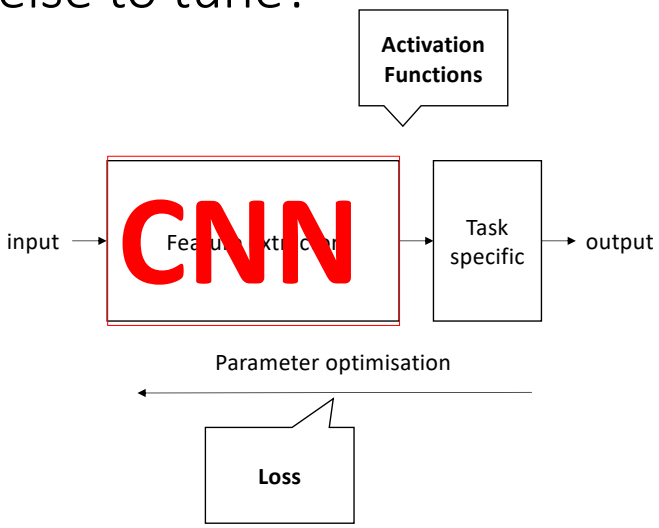
Deep Learning – Activation Functions

Bernhard Kainz

Let’s talk about activation functions

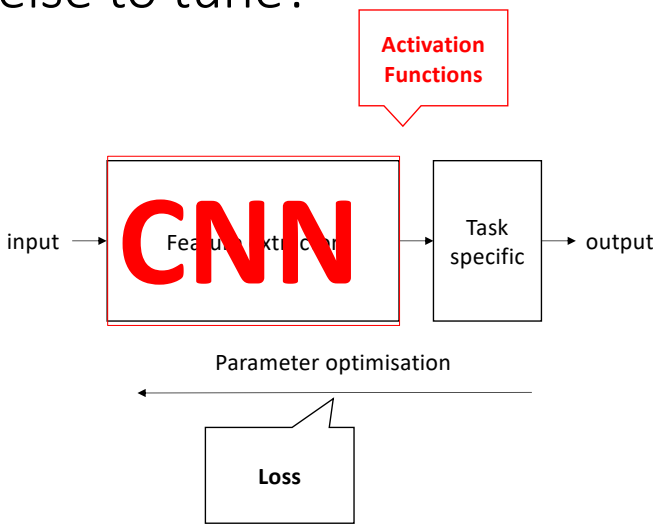
Where else to tune?



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?



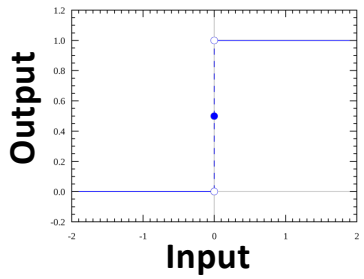
Let's dive into the role of activation functions in neural networks.
Activation functions are mathematical formulas that dictate the output of a neuron given a certain input.
In essence, they act as the "gatekeepers" of each node, deciding how much signal should pass through to the next layer.
A key 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,

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

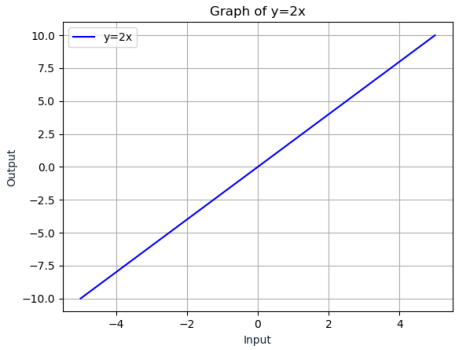
- Naïve activation:
  - If the value of Y is above a certain value, declare it activated.
  - Not differentiable, no backprop



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

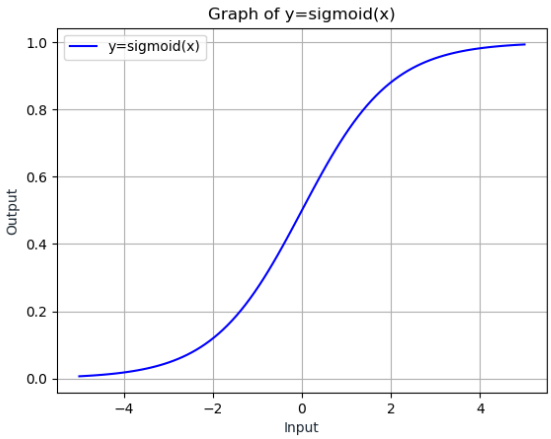
- $Output = c \cdot x$
- Constant gradient, no relationship to x during backprop



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

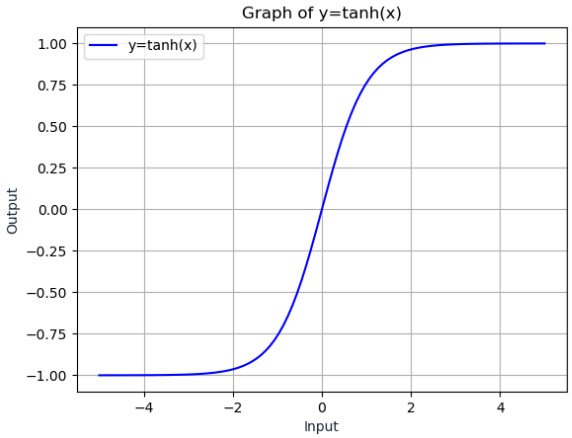
- $Output = \frac{1}{1+e^{-input}}$
- Nonlinear



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



Deep Learning – Bernhard Kainz

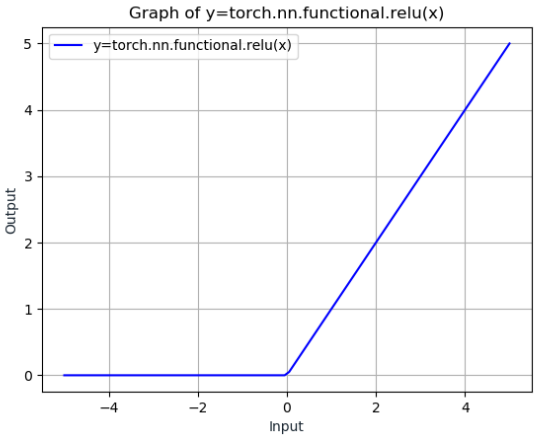
The tanh function is another non-linear activation function, mathematically defined as $Output = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

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 ReLU are non linear
- Bound: [0, inf)
- dying ReLu problem



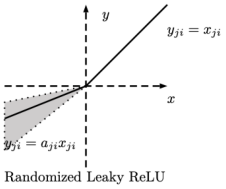
Deep Learning – Bernhard Kainz

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.

