while linear activation functions may have some use-cases, they aren't typically chosen for complex machine learning tasks that require the network to capture more complex, non-linear relationships in the data.

Sigmoid function

- $Output = \frac{1}{1+e^{!input}}$
- Nonlinear



The sigmoid function is one of the earliest and simplest non-linear activation functions used in neural networks. Mathematically, it's expressed as $Output = \frac{1}{1+e^{!input}}$ This function is non-linear, allowing us to stack layers in a neural network, thereby facilitating the learning of more complex representations. Moreover, unlike step functions which are binary, the sigmoid function gives a more analog or continuous output, ranging from 0 to 1.

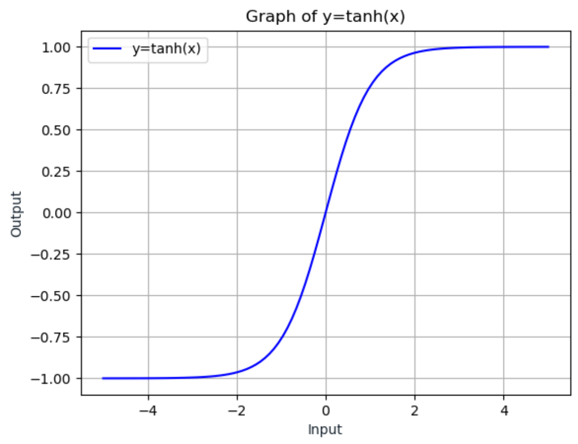
The sigmoid function has an S-shaped curve, and its gradient is smooth, which is crucial for gradient descent algorithms. One notable characteristic is that between the X values of -2 and 2, the curve is especially steep. This implies that small changes in the input within this region result in significant shifts in output, facilitating rapid learning during the training phase.

However, the sigmoid function is not without its drawbacks. Towards the tails of the function, the curve flattens out, and the output values become less sensitive to changes in input. This results in a vanishing gradient problem, where gradients become too small for the network to learn effectively, leading to slow or stalled training.

So, while sigmoid functions were seminal in the early development of neural networks, these limitations have led researchers to explore alternative activation functions that can mitigate these issues.

tanh function

- $Output = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Scaled sigmoid



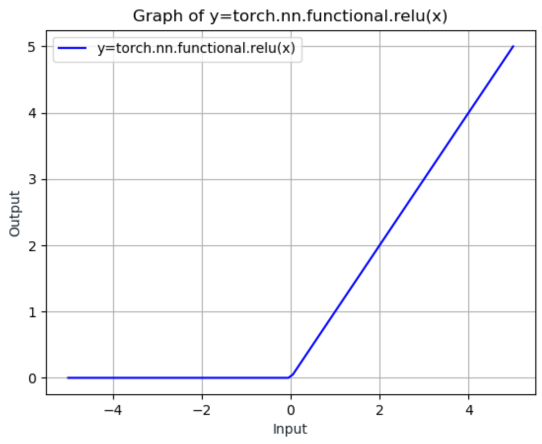
The tanh function is another non-linear activation function, mathematically defined as $Output = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

It's essentially a scaled version of the sigmoid function, but ranges from -1 to 1, instead of 0 to 1.
Like the sigmoid, the tanh function is also non-linear, meaning we can stack multiple layers of neurons using this activation function.
Because its output range is between -1 and 1, there is less concern about activations becoming too large and dominating the learning process.
One key benefit of tanh over sigmoid is that its gradient is stronger; that is, the derivatives are steeper.
This can make it a better choice for certain problems where faster convergence is desired.
Another advantage of tanh is that its outputs are zero-centered, meaning the average output is close to zero.
This is beneficial for the learning process of subsequent layers, as it tends to speed up convergence by allowing for a balanced distribution of outputs and gradients.
However, like the sigmoid function, tanh also suffers from the vanishing gradient problem when you stack many layers, which can slow down learning.
Careful normalization of the inputs is also essential when using tanh to ensure effective

learning.
Choosing between sigmoid and tanh will depend on your specific requirements, particularly regarding the strength of the gradient and the range of the activation function.

ReLU

- $Output = \max(0, x)$
- Efficient
- combinations of ReLU and ReL are non linear
- Bound: [0, inf)
- dying ReLu problem

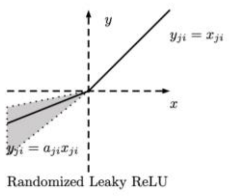


ReLU, or Rectified Linear Unit, is defined as $Output = \max(0, x)$
It's a piecewise linear function that outputs the input directly if it's positive, otherwise, it outputs zero.
At first glance, ReLU might seem linear, especially since it is linear over the positive axis.
However, ReLU is inherently non-linear when considered as a whole, particularly due to the sharp corner at the origin.
Interestingly, combinations of ReLU functions are also non-linear, enabling us to stack layers in neural networks effectively.
This is because ReLU is a universal approximator, meaning that it can represent a wide variety of functions when used in a neural network.
An important note about ReLU is that it is unbounded, meaning its range is [0,∞).
While this can be useful for certain types of data, it could also cause the activations to explode if not managed properly.
Another significant feature of ReLU is that it tends to produce sparse activations.
In a neural network with many neurons, using activation functions like sigmoid or tanh would cause almost all neurons to activate to some degree, leading to dense activations.
ReLU, on the other hand, will often output zero, effectively ignoring some neurons, which can make the network more computationally efficient.
However, this sparsity leads to a known issue called the "Dying ReLU" problem.
If a neuron's output is always zero (perhaps due to poor initialization), the gradient for that neuron will also be zero.

