Deep Learning

Basics

Antoine PALISSON

TABLE OF CONTENTS

01

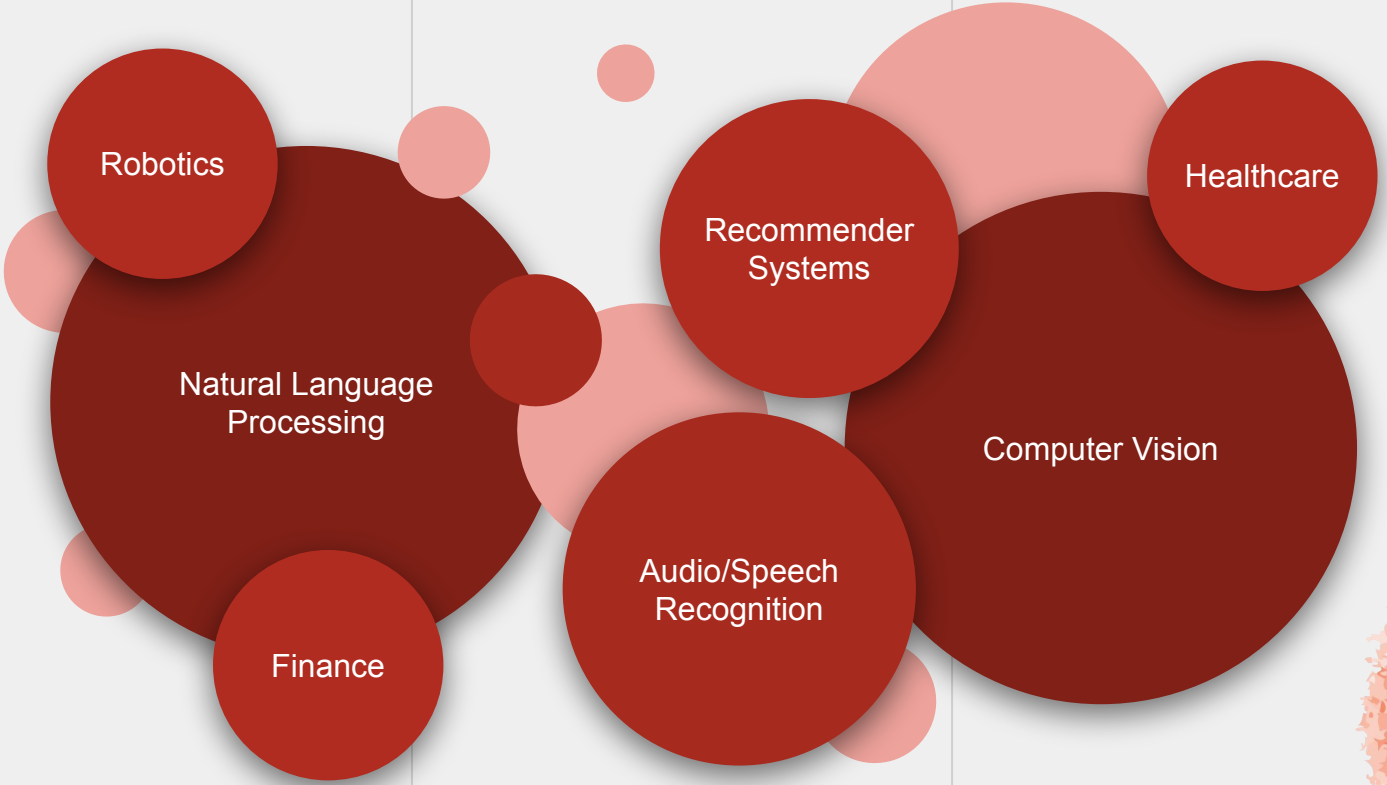
Introduction

Introduction

Deep learning is a type of **artificial intelligence** that uses complex algorithms to learn from data and make predictions or decisions. It is based on the idea of artificial neural networks, which are **inspired by the structure and function of the human brain**.

These networks are made up of many layers, and **each layer processes and analyzes the data to identify patterns and make decisions**. The more data a deep learning algorithm processes, the better it becomes at recognizing patterns and making predictions.

Usages


Robotics
Natural Language Processing
Finance
Recommender Systems
Audio/Speech Recognition
Computer Vision
Healthcare

History

Deep learning has its roots in the field of artificial neural networks, which was first developed in the 1940s and 1950s. However, early neural networks were limited in their capabilities due to the **lack of computing power** and the **absence of sufficient training data**.

The concept of deep learning only emerged in the late 2000s, when advancements in hardware and the availability of large amounts of data made it possible to train deep neural networks with many layers.

History (2)

The origins

1943 — **First concept**

Warren McCulloch and Walter Pitts published a paper proposing the concept of artificial neural networks.

1957 — **Perceptron**

The perceptron neural network was largely inspired from the work of Frank Rosenblatt.

60's 70's — **Deep Neural Networks**

Deep Neural Networks were developed during the 60's and the 70's. However, the interest in deep learning plummeted as the training of the networks was hard.

80's — **RNN & CNN**

Recurrent Neural Network and Convolutional Network were developed during the 80's by researchers such as Rumelhart and LeCun .

1986 — **Backpropagation**

The backpropagation algorithm was improved by Rumelhart, Hinton and Williams. It allowed for the efficient training of deep neural networks.

90's — **Resurgence of interest**

Neural networks interest rose again due to the development of powerful GPUs. The RNN were particularly popular especially with the creation of the LSTM networks.

2006 — **Deep Belief Networks**

Hinton and his team demonstrated that deep neural networks could be trained in an unsupervised manner effectively.

00's — **Development**

The surge of interest about deep learning and the growing computational power led to the foundation for the breakthroughs that would come in the 2010s.

History (3)

The rise

2012

AlexNet

Alex Krizhevsky and his team won the ImageNet competition using a CNN model. The error was cut down from 26% to 16%. It is one of the most cited paper in the AI field.

2014

GANs

Generative Adversarial Networks were introduced by Ian Goodfellow and his colleagues. GANs are able to generate very realistic data such as images.

RNN Encoder-Decoder

The ability of RNNs to capture relations between elements in a sequence was used to process and generate text and speech.