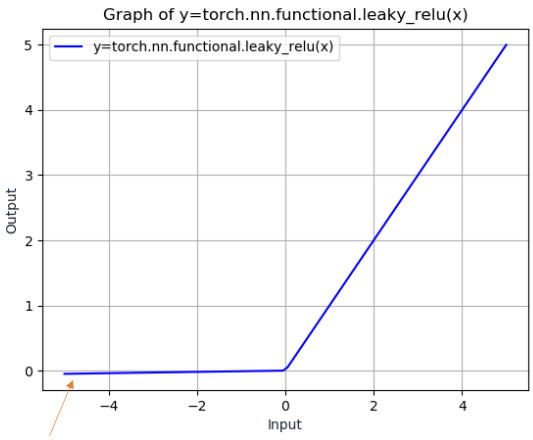
As a result, during backpropagation, the weights of that neuron remain unchanged, effectively "killing" the neuron.
This can result in a portion of the neural network becoming inactive, thereby limiting its capacity to model complex functions.
Despite this drawback, ReLU remains one of the most widely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

Leaky ReLU

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



Randomized Leaky ReLU



atcold.github.io

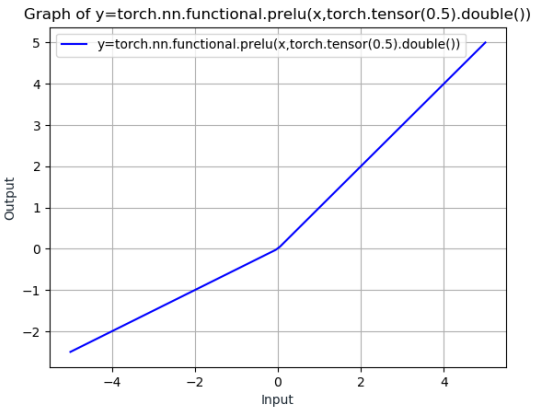
Deep Learning – Bernhard Kainz

Leaky ReLU is a modified version of the ReLU function, defined as $Output = \begin{cases} x & for\ x \geq 0 \\ e.g.\ 0.01 \cdot x & for\ x < 0 \end{cases}$ 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

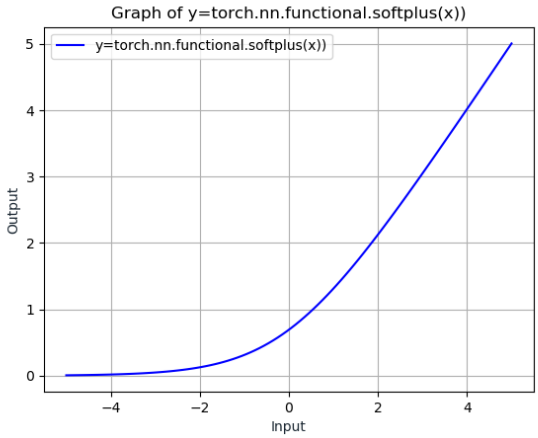
- $Output = \begin{cases} x & for\ x \geq 0 \\ a \cdot x & for\ x < 0 \end{cases}$
- a is learnable
- Either one or a separate a is used for each input channel



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

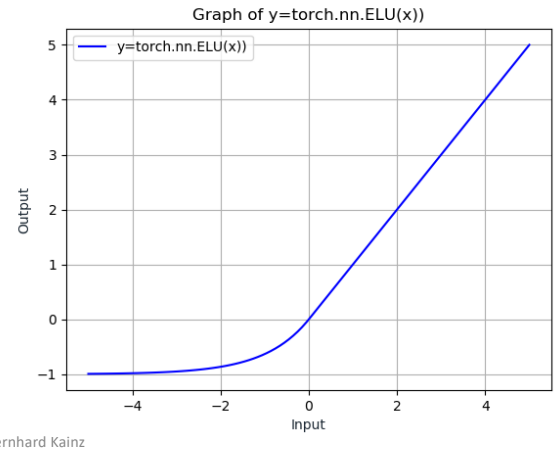
- $Output = \frac{1}{\beta} \cdot \log\left(1 + e^{(\beta \cdot x)}\right)$
- Smooth approximation of ReLU
- Output always positive
- Numerical stability:
  use linear if
  $(\beta \cdot x) > threshold$



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

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



Deep Learning – Bernhard Kainz

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.

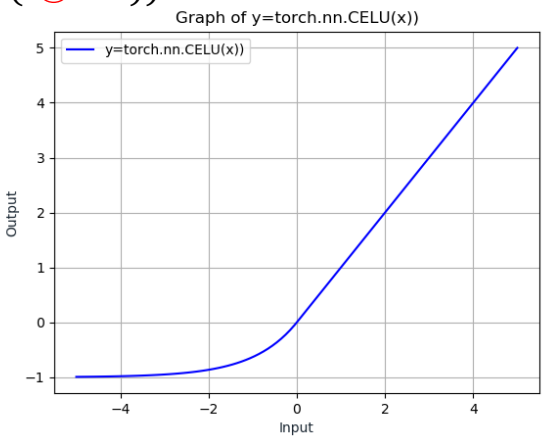
Like SoftPlus, ELU is also a soft, smooth version of the ReLU function. It combines the benefits of both positive and negative output capabilities, making it more versatile depending on the specific application you have in mind.

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

- $Output = \max(0, x) + \min(0, \alpha \cdot (e^{\frac{x}{\alpha}} - 1))$
- 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.

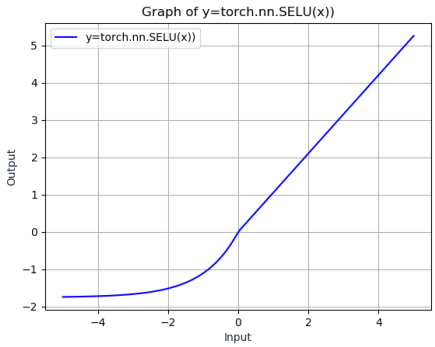
By ensuring that α is not equal to one, the CELU function becomes continuously differentiable across its entire domain. C1 continuity is often desirable in optimization tasks as it ensures a smooth gradient, facilitating more efficient backpropagation.

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

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



Deep Learning – Bernhard Kainz

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.