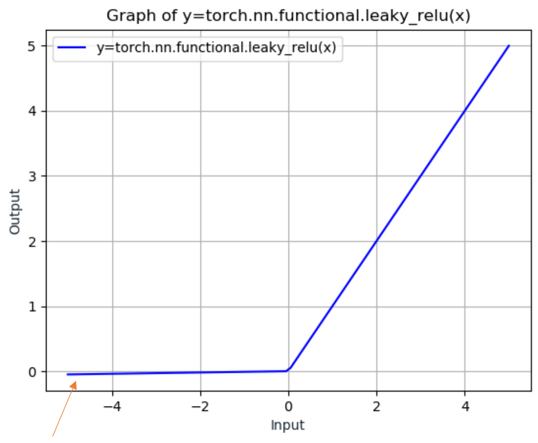
As a result, during backpropagation, the weights of that neuron remain unchanged, effectively "killing" the neuron.
This can result in a portion of the neural network becoming inactive, thereby limiting its capacity to model complex functions.
Despite this drawback, ReLU remains one of the most widely used activation functions due to its efficiency and effectiveness in many applications.

Leaky ReLU

- $Output = \begin{cases} x & for\ x \ge 0 \\ e.g. 0.01)\, x & for\ x < 0 \end{cases}$
- Mitigates dying ReLu



atcold.github.io

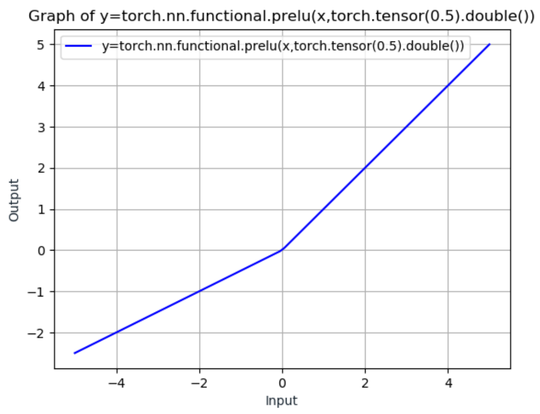


Leaky ReLU is a modified version of the ReLU function, defined as $Output = \begin{cases} x & for\ x \ge 0, \\ e.g. 0.01)x & for\ x < 0 \end{cases}$ for a small constant, typically 0.01.
It's designed to address the "Dying ReLU" problem.
In the standard ReLU function, the gradient becomes zero for negative inputs, causing neurons to "die" during training.
Leaky ReLU attempts to solve this by introducing a small slope for negative values, typically 0.01, to ensure the gradient is non-zero.
This small slope allows "dead" neurons to reactivate during the course of training.
In other words, it provides a pathway for gradients to flow, even when the neuron is not active.
Like traditional ReLU, Leaky ReLU is computationally efficient.
Simple mathematical operations are involved, making it less computationally expensive than tanh and sigmoid, which is an advantage in the design of deep networks.

Leaky ReLU maintains many of the benefits of the original ReLU function, such as sparsity and the ability to approximate a wide range of functions.
However, it adds the benefit of mitigating the risk of dead neurons, making it a valuable alternative in many contexts.

PReLU

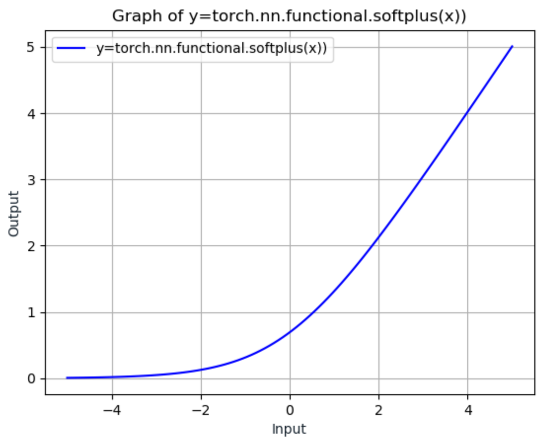
- $Output = \begin{cases} x & for\ x \geq 0 \\ a)x & for\ x < 0 \end{cases}$
- a is learnable
- Either one or a separate a is used for each input channel



PReLU stands for Parametric ReLU, and it's an extension of the Leaky ReLU function. In PReLU, the negative slope a becomes a learnable parameter, meaning it's adjusted during the training process.
This added flexibility allows PReLU to adapt during training, potentially leading to better performance than Leaky ReLU in some scenarios. In PyTorch, when PReLU is invoked without arguments, a single learnable parameter a is used across all input channels. Alternatively, you can specify the number of channels as an input argument, and PyTorch will use a separate a for each input channel. This per-channel adaptability can provide more nuanced behavior during the training process.
An interesting feature of ReLU variants, including PReLU, is scale invariance. That is, if you multiply the input by a scalar, the shape of the output remains the same, just scaled. In the context of CNN architectures, where scale invariance can be valuable, PReLU and its variants can be especially useful. By making the negative slope learnable, PReLU adds another layer of adaptability, potentially making it a strong choice for certain types of neural network architectures.