2015

AlphaGo

DeepMind developed a CNN model trained with reinforcement learning that beat a human pro at the game of Go. Prior to 2015, the best Go reached amateur level.

2017

Attention is all you need

Vaswani and his team introduced attention neural networks also known as transformer that led to major advances in natural language processing.

2020

AlphaFold 2

DeepMind team introduced a deep learning system that can predict the 3D structure of proteins with remarkable accuracy (~90%).

GPT-3

OpenAI developed the biggest transformer model for NLP at that time. This model was then used as a base for the DALL-E 2 (a realistic image generation model) and ChatGPT (a very performant chatbot).

Deep Learning Architectures

Three main **neural network architectures** can be separated based on their neuron mechanisms:

- **MLPs** - *Stands for Multi-Layer Perceptron. It has multiple layers of densely connected neurons that perform a linear combination of the inputs followed by a nonlinear function.*
- **CNNs** - *Stands for Convolutional Neural Network. It is particularly effective at processing multidimensional data with a grid-like topology such as images by learning spatial relationships.*
- **RNNs** - *Stands for Recurrent Neural Network. It has loops in its architecture, allowing it to process sequences such as time series or text while maintaining a memory of the elements in the sequence.*

High Level Architectures

High-level architectures can be composed of a variety of lower-level neural network components, such as MLPs, CNNs, and RNNs, and may also include specialized components. Some of the main ones include:

- **Autoencoder** (AEs) - *It is used for unsupervised learning of feature representations. It has an encoder part that maps the input data to a lower-dimensional latent space and a decoder part that maps the latent space back to the input space.*
- **GANs** - *Stands for Generative Adversarial Network. It has two networks : the first learns to generate new data that is similar to the training data while the second learns to distinguish between the generated data and the real training data.*

High Level Architectures (2)

High-level architectures can be composed of a variety of lower-level neural network components, such as MLPs, CNNs, and RNNs, and may also include specialized components. Some of the main ones include:

- **Transformer** - *It uses self-attention mechanisms to process sequences that make it able to selectively focus on different parts of the sequence. This allows it to have a better understanding of the context than traditional RNNs. Recently, the transformer architecture has also been extended to images. As of today, it is one of the most used and one of the most efficient architecture.*

Organizations

Today, a handful of organizations are actively contributing to the development and advancement of deep learning, including:

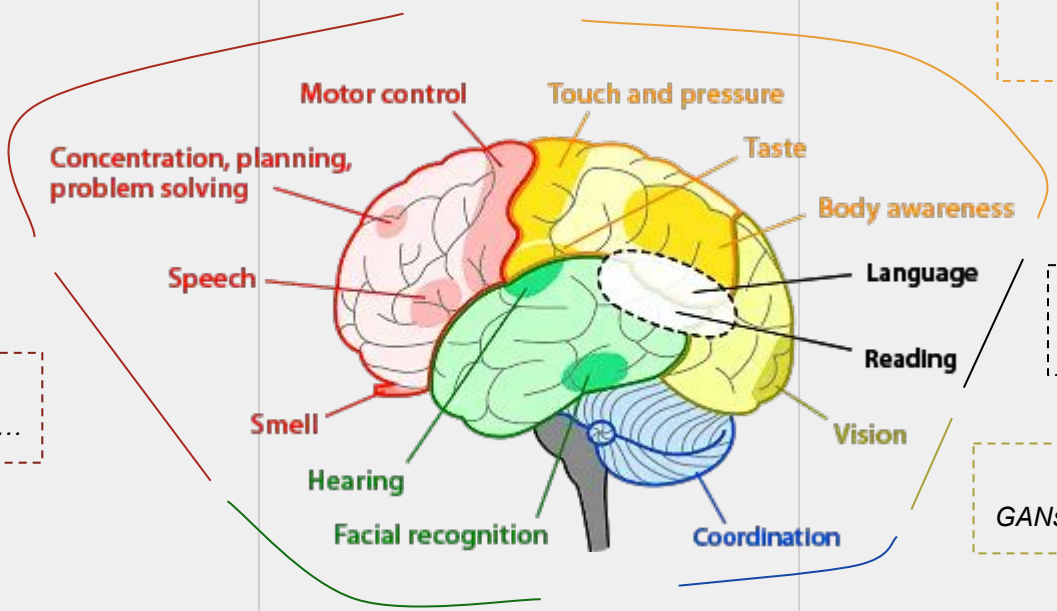
- **Google** : *It has released many deep learning frameworks and tools, including TensorFlow. It also possess **DeepMind** which is one of the most active organization in the development of AI.*
- **OpenAI** : *It is known for releasing powerful deep learning models, including GPT-3 and its variants.*
- **Meta** : *It has released many deep learning frameworks and tools, including PyTorch.*

Other organizations are also involved such as Microsoft, NVIDIA, IBM, Amazon, Baidu and so on.

An AI Brain

Deep Reinforcement Learning

CNNs & RNNs
GANs, transformer …



CNNs & RNNs
transformer …

CNNs & RNNs
GANs, VAEs, transformer …

CNNs
GANs, VAEs, transformer …

CNNs & RNNs
GANs, AEs, transformer …

Deep Reinforcement Learning

Source : Arizona State University

An AI Brain (2)

The human brain is a complex system with many different regions that perform specialized functions such as the **frontal**, the **parietal**, the **occipital** and the **temporal** lobes as well as the cerebellum.

Replicating this complexity with a single algorithm is not currently possible. However, certain deep learning algorithms can be used to perform tasks similar to those performed by specific regions of the brain.

Keep in mind that **the human brain is still not fully understood** and the functions of each region are still being studied. Additionally, deep learning research and development is rapidly evolving.

Thus, **the way AIs are designed will most probably change in the near future**.