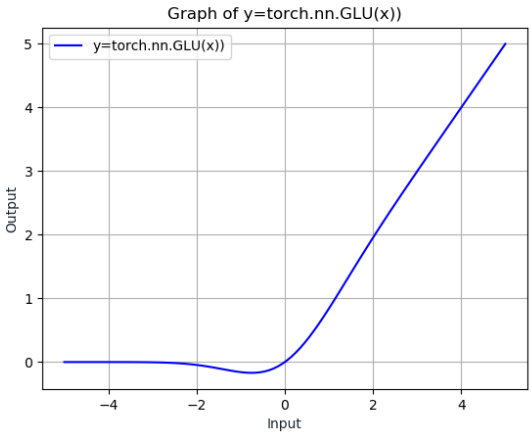
And one more thing: SELUs never die. Due to its design, SELU avoids the "dying unit" problem that plagues some other activation functions, making it a robust choice for certain types of networks.

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

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



Deep Learning – Bernhard Kainz

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).

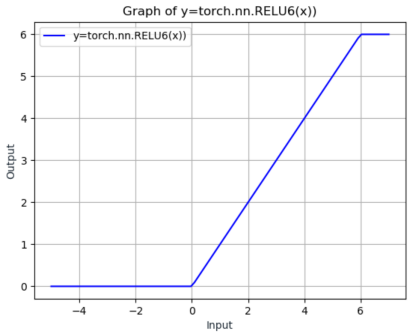
This activation function draws inspiration from the Gaussian distribution, which is a fundamental concept in statistics. The cumulative distribution function, Φ(x), gives us the probability that a normally distributed random variable takes a value less than or equal to x.

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

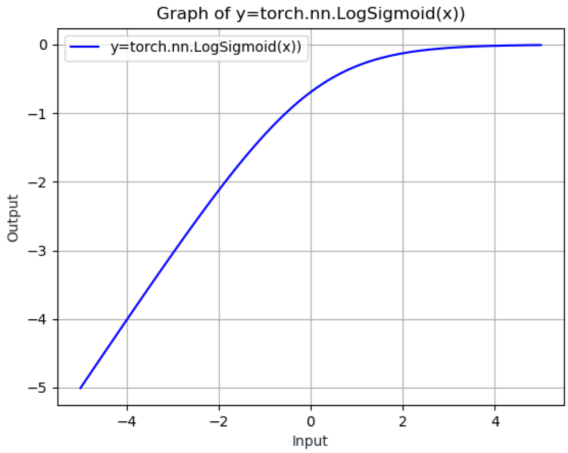
- $Output = \min(\max(0, x), 6)$



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



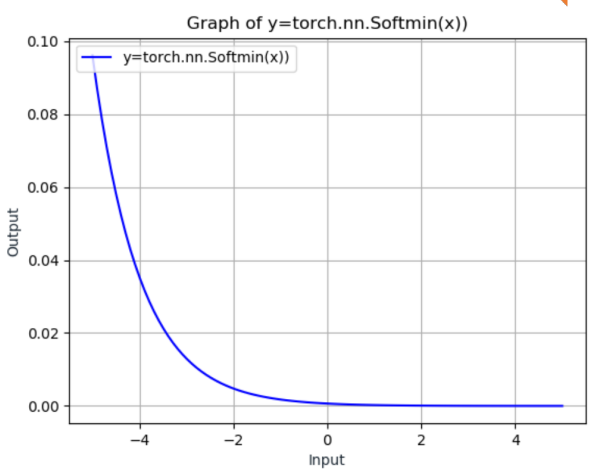
Deep Learning – Bernhard Kainz

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

- $Output = \frac{e^{-x_i}}{\sum_j e^{-x_j}}$
- 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.

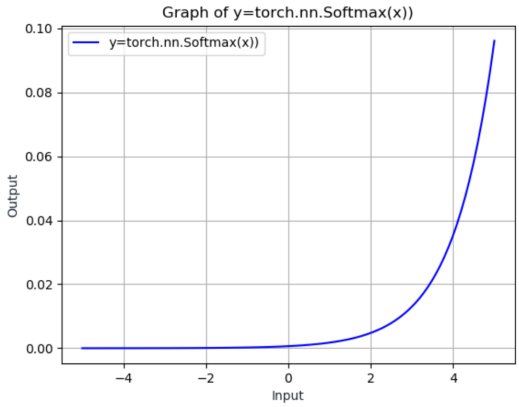


Deep Learning – Bernhard Kainz

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

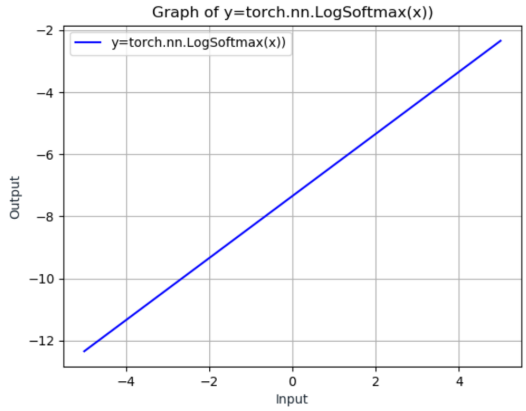
- $Output = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Applies the Softmax function to an n-dimensional input Tensor rescaling them so that the elements of the n-dimensional output Tensor lie in the range [0,1] and sum up to 1.



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

- $Output = \log(\frac{e^{x_i}}{\sum_j e^{x_j}})$
- 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.