SoftPlus

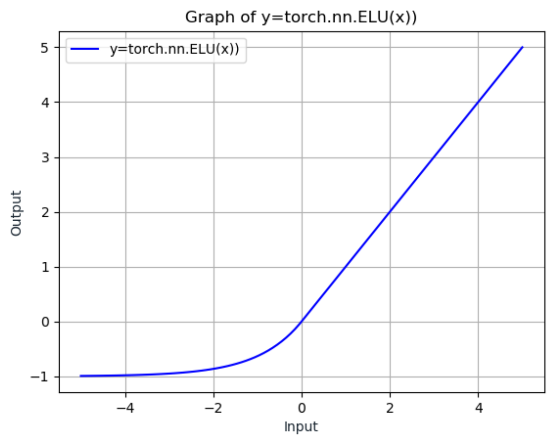
- $Output = \frac{1}{\beta} \log\left(1 + e^{\beta \cdot x}\right)$
- Smooth approximation of ReLU
- Output always positive
- Numerical stability: use linear if $(\beta) x > threshold$



The SoftPlus function serves as a smooth and differentiable approximation to the well-known ReLU function. Because of its smoothness, SoftPlus is easier to differentiate, making it potentially advantageous during the optimization process.
SoftPlus is parameterized by a scale factor β, which controls how closely the function approximates ReLU. The higher the value of β, the closer SoftPlus mimics ReLU's behavior.
A notable feature of SoftPlus is that it outputs only positive values. This makes it suitable for layers where you specifically require positive activations.
However, SoftPlus isn't perfect; it has numerical stability issues for large input values. To handle this, the PyTorch implementation switches to a linear function when the condition β×x exceeds a predefined threshold.
One important thing to note is that SoftPlus, like ReLU and its variants, is non-linear across its entire domain. This non-linearity is crucial for enabling the neural network to capture complex relationships in the data.
SoftPlus is sensitive to the amplitude of the input signal, which means that it's non-linear regardless of the input size. That's beneficial for models where amplitude variation is a significant feature.
SoftPlus offers a smoother, differentiable alternative to ReLU and is particularly useful when you want positive activations and better numerical stability.

ELU

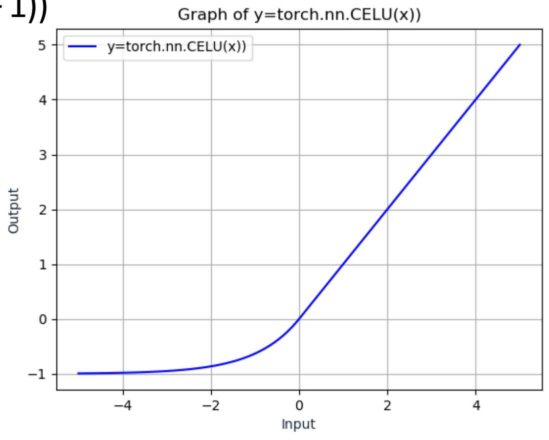
- $Output = \max(0, x) + \min(0, \alpha(e^x - 1))$
- Element-wise



The Exponential Linear Unit, or ELU, is a fascinating variation of the ReLU function. It's designed to be element-wise, operating on each element of the input independently. One thing that sets ELU apart is its ability to output negative values. Unlike ReLU, which only outputs positive values, ELU can go below zero.
ELU is parameterized by α, which scales the exponential function for negative inputs. When x<0, the function becomes α×(ex−1), making it smooth and differentiable across the negative domain.
Being able to output negative values allows ELU to push the mean activation closer to zero. A zero-centered mean can help the network converge faster, a useful property in deep learning models.
Like SoftPlus, ELU is also a soft, smooth version of the ReLU function. It combines the benefits of both positive and negative output capabilities, making it more versatile depending on the specific application you have in mind.
The choice of the parameter α can be crucial. Different values of α will influence how closely ELU mimics ReLU for negative values, affecting the function's smoothness and the range of its output.
ELU is a strong candidate for scenarios where you want a balance of smoothness, differentiability, and the ability to have a mean activation around zero. It offers a unique blend of features that can be tailored to specific applications for potentially better performance.