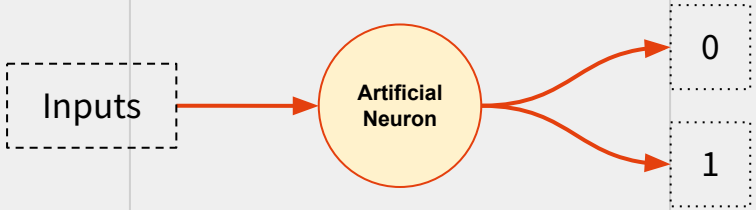
Perceptron &
Multi Layer Perceptron

Artificial Neurons

Definition

Before the perceptron was invented, McCulloch and Pitts proposed another architecture called the **artificial neurons**. It is a simple binary model of a neuron that receives input signals and **produces an output signal of either 0 or 1** :

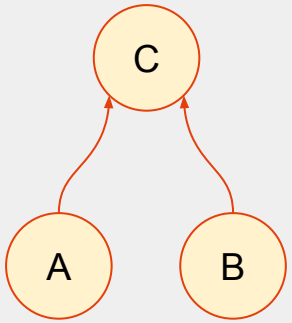
- If the output meets a certain threshold, say 1, the artificial neuron "fires" and produces an output of 1
- Otherwise it produces an output of 0



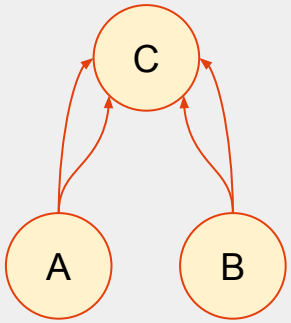
Artificial Neurons (2)

Logical Computations

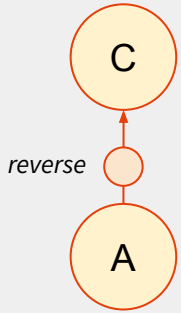
This artificial neuron could perform **simple logical computations** such as the **AND**, **OR** and **NOT**. Let's assume that A and B neurons are fires 1 when activated and 0 otherwise. Additionally, the C neuron activation threshold is set to 1.



AND : *if A and B are firing, the average of A and B fires is 1, thus C is activated.*



OR : *if A is firing, the average of 2*A is 1, thus C is activated. Same for B.*



NOT : *A output is reversed, thus C is activated if A is not firing.*

Perceptron

Definition

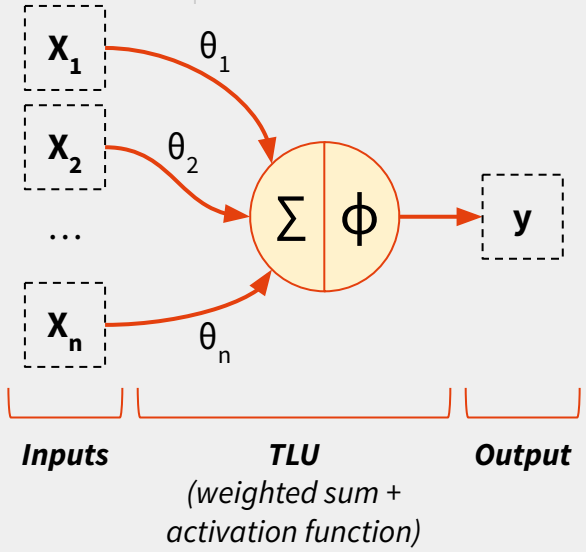
The **perceptron** was introduced in 1957 by Frank Rosenblatt. It is composed of multiple artificial neurons called **Threshold Logic Unit** (TLU) which have some key differences with the McCulloch and Pitts artificial neuron :

- **Output** - *the perceptron can output any value between -1 and 1.*
- **Training** - *the perceptron is trainable whereas the McCulloch and Pitts model was not designed to be trained.*
- **Activation Function** - *the perceptron output is generally passed through a nonlinear function such as the Heaviside or the Sign functions.*

Perceptron (2)

TLU architecture

The TLU computes a **weighted sum of its inputs** and then **applies a function**, called activation function, to the output.

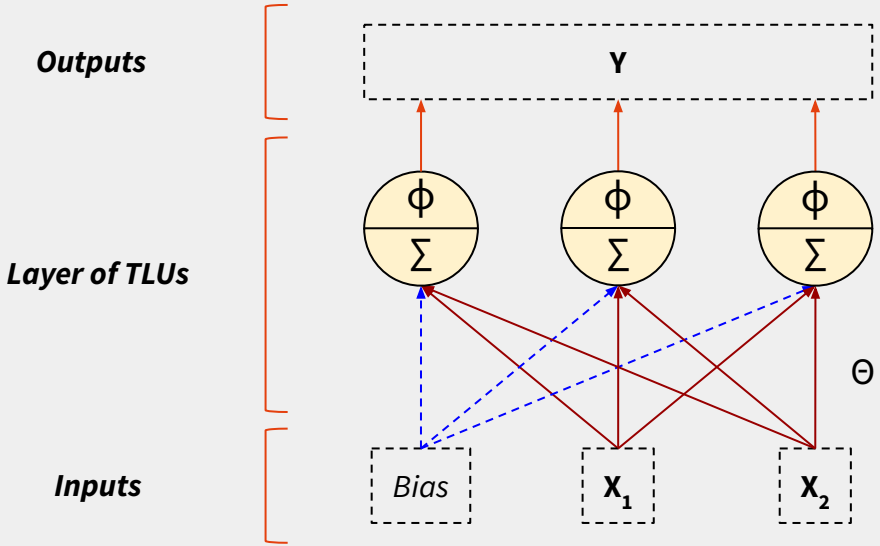


$$y = \Phi\left(\sum_{i=1}^{n} \theta_i X_i\right)$$

Perceptron (3)

TLU layer

Generally, the perceptron was composed of two layers : an **input layer** (inputs and the bias) and a **layer with multiple TLUs**.



Perceptron (4)

Weights

The inputs, the bias, the activation function and the outputs are fixed which means that the only parameters that can be updated are the weights. Thus, **the weights are the "predictive power" of the perceptron**.

However, if all weights in a perceptron are initialized to the same value, then all TLUs in the same layer will compute the same output resulting in a symmetry problem.

Consequently, **the weights of the perceptron were typically initialized randomly**, using a normal distribution with mean 0 and a small standard deviation.

Perceptron (5)

Activation Function

At first, **the perceptron was designed for binary classification**. To do so, the the heaviside or the sign activation functions ϕ were used.

$$f(x) = \begin{cases} 0 \text{ if } z < 0 \\ 1 \text{ if } z \geq 0 \end{cases}$$

The heaviside function

$$f(x) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0 \end{cases}$$

The sign function

Later on, other activation functions were used along with the perceptron such as the sigmoid and the ReLU functions.