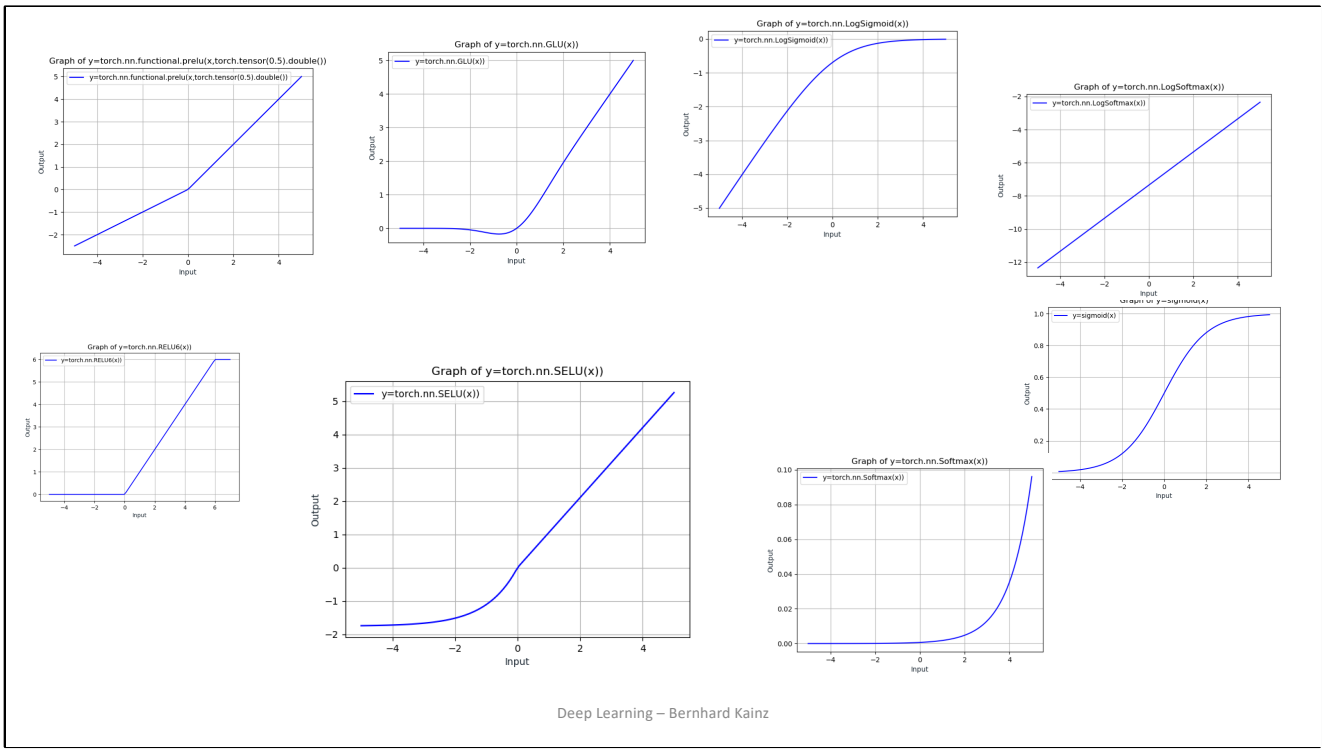
Periodic activations, SIREN

- 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/

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

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.

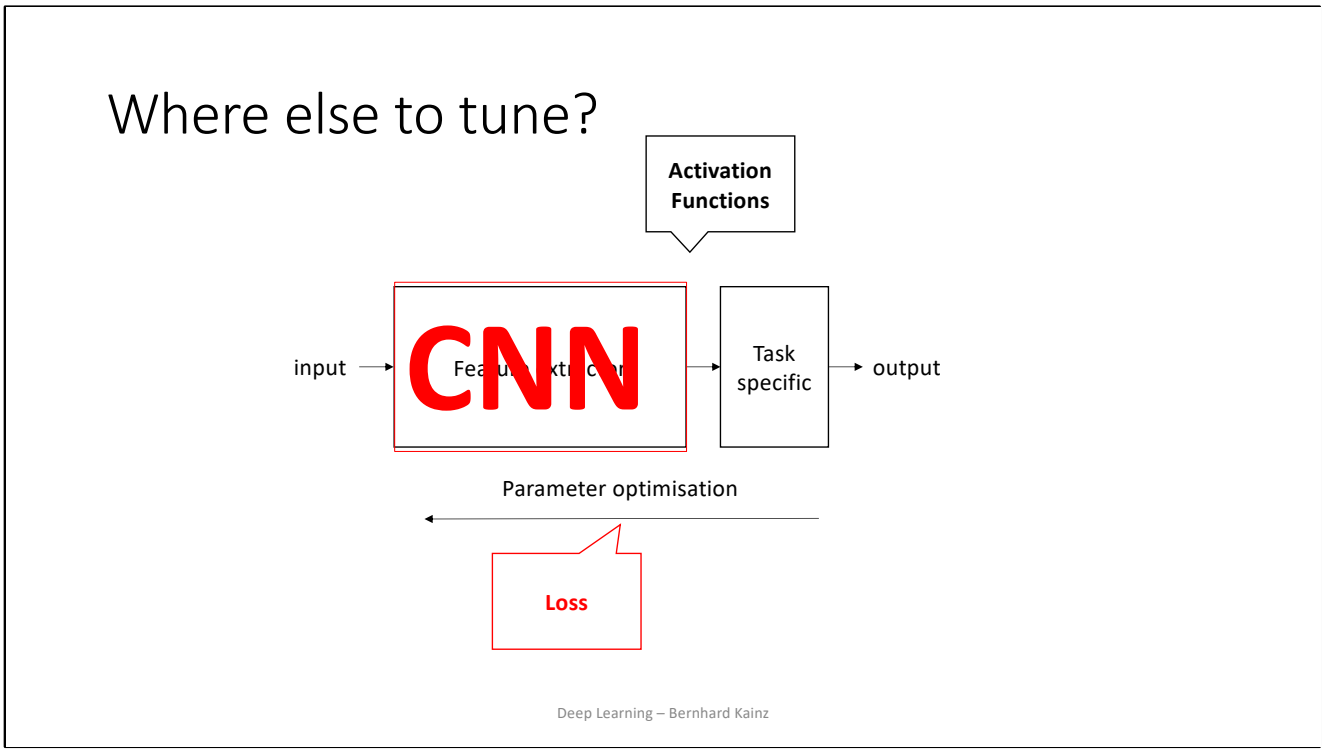
What do we learn from this?

- 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.

Deep Learning – Loss Functions

Bernhard Kainz

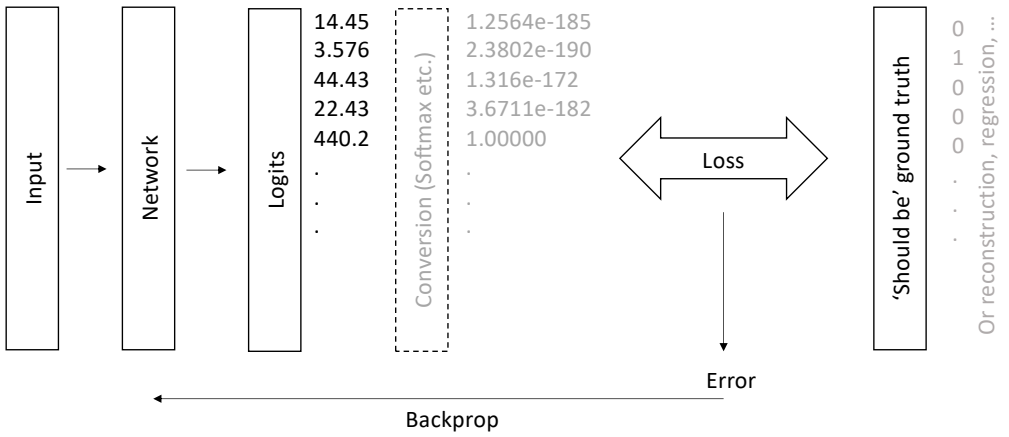
Let’s talk about loss functions.