CELU

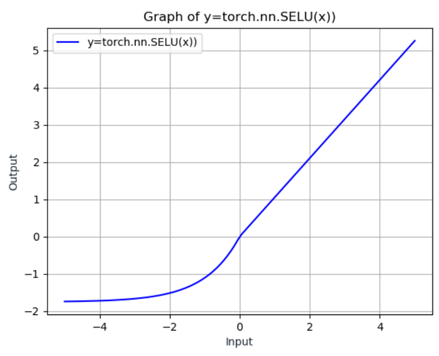
- $Output = \max(0, x) + \min(0, \alpha\,(e^{\frac{x}{\alpha}} - 1))$
- Element-wise
- Barron (2017) https://arxiv.org/abs/1704.07483



Let's move on to another variant of the ELU function, known as the CELU, or Continuously Differentiable Exponential Linear Units. It was introduced by Jonathan T. Barron in 2017, and you can read more about it in the paper linked here: Barron 2017. Like ELU, CELU is also an element-wise function, making it computationally efficient to apply across large tensors. What sets CELU apart is its special emphasis on continuous differentiability, denoted as C1 continuity.
The CELU function uses the parameter α, which is similar to the ELU function, but with a twist. In CELU, α doesn't just scale the exponential function for negative values, it also scales the input x inside the exponential term.
By ensuring that α is not equal to one, the CELU function becomes continuously differentiable across its entire domain. C1 continuity is often desirable in optimization tasks as it ensures a smooth gradient, facilitating more efficient backpropagation.
Another noteworthy feature is that like ELU, CELU can also produce negative values. This enables it to center the mean activation towards zero, which in turn, could speed up the convergence of deep learning models.
CELU offers a nuanced balance of features, with continuous differentiability being its most distinct characteristic. It builds upon the strengths of ELU, adding more mathematical rigor to suit certain applications and optimization scenarios.

SELU

- $Output = scale\,(\max(0, x) + \min(0, \alpha(e^x - 1)))$
  - with α = 1.6732632423543772848170429916717 and scale = 1.0507009873554804934193349852946

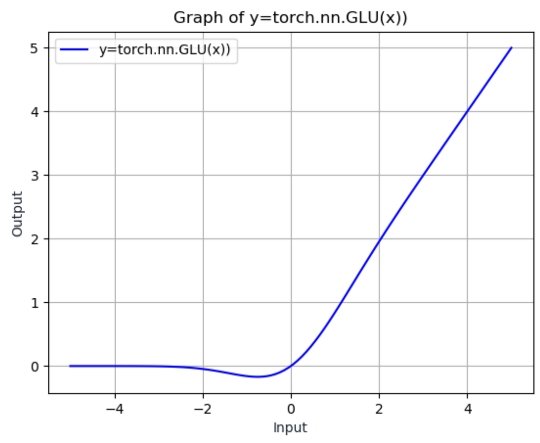


Let's explore another intriguing activation function, the SELU or Scaled Exponential Linear Units. SELU comes with pre-defined parameters α and scale, which have been meticulously optimized.
Unlike other activation functions, the magic of SELU lies in its ability to perform internal normalization. These specific constants -- α and scale -- are solutions to a fixed-point equation designed to maintain a mean of 0 and a variance of 1 across layers.
Normalizing activations in a neural network can happen at three levels. The first is input normalization, where we scale input features, like grayscale pixel values, into a specific range, such as 0 to 1. The second level is batch normalization, a technique specifically designed for neural networks to stabilize the learning process.
SELU shines at the third level, which is internal normalization. The design of SELU ensures that the mean and variance of the activations are preserved from one layer to the next.
To achieve this normalization, the function needs to produce both positive and negative outputs, which allows it to shift the mean towards zero. Interestingly, the very characteristic that causes vanishing gradients in other activation functions -- gradients close to zero -- is actually beneficial in SELU for internal normalization.
And one more thing: SELUs never die. Due to its design, SELU avoids the "dying unit" problem that plagues some other activation functions, making it a robust choice for certain types of networks.
If you're looking to build deep networks without worrying too much about manual

normalization techniques, SELU offers an exciting pathway. It's specifically designed to keep the internal statistics of your network stable, which can lead to faster and more reliable training.

GELU

- $Output = x\,)\,\Phi(x)$
- $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution.



Let's dive into another interesting activation function: GELU or Gaussian Error Linear Unit.

GELU's output is defined as x multiplied by the cumulative distribution function (CDF) of the Gaussian distribution, denoted as Φ(x).

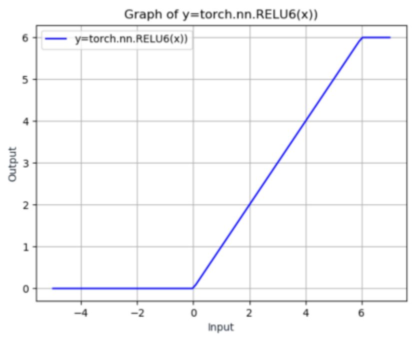
This activation function draws inspiration from the Gaussian distribution, which is a fundamental concept in statistics. The cumulative distribution function, Φ(x), gives us the probability that a normally distributed random variable takes a value less than or equal to x.

By incorporating the Gaussian distribution's CDF, GELU introduces a probabilistic flavor to the activation process. This feature can have implications for the regularization of neural networks, potentially providing a beneficial influence on network training and performance.