Perceptron (6)

Perceptron learning rule

As opposed to the first artificial neurons, **the weights θ of the perceptron are updated** during a learning, also called training, phase. This optimization step is called the **perceptron learning rule**.

$$\theta_{i+1,j} = \theta_{i,j} - \eta \left(y_j - \hat{y}_j \right) X_i$$

with :

- $\theta_{i+1,j}$ the updated weight
- $\theta_{i,j}$ the current weight
- η the learning rate
- y_j the target output of the TLU **j**
- $\hat{y}_j$ the prediction of the TLU **j**
- X_i the input **i**

Perceptron (7)

Loss Function

The Perceptron Learning rule uses **the errors made by the network predictions** to update the weights:

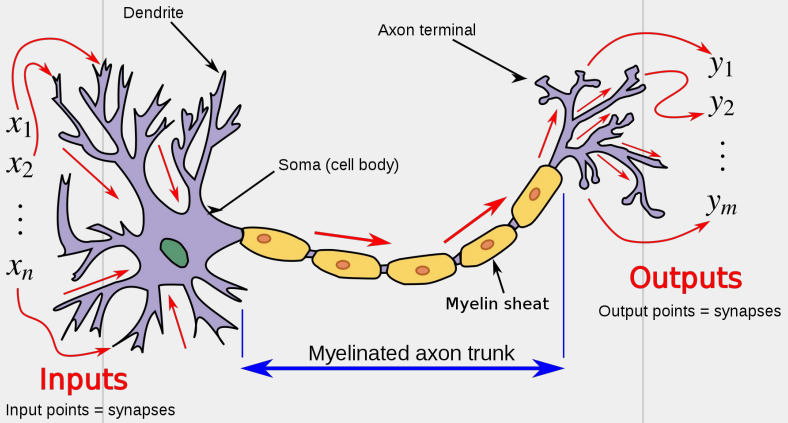
$$\theta_{i+1,j} = \theta_{i,j} - \eta \left(y_j - \hat{y}_j\right) X_i$$

The difference between the true value and the prediction of the perceptron.

This “error function” is also called the **Loss Function**. The term “loss” refers to the **amount of information that is lost** as a result of the model's predictions being different from the true output.

Biological Neurons

The structure and function of the human brain was a **major inspiration for the development of deep learning algorithms**. Just as neurons in the brain process and transmit information to each other, artificial neurons in a deep learning algorithm process information and transmit it to other neurons in the next layer.



Source : Wikipedia

Biological Neurons (2)

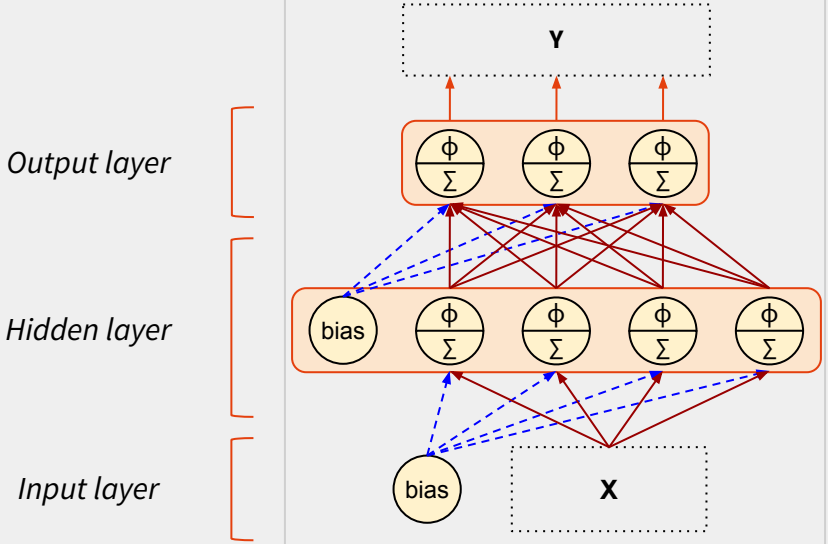
However, artificial neural networks (ANN) have **gradually become quite different from the biological neural networks** :

- **Online training** - *ANNs have static connections that are set during the training process, whereas the human brain is constantly adapting and improving over time.*
- **Parallelization** - *The human brain is capable of processing a vast amount of information in parallel, whereas ANNs can only process information in a sequential manner.*
- **Versatility** - *The human brain is capable of performing many functions, such as perception, thought, and emotion, whereas ANNs are typically designed to perform specific tasks.*

Multilayer Perceptron

Definition

An **MLP** is composed of **stacked layer of TLUs**. This architecture allows for **more complex relationship** between inputs and outputs and to solve problems that are **not linearly separable**.



Multilayer Perceptron (2)

Tasks

As opposed to Perceptrons, MLPs were used for **both classification and regression tasks**.

For **classification** tasks, the output layer of an MLP typically consists of one or more binary neurons that represent the class labels. The activation used was either the **heaviside or the sign functions**.

For **regression** tasks, the output layer of an MLP typically consists of a single continuous-valued neuron that represents the predicted output. The most commonly used activation function in the output layer was the **linear activation function**.

Multilayer Perceptron (3)

Training Issues

At first, researchers tried to train MLPs with a gradient learning method called **delta rule**. This method works well with single layer MLP.

However, the delta rule had several limitations that made it difficult to train MLPs effectively : it was **slow**, **computationally expensive** and could get stuck in **suboptimal solutions**.

It was not until the introduction of **backpropagation** in the 1980s that the training of MLPs was made possible efficiently.

Training

In order to train an MLP, you need to:

- Create the **architecture**
- Initialize the **weights**
- Choose the **activation functions**
- Choose a **loss function**
- Choose an **optimization algorithm**

These are the very basic requirements as training and especially optimizing a MLP will require much more steps.

MLP

Activation Functions

Definition

An activation function in DNNs is a mathematical function that is **applied to the output of each neuron** in a layer to **introduce non-linearity** into the model.

The activation function transforms the input signal into an output signal and determines whether or not the neuron will be activated, meaning it will contribute to the output of the network.