As we delve deeper into the realm of deep learning, it's important to remember that the network's depth is just one aspect that can be manipulated for improved performance. Beyond network depth, there are two other pivotal areas that demand our attention.
The first aspect is how neurons within the network activate when presented with a set of inputs. This activation process influences the network's ability to capture and represent complex patterns within the data. Careful consideration of activation functions can impact not only convergence speed but also the network's capacity to model intricate relationships.
The second area of focus, and the one we'll be exploring in depth, revolves around the definition of the error or loss that's backpropagated during training. This loss function plays a pivotal role in guiding the network towards optimal weights and biases. It essentially quantifies the gap between the network's predictions and the ground truth.
Interestingly, while activation functions have garnered significant attention, recent years have seen a surge in research dedicated to refining and designing loss functions. The landscape of deep learning is constantly evolving, and this shift towards exploring and innovating loss functions is reflective of the community's commitment to pushing the boundaries of performance and capabilities.
In the early days, much emphasis was placed on activation functions, and their impact on training dynamics. However, the growing recognition of the significance of loss functions has spurred a resurgence of interest in this domain. While losses might have taken a backseat initially, their role in shaping network behavior and fine-tuning training

outcomes cannot be underestimated.
So, as we proceed through this exploration of loss functions, keep in mind that it's yet another key area where innovation and careful consideration can lead to improved neural network performance and remarkable results.

Loss functions



Deep Learning – Bernhard Kainz

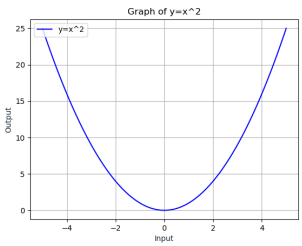
Let's demystify the concept of a loss or error function. At its core, a loss function serves a fundamental purpose within the realm of deep learning.
Imagine you have a trained system that produces certain outputs based on given inputs. Now, in the real world, you often have a desired or expected output for those same inputs.
The role of a loss function is to quantitatively measure how different these produced outputs are from the desired ones.
Think of it as a mathematical tool that assesses the discrepancy between what your system predicts and what it should ideally predict. This discrepancy is distilled into a single numerical value, which encapsulates the extent of the difference.
This numerical value essentially quantifies the error or loss incurred by the system's predictions. In essence, a loss function provides a measure of how well your model is performing in relation to the ground truth. The lower the value of the loss function, the closer your system's predictions are to the desired outcomes.
This concept lies at the heart of training neural networks -- minimizing this loss function drives the network's learning process, guiding it to adjust its parameters and weights to make predictions that align more closely with the ground truth.
So, as we embark on our exploration of different types of loss functions, keep in mind that each function carries a unique way of assessing the performance of your model, and the choice of which to use depends on the specific task, data, and desired outcomes.

L2 Norm, mean squared error

$$\ell(x, y) = \mathcal{L} = \{l_1, \ldots, l_N\}^T, \qquad l_n = (x_n - y_n)^2$$

Reduction to a single value can be either $mean(\mathcal{L})$ or sum$(\mathcal{L})$.

pytorch: `nn.MSELoss()`



The L2 norm is one of the most commonly used loss functions in deep learning -- the Mean Squared Error or MSE.
This loss function is quite intuitive to grasp. When we talk about the mean squared error, we're referring to a measure of the discrepancy between the predicted output of our model and the actual ground truth for a given data sample.

Imagine you have a prediction made by your model, and you compare it to the corresponding actual value.
You calculate the difference between these two, square it, and then take the average of these squared differences.
This is often referred to as the Mean Squared Error, as it computes the average of the squared errors across the entire dataset or a mini-batch of data.

In the context of a mini-batch, you perform this squared difference calculation for each sample within the mini-batch.
The resulting individual squared errors can then be combined into a list.
This list of squared errors for each sample in the mini-batch can be represented as a vector, and this vector is what we denote as $\mathcal{L}$ or l.

Mathematically, $\mathcal{L}$ is defined as a vector of individual squared errors, where each element of the vector represents the squared difference between the predicted output

(x) and the ground truth (y) for a specific data point.
This vector of squared errors can then be either reduced to a single value by taking the mean or the sum of the squared errors across all samples in the mini-batch.

In PyTorch, you'll often encounter this loss function as nn.MSELoss(), which provides an efficient way to compute the mean squared error between predictions and ground truths.

So, the L2 norm, through the Mean Squared Error loss function, serves as a key tool in quantifying how well our model's predictions align with the actual data.
It's a cornerstone in many regression tasks and plays a significant role in training neural networks.

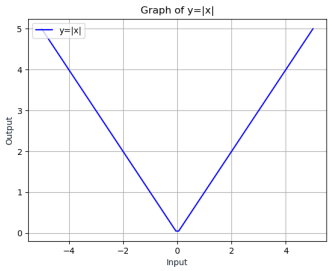
L1 Norm

$$\ell(x, y) = \mathcal{L} = \{l_1, \ldots, l_N\}^T, \qquad l_n = |x_n - y_n|$$

Reduction to a single value can be either $mean(\mathcal{L})$ or $sum(\mathcal{L})$.

Use for robust regression (noisy data)

pytorch: `nn.L1Loss()`



The L1 norm provides us with a different perspective on measuring the error between our model's predictions and the true ground truth.

Mathematically, the L1 norm is represented by a vector $\mathcal{L}$, where each element of the vector corresponds to the absolute difference between the predicted output (x) and the actual ground truth (y) for a specific data sample. This can be denoted as $l\_n = |x\_n - y\_n|$.

Just like we discussed with the Mean Squared Error, this vector of absolute differences can be reduced to a single value by taking either the mean or the sum of the absolute differences across all samples in the dataset or a mini-batch.

The L1 loss is particularly interesting because it's useful for robust regression, especially when dealing with noisy data. In robust regression, you want to give significant weight to small errors, but not as much weight to large errors, making it less sensitive to outliers in your dataset.

However, there's a challenge with the L1 loss—it's not differentiable at exactly zero due to its pointy corners. This lack of differentiability can pose a problem during backpropagation in training neural networks. To address this, you can use a smoothed version of the L1 loss, such as the SmoothL1Loss in PyTorch, which provides a

continuous and differentiable approximation.

In PyTorch, you can conveniently use the nn.L1Loss() function to compute the L1 loss between predictions and ground truths.