In essence, GELU adds an interesting twist to activation functions by bringing in concepts from probability theory, which can have unique effects on the behaviour of the neural network during training and inference.

ReLU6

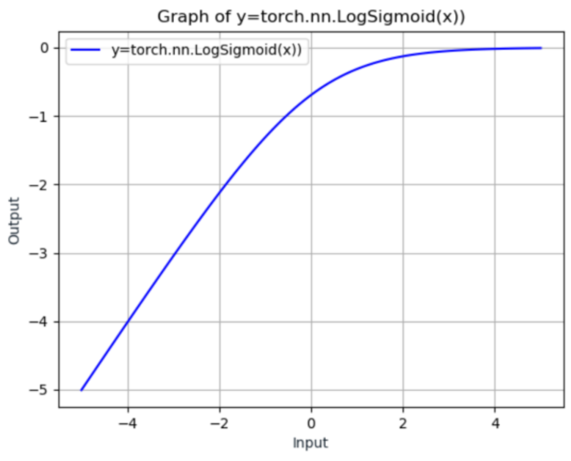
- $Output = \min(\max(0, x), 6)$



Now let's explore the ReLU6 activation function, which is a variation of the standard ReLU.
ReLU6's output is defined as the minimum of the maximum between 0 and x and the value 6. In simpler terms, if the input x is positive, it passes through unchanged up to a maximum value of 6. If x is negative, it's simply truncated to 0.
One interesting aspect of ReLU6 is that it can be seen as a way to saturate the activations. Saturating means that the activations are limited to a certain range, which can be useful in preventing activations from growing excessively and causing numerical instability.
The value 6 in ReLU6's definition might seem somewhat arbitrary, but it's actually a parameter that can be adjusted to achieve different levels of saturation. This flexibility allows for experimentation with various configurations.
Visually, ReLU6 may remind you of other activation functions like the hard sigmoid or the tanh function. This similarity in appearance doesn't necessarily imply identical behaviour, but it does offer a familiar visual analogy.
ReLU6 presents an alternative way to introduce saturation to the ReLU activation, offering potential benefits in preventing runaway activations while allowing some degree of fine-tuning through the parameterization.

LogSigmoid

- $Output = \log(\frac{1}{1+e^{-x}})$
- Element-wise

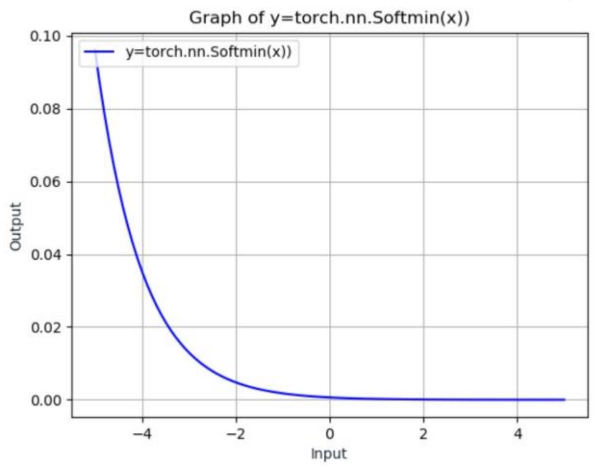


LogSigmoid's output is calculated as the natural logarithm of x: $Output = \log(\frac{1}{1+e^{-x}})$

This function operates element-wise, meaning it's applied separately to each element of the input tensor.
Unlike many other activation functions, LogSigmoid is predominantly used in the context of cost functions rather than as a primary activation function in neural network layers.
The main utility of LogSigmoid lies in its role within loss functions. Loss functions are crucial components in training neural networks as they quantify the difference between predicted values and actual target values. LogSigmoid's characteristics make it particularly suitable for certain types of loss functions.
LogSigmoid's appearance in loss functions serves specific purposes that contribute to effective training. We'll explore later on how this function operates in the context of loss optimization.
While LogSigmoid may not be a common choice for standard activation functions in neural network layers, its presence is significant in the realm of loss functions, where it contributes to the optimization process during training.

Softmin

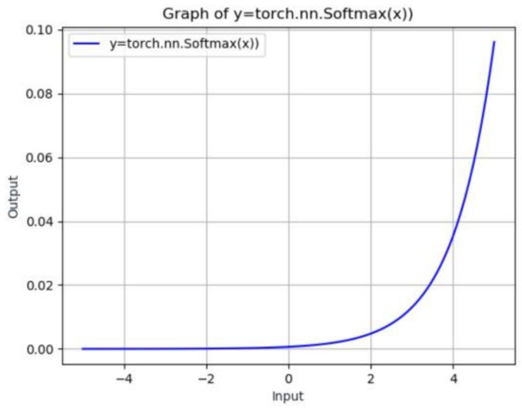
- $Output = \frac{e^{!x_i}}{\sum_{)} e^{!x_)}}$
- Applies the Softmin function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