Properties

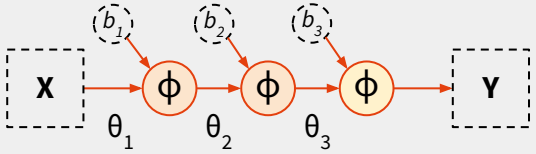
An activation function should/mut include the following properties:

- **Non-linearity** - *mandatory (excepted for the input and output layers)*
- **Continuously differentiable** - *mandatory*
- **Monotonic** - *preferred*
- **Bounded outputs** - *preferred*
- **Computationally efficient** - *preferred*

The **heaviside** and the **sign** activation functions did not met some of these desirable properties. Thus, other activation have been used.

Non-Linearity

One of the most important properties is **Non-linearity**. It means that the relationship between the inputs and the outputs of the model is **not a straight line**. Without activation functions, DNNs would essentially be a linear function of the input, regardless of the number of layers. This would limit the capacity of the model to learn complex patterns in the data.



Without any activation function
ϕ(x) = x

$$
\begin{aligned}
Y &= \Phi(\theta_3\Phi(\theta_2\Phi(\theta_1 X + b_1) + b_2) + b_3) \\
&= \theta_3(\theta_2(\theta_1 X + b_1) + b_2) + b_3 \\
&= \theta_3\theta_2\theta_1 X + \theta_3\theta_2 b_1 + \theta_3 b_2 + b_3 \\
&= \Theta X + B
\end{aligned}
$$

Continuously differentiable

Activation functions must be continuously differentiable because of the **optimization algorithm** used in training DNNs, namely **backpropagation**. Indeed, the backpropagation uses the gradients, i.e. partial derivatives, to update the weights of a DNN.

Functions such as the heaviside and the sign are not differentiable because they are not continuous.

Monotonic

A **monotonic** activation function ensures that increasing the input to a neuron will always result in an increase in the output, and decreasing the input will always result in a decrease in the output.

This property is useful to **ensure that the model learns meaningful representations** of the input.

While monotonicity is a desirable property, it is **not strictly necessary** for all activation functions.

Bounded output

The range of output values should match the requirements of the problem:

	Minimum	Maximum
Regression	Any*	
Binary Classification	0	1
Multi-class Classification	$p(c_j) = 1 - \sum_{i \neq j} p(c_i)$	
Multi-label Classification	0	1

**it is generally better to standardize the labels of a regression task*

Activation Functions

The choice of the activation function depends on the **task problem type** (regression or classification) and the **position of the neuron in the network** (hidden layer or output layer).

Some of the most commonly used ones include:

- Linear
- Sigmoid
- TanH
- ReLU and its variants
- Softmax

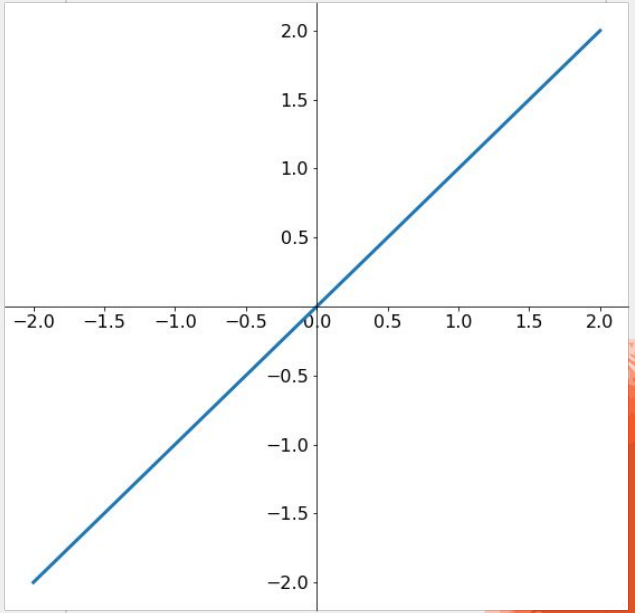
Linear

$$\Phi(z) = z$$

The **linear** activation function is the identity function. It does nothing on the inputs.

It does not meet the nonlinearity criterion. Thus, it **should not be used inside of the neural network**.

However, it can be useful for **Regression task** where it is used as the **output layer activation function**.



Linear (2)

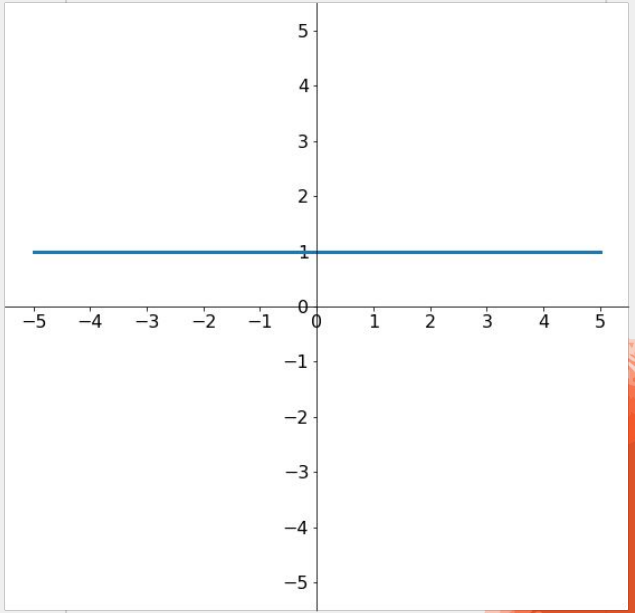
Derivative

$$\frac{\partial \Phi(z)}{\partial z} = 1$$

The derivative of the **linear** activation function is a **constant** which means that it does not change with respect to the input to the activation function.

This leads to two problems:

- It can lead to the **vanishing gradient problem**.
- The **nonlinearity** property is not met.