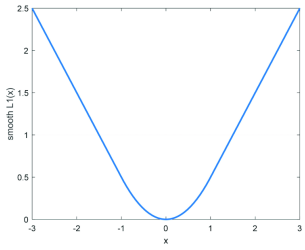
So, the L1 norm plays a key role in training models that can handle noisy data and outliers effectively, making it a valuable tool in various scenarios.

Smooth L1

$$loss(x, y) = \frac{1}{n}\sum_i z_i$$

$$z_i = \begin{cases} 0.5(x_i - y_i)^2, & if\ |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5, & otherwise \end{cases}$$

pytorch: `nn.SmoothL1Loss()`



Deep Learning – Bernhard Kainz

Now, let's delve into the Smooth L1 loss, a variant of the L1 loss that offers some interesting benefits in specific scenarios. This loss function is particularly designed to balance the advantages of both the L1 and L2 norms.

Mathematically, the Smooth L1 loss is calculated as the average of a set of smooth L1 loss terms, each computed for every element z_i in the difference between the predicted output (x_i) and the ground truth (y_i) of a data sample. This smooth L1 term, z_i, is defined based on two conditions: If the absolute difference $|x\_i - y\_i|$ is less than 1, then it's 0.5 times the squared difference; otherwise, it's the absolute difference minus 0.5.

In PyTorch, you can readily utilize the nn.SmoothL1Loss() function to compute this loss efficiently.

What makes the Smooth L1 loss intriguing is its behavior. For errors close to zero, it behaves like the L2 loss, which means it's quadratic in that region. As the error magnitude increases, it transitions to behaving like the L1 loss, which is linear for larger errors. This dual behavior allows the loss to strike a balance between the two norms, making it more robust to outliers while still accounting for smaller errors.

This robustness against outliers makes the Smooth L1 loss suitable for tasks where the

dataset might contain noisy or outlier-ridden samples. By mitigating the extreme sensitivity of the L1 loss to outliers, the Smooth L1 loss helps prevent undue influence on the training process from those data points that deviate significantly from the norm.

However, there is a trade-off. The Smooth L1 loss introduces a scale factor, often set to 0.5, which determines the point at which the transition from quadratic to linear behavior occurs. Choosing an appropriate scale factor can be a challenge since it depends on the distribution of the errors in your dataset, which might not be known beforehand.

The Smooth L1 loss provides a middle ground between L1 and L2 norms, offering improved robustness and adaptability, making it a valuable choice when handling real-world data that could have diverse characteristics.

Negative log likelihood loss

- Assumption: Network output represents log liklihoods.
- Make the desired output as large as possible and all others as small as possible

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T, \qquad l_n = -w_{y_n} x_{n,y_n},$$
$$w_c = weight[c] \cdot 1\{c \neq ignore_{index}\}$$

$$\ell(x, y) = \begin{cases} \sum_{n=1}^{N} \frac{1}{\sum_{n=1}^{N} w_{y_n}} l_n, & if\ reduction = mean \\ \sum_{n=1}^{N} l_n, & if\ reduction = sum \end{cases}$$

pytorch: `nn.NLLLoss()`
Implementation valid for all such problems, not only likelihoods.

The Negative Log Likelihood (NLL) loss, is a loss function that's particularly relevant when dealing with classification tasks and probability-based outputs. This loss is built upon a simple assumption: that the network's output can be interpreted as log likelihoods, usually representing class probabilities.

Mathematically, the NLL loss is calculated for each data sample, with each term representing the negative logarithm of the network's output probability for the ground truth class. The weights *w_c* are used to assign different importance to different classes. It's designed to maximize the likelihood of the correct class while minimizing the likelihoods of other classes.

In PyTorch, the nn.NLLLoss() function can be used to compute this loss effectively. Importantly, its implementation extends beyond likelihood-based problems; you can apply it to various scenarios.

The equation itself involves a summation across the data samples, aiming to minimize the negative log likelihood for the correct class's score while considering the weights. If the reduction strategy chosen is "mean," it calculates the mean NLL over the batch; if it's "sum," it calculates the sum of the NLL values.

Although the name "Negative Log Likelihood" suggests its association with likelihood-

based scenarios, it's worth noting that the concept isn't limited to likelihoods. The key idea is to drive the network to favor the correct class while penalizing other classes, making it versatile for a range of classification problems.

One significant feature of the NLL loss is its flexibility to assign different weights to classes. This can be particularly valuable when your training data exhibits class imbalance, where some classes are much rarer than others. Assigning appropriate weights can help counterbalance this issue during training, ensuring that the network learns effectively even when faced with skewed class distributions.

An alternative approach to handling class imbalance is to increase the frequency of rare classes during training, but this may not always be practical. The NLL loss's ability to handle class weighting offers a more adaptable solution.

So, while the name "Negative Log Likelihood" may sound specific, its utility extends beyond likelihood-based scenarios, making it an essential tool for training classifiers in diverse contexts.

Cross Entropy (CE) Loss

- Combines LogSoftmax and NLLLoss
- Useful for classification problems with C classes

$$loss(x, class) = -\log\left(\frac{e^{x[class]}}{\sum_j e^{x[j]}}\right) = -x[class] + \log\left(\sum_j e^{x[j]}\right)$$

Classes can also be weighted.

The losses are averaged across observations for each minibatch.

pytorch: `nn.CrossEntropyLoss()`

The Cross Entropy (CE) Loss is one of the most widely used loss function that's especially relevant for classification tasks involving multiple classes.

Mathematically, the Cross Entropy Loss is a combination of two components: the LogSoftmax activation and the Negative Log Likelihood (NLL) loss. This makes it a powerful tool for training classification models.

In classification problems with C classes, the CE loss functions by taking the LogSoftmax of the input scores and then applying the NLL loss. It's designed to make the output of the LogSoftmax as large as possible for the correct class while minimizing the scores for other classes. This is achieved by taking the negative logarithm of the softmax probability for the correct class.

Notably, you can also introduce class weights to the CE loss, allowing you to assign different levels of importance to different classes. This can be particularly useful when some classes are more significant or rare in your dataset.

In PyTorch, you can readily use the nn.CrossEntropyLoss() function to compute this loss efficiently. The loss values are typically averaged across observations within each minibatch.

The cross entropy loss is particularly favored for classification tasks, as it elegantly combines the log softmax operation and the negative log likelihood loss. Conceptually, it takes the scores, passes them through a softmax function, takes the logarithm of those values, and then optimizes to maximize the correct class's output while minimizing those of other classes.

When you backpropagate through the LogSoftmax operation, it has the effect of making the scores of incorrect classes as small as possible, thereby focusing on improving the correct class's score. In simpler terms, it aims to minimize the negative score of the correct class and adds a log term to diminish the influence of scores from other classes.