The Softmin function operates on an n-dimensional input tensor, transforming the elements in a specific way. It's formulated as the exponential of the negative of each element divided by the sum of the exponentials of all elements in the tensor.
The primary outcome of applying the Softmin function is a rescaling of the n-dimensional input tensor's elements. This rescaling ensures that the elements of the resulting n-dimensional output tensor fall within the range of [0, 1] and collectively sum up to 1. Essentially, this function transforms the inputs into a probability-like distribution.
Softmin introduces multi-dimensional non-linearities to the neural network. It's a transformation from a vector in to a vector out. In the context of neural networks, these "energies" or "penalties" can be conceptualized as a way to model various aspects of data.
One way to perceive Softmin is as a method to convert a set of numbers into a representation that resembles a probability distribution. This characteristic makes it valuable in scenarios where you want to assign relative weights or preferences among multiple alternatives.
Softmin's role lies in rescaling and transforming input tensors into probability-like distributions, contributing to the multi-dimensional non-linearities of neural networks and enabling the modelling of various factors in the data.

Softmax

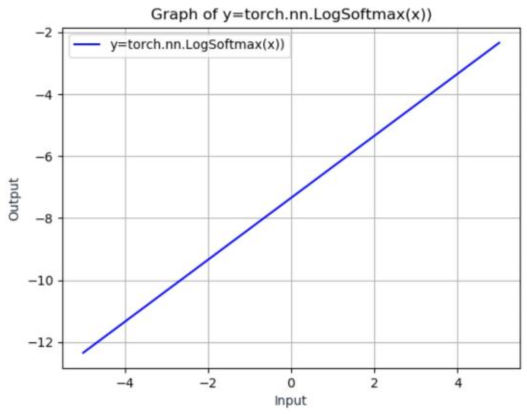
- $Output = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



Now, let's delve into one of the most common activation functions in deep learning: the Softmax function. This function plays a vital role in various aspects of neural networks.
The Softmax function is designed to operate on an n-dimensional input tensor. It transforms the individual elements of the tensor using the exponential of each element divided by the sum of the exponentials of all elements within the tensor.
The primary outcome of applying the Softmax function is the rescaling of the n-dimensional input tensor's elements. This transformation ensures that the resulting n-dimensional output tensor's elements lie within the range of [0, 1] and, crucially, add up to a sum of 1. This property makes the Softmax function particularly useful when working with probability distributions.
The Softmax function's significance lies in its widespread application. One of its primary use cases is to convert the raw scores or logits generated by a neural network into meaningful class probability scores. These probability scores represent the likelihood of each input belonging to a specific class within a multi-class classification scenario.
For instance, when dealing with image classification, the Softmax activation function is commonly employed to transform the network's logits into probabilities. These probabilities can then be used to make decisions about the most likely class for a given input.
The Softmax activation function is a cornerstone in deep learning, responsible for transforming logits into class probabilities. Its ability to rescale and normalize these scores makes it an essential tool for various tasks involving probability distributions and multi-class classification.

LogSoftmax

- $Output = \log\left(\frac{e^{x_i}}{\sum_j e^{x_j}}\right)$
- Applies the log(Softmax) function to an n-dimensional input Tensor



The LogSoftmax function operates similarly to the Softmax function, but with a significant difference: it applies the natural logarithm to the values obtained from the Softmax transformation. Just like Softmax, LogSoftmax also operates on an n-dimensional input tensor. The fundamental purpose of LogSoftmax is to compute the logarithm of the normalized exponentials of the input tensor's elements. This logarithmic transformation can offer benefits in certain contexts, especially when dealing with probabilities and handling numerical stability.
LogSoftmax is particularly useful when constructing loss functions for neural networks. The logarithmic transformation provides a way to manipulate the Softmax probabilities to better align with the structure of certain loss functions. By utilizing LogSoftmax in a loss function, it's possible to simplify mathematical calculations and potentially improve the training process.
It's important to note that LogSoftmax isn't typically used as an activation function in the output layers of neural networks, as its output values are not directly interpretable as class probabilities. Instead, it often plays a role in loss functions, helping to define the optimization process.
The LogSoftmax activation function is an extension of the Softmax function that applies a logarithmic transformation to the normalized exponentials of input tensor elements. Its primary application lies in constructing loss functions, where the logarithmic properties can be advantageous for numerical stability and simplifying mathematical operations.

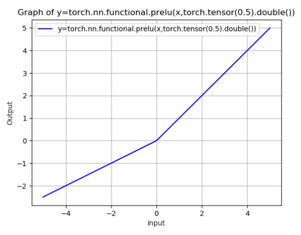
Periodic activations, SIREN

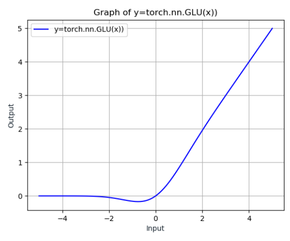
- Difficult convergence properties for general problems
- Has been used for implicit representations (find a continuous function that represents sparse input data, e.g. an image)
- https://vsitzmann.github.io/siren/