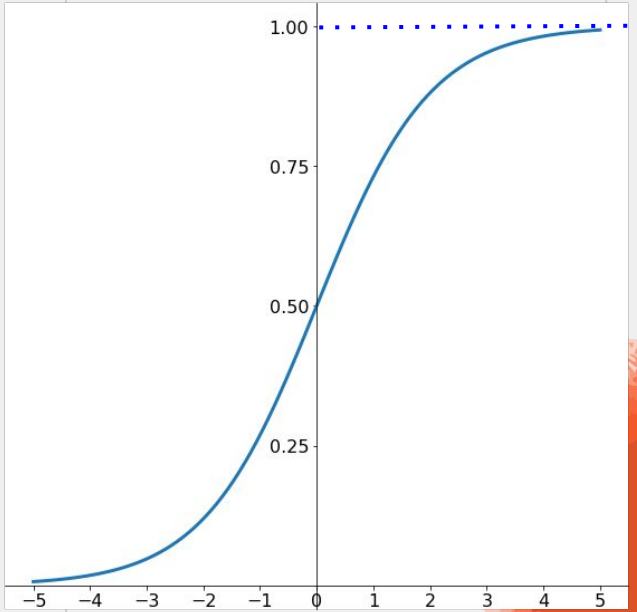
Sigmoid

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

The **sigmoid** activation function maps input values to a range between 0 and 1, which can be interpreted as **probabilities**.

It is used for **Binary** and **multi-label classification tasks** as the **output layer activation function**.

It is not much used as the hidden layer activation function because it can lead to **slow convergence**, to **the vanishing gradient problem** and **underfitting**.



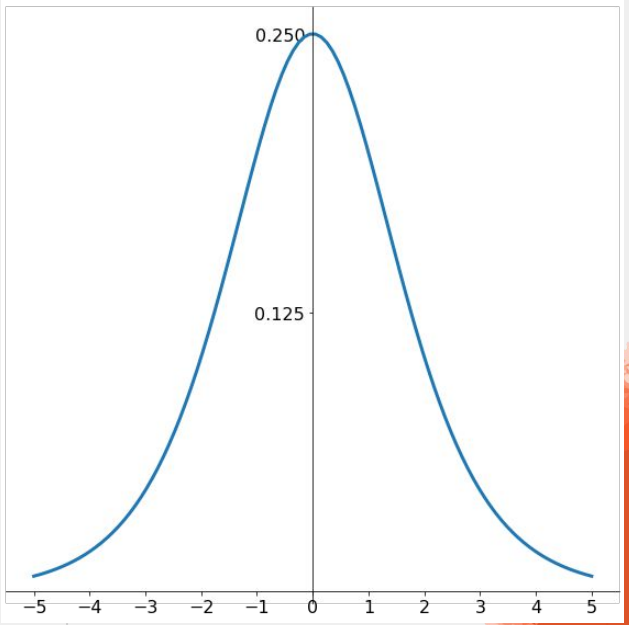
Sigmoid (2)

Derivative

$$\frac{\partial \Phi(z)}{\partial z} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

The **sigmoid derivative** has has a well-defined range between 0 and 0.25, which means that it can **prevent the gradients from exploding**. Moreover, it is smooth and continuous which helps to **avoid sharp transitions in the output** of the network.

However, the derivative of the sigmoid function is close to zero for large or small values of the input which causes the **problem of vanishing gradients**.



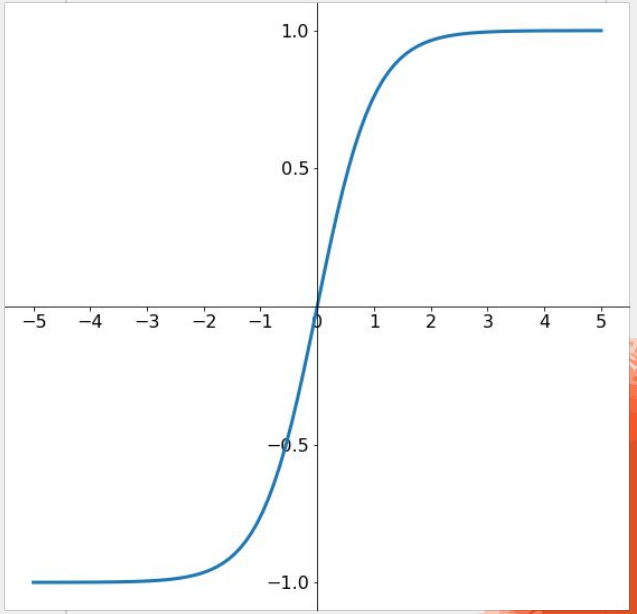
TanH

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

The **TanH** function is similar to the sigmoid function, but maps input values to a range between -1 and 1.

It is **used as the hidden layer activation function**.

It has a **faster convergence** than the sigmoid but it **still suffers from the the vanishing and exploding gradient problem**. It also suffers from the problem of **saturation** because its values can quickly reach a plateau.



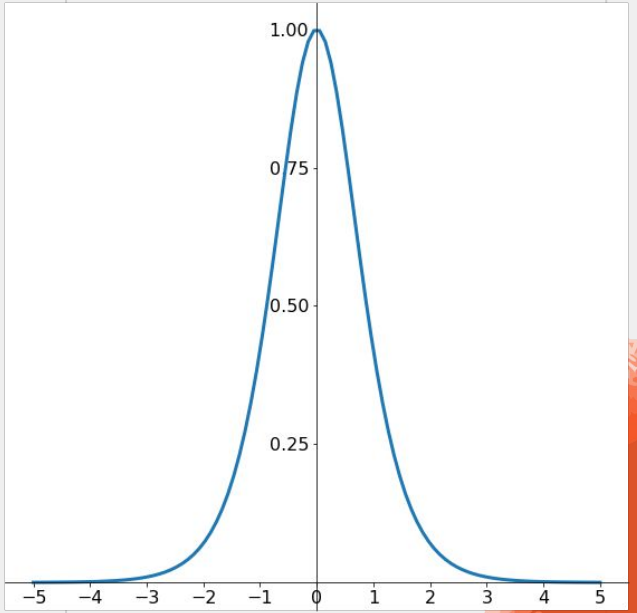
TanH

Derivative

$$\frac{\partial \Phi(z)}{\partial z} = 1 - tanh(z)^2$$

The **TanH derivative** has the same advantages as the sigmoid derivative but is steeper for inputs close to zero which can help the network to **learn faster** in the regions where the output is changing the most.