The Cross Entropy Loss stands as a powerful tool for training classifiers, adeptly handling multiple classes and their associated probabilities. Its combination of LogSoftmax and Negative Log Likelihood makes it a cornerstone of classification-focused deep learning tasks.

Binary Cross Entropy (BCE) Loss

- CE loss for only two classes

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T,$$
$$l_n = -w_n[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)$$

Reduction to a single value can be either $mean(\mathcal{L})$ or sum$(\mathcal{L})$.

pytorch: `nn.BCELoss()`

Requires [0,1] probabilities. If this cannot be guaranteed, use `nn.BCEWithLogitsLoss()`

The Binary Cross Entropy (BCE) Loss is a specific variant of the Cross Entropy loss designed for tasks involving only two classes.

Mathematically, the BCE Loss is formulated as the negative weighted sum of two entropy terms. For each observation in the batch, the loss is calculated using the following equation:

$\ell(x,y)$ = $\mathcal{L}$ = {l_1, ⟦...,l⟧_N }^T, l_n = −w_n [y_n·log (1−x_n)]

Here, x_n represents the predicted probability for the positive class, y_n is the ground truth label (0 or 1) for that observation, and w_n is a weight associated with that sample.

Just like the Cross Entropy Loss, the BCE Loss can also be reduced to a single value using either the mean or sum operation.

In PyTorch, you can conveniently employ the nn.BCELoss() function to compute this loss. However, it's important to note that the input probabilities, x_n, must lie in the [0, 1] range to ensure the loss's validity. If such guarantees can't be made about the input probabilities, it's advisable to use the nn.BCEWithLogitsLoss() function instead.

Notably, the BCE Loss does not incorporate the log softmax operation and instead directly models the entropy of a Bernoulli distribution. This makes it suitable for binary classification tasks where each observation belongs to one of two classes.

It's worth highlighting that the BCE Loss finds significant utility in reconstruction tasks, such as those involving auto-encoders, where the goal is to measure the error of a reconstruction compared to the original data.

In essence, the Binary Cross Entropy Loss provides an efficient and effective way to measure the difference between predicted probabilities and ground truth labels in binary classification scenarios.

Kullback-Leibler Divergence Loss

- Measures distance between distributions

$$\ell(x, y) = \mathcal{L} = \{l_1, \dots, l_N\}^T, \qquad l_n = y_n \cdot (\log y_n - x_n)$$

Pytroch: `nn.KLDivLoss()`

Deep Learning – Bernhard Kainz

The Kullback-Leibler (KL) Divergence Loss is a loss function that helps measure the distance between probability distributions.

Mathematically, the KL Divergence Loss is defined as the element-wise product of the ground truth probabilities (y_n) and the logarithm of the ratio between the ground truth probabilities (y_n) and the predicted probabilities (x_n):

$\ell(x,y) = \mathcal{L} = \{l_1, [\![...,l]\!]\_N \}\hat{}T$, $l\_n = y\_n \cdot (\log y\_n - x\_n)$

The nn.KLDivLoss() function in PyTorch facilitates the computation of this loss.

It's essential to note that the KL Divergence Loss is particularly suitable when your target distribution is represented as a one-hot distribution, where a single category is assigned a value of 1 and the others are 0. The loss assumes both predicted probabilities (x_n) and ground truth probabilities (y_n) are valid probabilities, meaning they must lie between 0 and 1.

However, this loss has a notable limitation—it lacks the incorporation of a softmax or log-softmax operation. As a result, it can sometimes suffer from

numerical stability issues, especially when dealing with small probability values.

Despite this limitation, the KL Divergence Loss proves to be very useful in certain scenarios. For instance, it finds a strong application in variational autoencoders (VAEs), a type of generative model used in unsupervised learning. VAEs aim to learn the underlying distribution of data and use the KL Divergence Loss as a crucial component in their training process.

So the Kullback-Leibler Divergence Loss serves as a means to quantify the divergence between two probability distributions, and although it might exhibit numerical challenges, it remains a valuable tool for specific tasks like variational autoencoders.

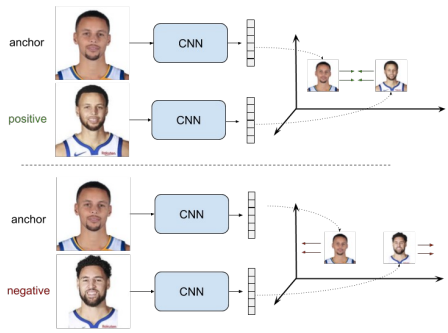
Margin Ranking Loss/Ranking Losses/Contrastive loss

$$loss(x, y) = \max(0, -y \cdot (x_1 - x_2) + margin)$$

Useful to push classes as far away as possible and for metric learning

Practical: take category that scores is closest or higher than correct one change until difference is at least the margin

pytorch: `nn.MarginRankingLoss()`



https://gombru.github.io/

This class of loss functions has a unique purpose—instead of directly predicting labels or values, they focus on predicting relative distances between inputs.

Mathematically, the Margin Ranking Loss is defined as follows:
$loss(x,y)$ = max(0, $-y$·(x_1−x_2)+$margin$)

This loss function serves a crucial role in pushing classes away from each other as far as possible. It finds particular use in metric learning, a task where the objective is to learn a meaningful distance metric between data points.

To better understand its practical application, consider scenarios where you aim to identify if two inputs belong to the same class (similar) or different classes (dissimilar). Instead of predicting specific values or labels, you're concerned with how these inputs relate in terms of similarity.

In PyTorch, you'll find the nn.MarginRankingLoss() function ready for your use in implementing this loss.

Margin ranking losses offer a unique training methodology. You start by

extracting features from two inputs and obtaining embedded representations for them. Next, a metric function, like Euclidean distance, measures the similarity between these representations. The goal is to train the feature extractors to produce similar representations for similar inputs and distinct representations for dissimilar inputs. This powerful strategy generates potent representations applicable to various tasks.

In essence, Margin Ranking Losses operate around the concept of pushing one input's score above another's by a defined margin. If the difference in scores satisfies this condition, the cost is zero. However, if the difference falls below the margin, the cost increases linearly. This loss is particularly handy in cases like classification, where you compare the scores of the correct answer (x_1) with the highest-scoring incorrect answer (x_2) in the mini-batch.

Keep in mind, though, that Margin Ranking Losses aren't confined to classification tasks alone. They also shine in energy-based models, where they exert downward pressure on the correct answer's score and push the incorrect answer's score upward.

So, as we explore the diverse landscape of loss functions, remember that Margin Ranking Losses offer a powerful way to train models that capture the essence of relative distances and similarities in your data without defining the distance metric heuristically.