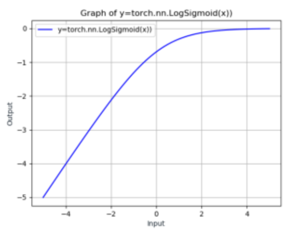
$$\Phi(\mathbf{x}) = \mathbf{W}_n (\phi_{n-1} \circ \phi_{n-2} \circ \ldots \circ \phi_0)(\mathbf{x}) + \mathbf{b}_n, \quad \mathbf{x}_i \mapsto \phi_i(\mathbf{x}_i) = \sin(\mathbf{W}_i \mathbf{x}_i + \mathbf{b}_i).$$

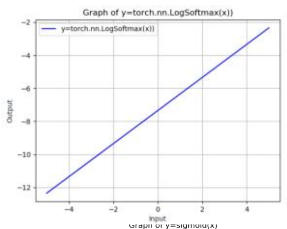
SIREN (Sinusoidal Representation Network) activation. This type of activation function presents some unique characteristics and applications within deep learning, but also comes with certain challenges.
Periodic activations like SIREN have been developed to tackle specific problems and use cases. However, they can be more challenging to work with compared to traditional activation functions due to their specialized nature.
One of the primary applications of SIREN is in dealing with implicit representations. Implicit representations involve finding a continuous function that represents sparse input data, such as images. This can be particularly useful for tasks where conventional approaches might not be sufficient.
SIREN introduces a unique architectural approach by leveraging sinusoidal activation functions. These functions are specifically designed for representing complex natural signals and their derivatives. This is a departure from traditional network architectures that may struggle to capture fine details and spatial/temporal derivatives of signals defined implicitly.
A key insight is that SIREN activation functions are well-suited for representing a wide range of signals, including images, wavefields, videos, and sounds, along with their derivatives. The architecture enables the representation of intricate details that are essential for many physical signals defined as solutions to partial differential equations.
To improve the utilization of SIREN, the authors propose an analysis of activation statistics that leads to a principled initialization strategy. This enhances the training and

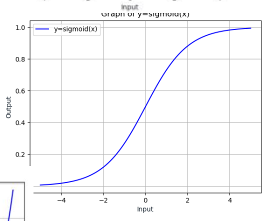
convergence properties of SIREN-based networks.
Furthermore, SIRENs can address challenging boundary value problems, such as Eikonal equations, the Poisson equation, and the Helmholtz and wave equations. This highlights the versatility and potential of SIRENs in solving complex and diverse tasks.
SIREN activation functions are a specialized type of periodic activation that has found significant utility in handling implicit representations and solving complex problems involving signals and their derivatives. While they bring unique capabilities, they also require careful considerations due to their distinct convergence properties and architecture. For more detailed insights, you can refer to the provided link to the SIREN research paper.

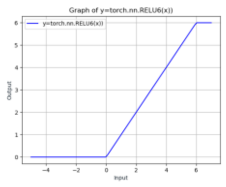


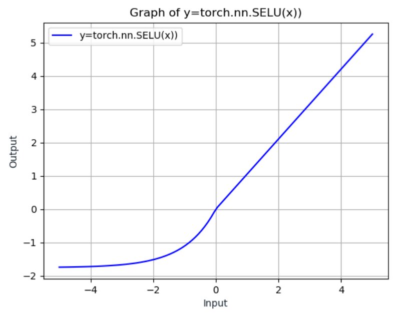


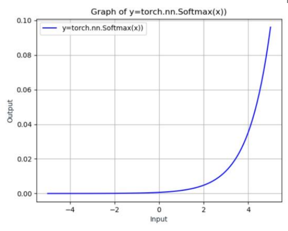












Now comes the critical question: which activation functions should we use in our neural networks? The answer, surprisingly, is not as straightforward as we might hope. It's not a one-size-fits-all scenario.

So, do we simply opt for ReLU in all cases, or perhaps sigmoid or tanh? The answer is both yes and no, depending on the specific context.
The choice of an activation function depends on the characteristics of the function you're aiming to approximate. If you have insights into the nature of the function, you can strategically select an activation function that aligns with those characteristics. This can significantly speed up the training process by allowing the network to approximate the desired function more efficiently.
For instance, let's take the sigmoid function. Its curve seems to possess properties ideal for a classifier. Choosing sigmoid as an activation function for a classifier can facilitate easier function approximation using combinations of sigmoid activations. Similarly, the choice of activation can impact the speed of convergence during training.
It's also worth noting that you're not limited to predefined activation functions. You can design and use custom activation functions tailored to your problem domain.
However, when you lack specific insights into the nature of the function you're trying to learn, starting with ReLU is often a practical approach. ReLU tends to work effectively as a general approximator and is widely used in many architectures.
In essence, the choice of activation function is a balancing act between the

characteristics of the target function and the efficiency of training.
While there's no universal answer, understanding the properties of different activation functions and how they interact with your problem can guide your decision-making process. It's a dynamic aspect of designing neural networks that combines intuition, experimentation, and informed decision-making.

What do we learn from this?

- Which function to use depends on the nature of the targeted problem.
- Most often you will be fine with ReLUs for classification problems. If the network does not converge, use leakyReLUs or PReLUs, etc.
- Tanh is quite ok for regression and continuous reconstruction problems.
- The representative power of you training set will usually outweigh the contribution of a smartly chosen activation function.