However, like the sigmoid activation function, the tanh activation function can also **suffer from the problem of vanishing gradients**.



ReLU

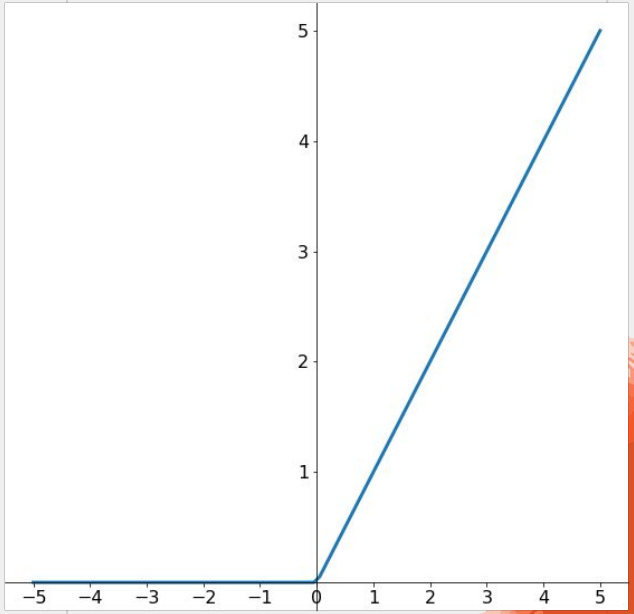
$$\Phi(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

The **ReLU** (Rectified Linear Unit) function is defined as the positive part of its argument.

It is the most commonly **used hidden layer activation function**.

It is a very **efficient**, **fast** and **non-saturating** activation function that can help to **alleviate the problem of vanishing gradients**.

However, it suffers from the **dying ReLU problem**.



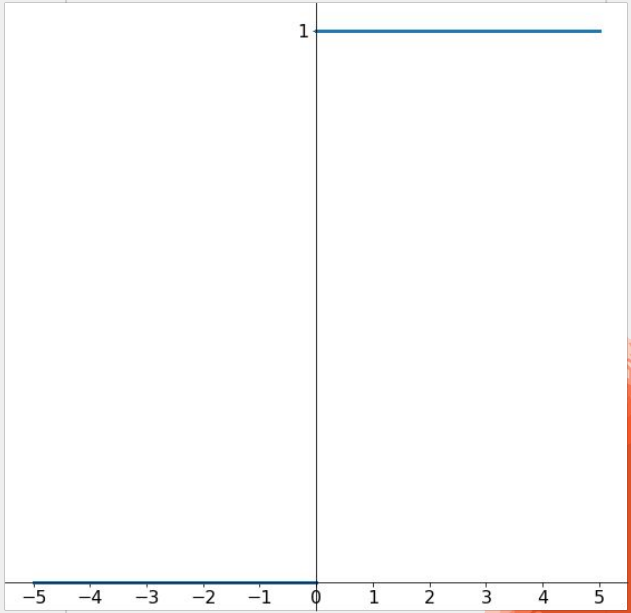
ReLU

Derivative

$$\frac{\partial \Phi(z)}{\partial z} = \begin{cases} 1 & \text{if } z <= 0 \\ 0 & \text{if } z > 0 \end{cases}$$

The ReLU derivative is computationally **very efficient** and **does not suffer from the problem of vanishing gradients**.

However, the derivative can get "stuck" at 0 which leads to the problem of **"dying ReLU"**. The parameters affected by this problem do not learn anymore during training.

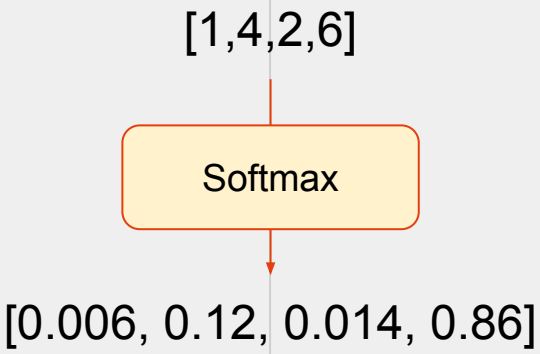


Softmax

The **Softmax** function transforms the inputs into a **probability distribution** that sums to one over the output classes.

It is only used for **multi-class classification** tasks as the **output layer activation function**.

$$\Phi(z_i) = \frac{e^{z_i}}{\sum_j^n e^{z_j}}$$



Usages

In MLPs

	Hidden Layer	Output Layer	
		Regression	***Classification***
Linear	Forbidden	Yes	No
Sigmoid	Rarely	Rarely	Binary & Multi-label
Softmax	Forbidden	No	Multi-Class
TanH	Situational	Rarely	No
ReLU	Yes	Rarely	No

MLP

Loss Functions

Definition

A **loss function**, also known as a cost function, objective function, or error function, is a mathematical function that **measures the difference between the predicted values and the actual values** in a machine learning model. This measure is then used to improve the model.

Loss functions should generally be:

- Continuous and differentiable
- Robust to outliers
- Meaningful, and interpretable to some extent
- Computationally efficient

History

The use of loss functions in deep neural networks (DNNs) can be traced back to the early days of neural networks in the 1950s and 1960s, with researchers using simple error functions such as **mean squared error** to train networks.

From the 1980s to the early 2000s, researchers explored various forms of error functions such as **cross-entropy**, **hinge loss** and more sophisticated loss functions such as the **Kullback-Leibler** divergence.

Since then, researchers have developed various **specialized loss functions** for specific applications such as the focal loss, the dice loss or the negative sampling loss.

Regression Losses

In a regression task for a DNN, the goal is to **predict a continuous numerical output** based on a set of input features.