Triplet Margin Loss

- $\ell(x, y) = \mathcal{L} = \{l_1, \ldots, l_N\}^T$,
  $l_n(x_a, x_p, x_n) = \max(0, \mathrm{m} + |\mathrm{f}(x_a) - f(x_p)| - |\mathrm{f}(x_a) - f(x_n)|$

actual positive sample negative sample

Make samples from same classes close and different classes far away.

Objective: Distance for the good pair has to be smaller than distance to the bad pair. Actual distance does not need to be small, just smaller.

Used for metric learning and Siamese networks

pytorch: `nn.TripletMarginLoss()`

Mathematically, the Triplet Margin Loss is expressed as:
$\ell(x,y)$ = max(0, m + |f(x_a)–$f(x\_p$)|–|f(x_a)–$f(x\_n$)|

At its core, this loss function is all about making samples from the same classes come close to each other, while simultaneously pushing samples from different classes farther apart. The objective is to ensure that the distance between a "good pair" (similar samples) is smaller than the distance between a "bad pair" (dissimilar samples).

The key to understanding this loss lies in the comparison of distances. The actual distances themselves don't need to be small, but rather the relative difference matters more. In essence, we aim to reduce the distance between the "good" pair while simultaneously increasing the distance between the "bad" pair by at least a certain margin, denoted as 'm'.

The Triplet Margin Loss is widely used in metric learning, a field that focuses on training models to generate meaningful representations that capture the relative similarities or differences between data points. One common scenario is in Siamese networks, where two inputs pass through a shared neural network, and their resulting vectors are compared.

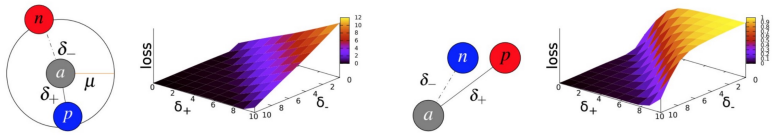
Consider a scenario where we feed two images of the same category into a CNN, yielding two vectors. In this case, we want the distance between these vectors to be minimized. Conversely, for two images of different categories, we want the distance between their vectors to be maximized.

This loss function effectively steers the model toward the desired outcome by nudging the distances accordingly. The important part is to ensure that the distance between the "good pair" is smaller than the distance between the "bad pair," with a margin of 'm' in between.

It's noteworthy that the concept of the Triplet Margin Loss found practical application in training an image search system for Google. This system encoded user queries into vectors and compared them to indexed image vectors. The system then retrieved images that closely matched the query vector, a concept fundamental to understanding this loss.

In PyTorch, implementing the Triplet Margin Loss is as simple as using the nn.TripletMarginLoss() function, making it a powerful asset in training models to comprehend and leverage the relative distances within data points.

Triplet Margin Loss



(a) (b) (c) (d)

Figure 1: (a) Margin ranking loss. It seeks to push n outside the circle defined by the margin μ, and pull p inside. (b) Margin ranking loss values in function of δ_-, δ_+ (c) Ratio loss. It seeks to force δ_+ to be much smaller than δ_-. (d) Ratio loss values in function of δ_-, δ_+

$$\hat{\delta_+} = ||f(\boldsymbol{a}) - f(\boldsymbol{p})||_2 \text{ and } \hat{\delta_-} = ||f(\boldsymbol{a}) - f(\boldsymbol{n})||_2.$$

$$\lambda(\delta_+, \hat{\delta_*}) = max(0, \mu + \delta_+ - \hat{\delta_*})$$

http://www.bmva.org/bmvc/2016/papers/paper119/index.html

In the original paper they had a nice figure illustrating this behaviour. Positive examples are pulled closer and negative examples pushed further away.

Cosine Embedding Loss

$$loss(x, y) = \begin{cases} 1 - \cos(x_1, x_2), & if\ y = 1 \\ \max(0, \cos(x_1, x_2) - margin), & if\ y = -1 \end{cases}$$

Measure weather two inputs are similar or dissimilar

Basically a normalised Euclidian distance

pytroch: `nn.CosineEmbeddingLoss()`

Cosine Embedding Loss is a type of loss function used for comparing the similarity or dissimilarity between two input vectors x1 and x2.

The function takes a label y as an input, where y=1 indicates that the vectors should be similar, and y=−1 indicates that they should be dissimilar.

Contrary to common belief, this is not just a normalized Euclidean distance. It specifically utilizes the cosine of the angle between vectors, which is a measure of orientation, not magnitude.

In PyTorch, this loss function is available as nn.CosineEmbeddingLoss().

For pairs labeled as similar (y=1), the loss tries to minimize the angle between the vectors, effectively making their cosine similarity close to 1.

For pairs labeled as dissimilar (y=−1), the loss aims to ensure that the cosine similarity is smaller than a specified margin. The margin is a hyperparameter and is usually a small positive value.

Why use cosine similarity over Euclidean distance? The key benefit lies in its focus on the direction of vectors, not their magnitude.

You can think of your data points as vectors on a high-dimensional sphere. After normalization, the data points reside on the surface of the sphere.

In such a high-dimensional space, the equator of the sphere has a large surface area, providing ample room to separate dissimilar vectors.

The goal, then, is to make similar vectors point in the same direction, while dissimilar vectors should be separated but not necessarily be antipodal. This is why we often set the margin to a small positive value, maximizing the use of the "equatorial space" for dissimilar vectors.

The Cosine Embedding Loss is especially useful for learning nonlinear embeddings and for semi-supervised learning tasks, where the objective is to understand the semantic relationships between data points.

What do we learn from this

- The choice of loss depends on the desired output (e.g., classification vs. regression)
- Loss functions are a hot topic of research.
- It informs how the overall system behaves during training
- Don’t get scared by the equations. If you look closely the underlying ideas are very simple.

The choice of a loss function is crucial because it directly influences the type of output your model will produce.
For example, you'll likely use a different loss function for classification tasks than you would for regression tasks.
Loss functions are an active area of research in the machine learning community.
This means that new functions and techniques are being developed frequently, so it's worth staying updated on the latest advancements.
The loss function you choose fundamentally guides the behavior of your model during the training process.
In essence, it sets the "rules of the game" for how the model should optimize its predictions.
While the mathematical formulations of loss functions may initially appear daunting, don't be intimidated.
Once you unpack them, the core ideas behind most loss functions are usually straightforward and intuitive.
The loss function is not just a formula you have to minimize; it's a crucial part of your model's learning process.
Selecting the right one can make the difference between a well-performing model and one that fails to capture the nuances of your data.