Activation functions play a critical role in neural networks, influencing both their training and performance. Here's an analysis of the advantages and disadvantages of using activation functions:

Advantages:

1. Non-linearity: Activation functions introduce non-linearity into the neural network, enabling it to learn complex patterns and relationships in data. Without non-linear activation functions, neural networks would only be able to model linear relationships, severely limiting their expressive power.
2. Gradient Propagation: Activation functions help in gradient propagation during the backpropagation algorithm, which is crucial for training deep neural networks. By providing non-zero gradients, activation functions allow the network to update its parameters effectively, leading to convergence and improved performance.
3. Squashing Inputs: Many activation functions, such as sigmoid and tanh, squash the input values into a bounded range (e.g., between 0 and 1 for sigmoid), preventing activations from growing too large or too small. This helps in stabilizing the training process and mitigating issues like vanishing and exploding gradients.
4. Sparse Activation: Some activation functions, like ReLU and its variants, promote sparse activation in the network by setting negative inputs to zero. Sparse activation can lead to more efficient computation and memory usage, as well as improved generalization by encouraging the network to focus on relevant features.
5. Versatility: There exists a variety of activation functions, each with its own characteristics and suitability for different types of problems. This versatility allows researchers and practitioners to experiment with different architectures and choose the activation function that best fits the specific requirements of their task.

Disadvantages:

1. Vanishing and Exploding Gradients: Certain activation functions, such as sigmoid and tanh, are prone to vanishing gradients, especially in deep networks. This can hinder the training process, as gradients become too small to propagate effectively through the network layers. Conversely, activation functions like ReLU can lead to exploding gradients in certain situations, causing unstable training.
2. Non-zero Centeredness: Activation functions like ReLU are not zero-centered, meaning they produce positive outputs for all non-negative inputs. This can lead to issues during optimization, as gradients may consistently push weights in one direction, slowing down convergence or causing oscillations in training.
3. Saturation: Some activation functions, such as sigmoid and tanh, saturate for large positive or negative inputs, resulting in flattened gradients and slower learning. This saturation can also cause the problem of "dead neurons," where neurons become non-responsive and stop learning altogether.
4. Limited Range: Certain activation functions have limited output ranges, which might not be suitable for certain types of data or architectures. For example, sigmoid functions

produce outputs in the range (0, 1), which may not be ideal for tasks where the target values lie outside this range.

5. Computational Complexity: Some activation functions, particularly those with complex mathematical formulations (e.g., sigmoid, tanh), can be computationally expensive to compute compared to simpler functions like ReLU. This increased complexity can impact training speed and inference time, especially in large-scale models.

In conclusion, while activation functions are indispensable in neural networks for introducing non-linearity and enabling effective training, the choice of activation function should be made carefully, considering the specific characteristics of the data and the architecture of the network. It's essential to strike a balance between the advantages and disadvantages of different activation functions to ensure optimal performance and stability in neural network training.

The choice of activation function in a neural network can significantly impact the gradient descent optimization process, particularly concerning the problem of vanishing gradients.

1. **Impact on Gradient Descent**:
   - Activation functions determine the non-linearity of the network, which is crucial for capturing complex patterns in data.
   - During backpropagation, the derivative of the activation function affects how gradients are propagated backward through the network layers.
   - The gradient of the loss function with respect to the activation of each neuron guides the weight updates in gradient descent.
   - Activation functions with well-behaved derivatives enable stable gradient propagation, leading to faster convergence during training.
2. **Problem of Vanishing Gradient**:
   - The vanishing gradient problem occurs when gradients become extremely small as they are backpropagated through many layers of the network.
   - Activation functions with derivatives that tend to zero (e.g., sigmoid, tanh) can exacerbate this problem, especially in deep networks.
   - As gradients approach zero, weight updates become negligible, impeding the learning process in deeper layers of the network.
   - The vanishing gradient problem often leads to slow convergence or stagnation during training, particularly in networks with many layers.
3. **Impact of Activation Functions on Vanishing Gradient**:
   - Activation functions like sigmoid and tanh are prone to the vanishing gradient problem due to their saturation properties.
   - In these functions, gradients become small for large positive or negative inputs, limiting the flow of information during backpropagation.
   - As a result, deeper layers in the network receive negligible gradient signals, hindering their ability to learn meaningful representations.
   - ReLU and its variants (e.g., Leaky ReLU, ELU) have been proposed as alternatives to mitigate the vanishing gradient problem.
   - ReLU has a simple derivative that is either 0 or 1, promoting more stable gradient flow in deep networks and mitigating the vanishing gradient problem to some extent.

In summary, the choice of activation function directly affects the behavior of gradient descent optimization and can influence the occurrence of the vanishing gradient problem. While activation functions like sigmoid and tanh are susceptible to vanishing gradients, ReLU-based functions offer better gradient flow in deep networks, alleviating this issue to some extent. However, selecting the appropriate activation function should consider various factors, including network architecture, training data characteristics, and the presence of vanishing gradient issues.