The most common loss functions for regression:

- **Mean Squared Error** Loss - *the most common loss for regression.*
- **Mean Absolute Error** Loss - *a situational loss for regression.*
- **Huber** Loss - *a combination of the MSE and the MAE.*
- **Log Cosh** Loss - *a smooth approximation of Huber loss.*

Regression Losses (2)

Mean Squared Error *Loss*

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2$$

Mean Absolute Error *Loss*

$$MAE = \frac{1}{n}\sum_{i}^{n}|y_i - \hat{y}_i|$$

Huber *Loss*

$$L_\delta = \frac{1}{n}\sum_{i}^{n}\begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| <= \delta \\ \delta\left(|y_i - \hat{y}_i| - \frac{\delta}{2}\right) & \text{if } |y_i - \hat{y}_i| > \delta \end{cases}$$

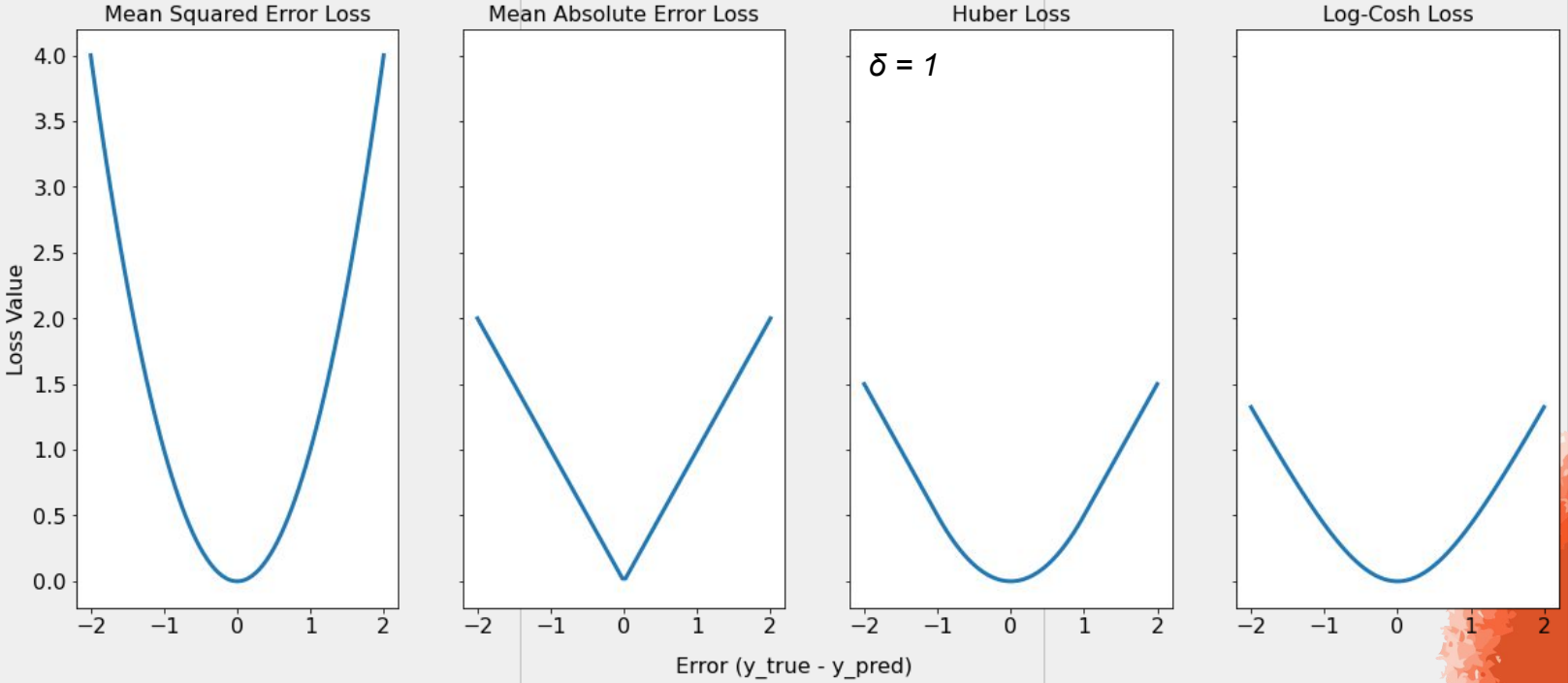
δ controls the range of values considered as outliers

Log-Cosh *Loss*

$$LogCosh = \frac{1}{n}\sum_{i}^{n} log\left(cosh\left(y_i - \hat{y}_i\right)\right)$$

Regression Losses (3)

Comparison



Regression Losses (4)

Pros & Cons

	Pros	Cons	Usage
MSE	• Amplifies large errors • Simple/fast to compute	• Sensitive to outliers	Useful when outliers are rare but need to be taken into account
MAE	• Less sensitive to outliers • Simple/fast to compute	• Difficult to optimize due to its non-differentiability at zero	Useful when outliers are common, when the data is very noisy
Huber	• Less sensitive to outliers than MSE • More robust than MAE	• Difficult to optimize due to the tuning of δ • Not great for data that don't follow a normal distribution • Slower than MSE and MAE	Useful when outliers exist, but most of the data follows a normal distribution
Log Cosh	• Less sensitive to outliers than MSE • Easier/faster to optimize than Huber	• Less intuitive interpretation • Not great for data that don't follow a normal distribution • Slower than MSE and MAE	

Classification Losses

In a classification task for a DNN, the goal is to **assign input data to one of several predefined categories** based on their features. Generally, the loss function maps the input features to a probability distribution over the output categories.

The most common loss functions for classification:

- **Cross-Entropy** Loss - *the most common loss for classification.*
- **Hinge** Loss - *it maximizes the margin between different classes.*
- **Focal** Loss - *it addresses class imbalance problems in multi-class classification tasks.*

Classification Losses (2)

Cross-Entropy

The basic idea behind cross-entropy loss is to **measure the difference between the predicted probability distribution and the true probability distribution of the output classes**. It can be defined as the negative logarithm of the predicted probability of the true class:

$$H = -\sum_{i}^{n}\left(\sum_{c}^{M} y_{i,c} log(p_{i,c})\right)$$

with :

n the number of instances
M the number of classes
$\mathbf{y_{i,c}}$ the true probability of the class c
$\mathbf{p_{i,c}}$ the predicted probability of the class c

Classification Losses (3)

(Sparse) Categorical Cross-Entropy

Categorical Cross-entropy is used where there are **more than two output classes** and each instance can belong to **two or more classes**.

Sparse Categorical Cross-entropy is used where there are **more than two output classes** and **each instance belongs to one class**.

***Categorical** Cross-Entropy*

	Features	Class 1	Class 2	Class 3
Instance 1	…	0	0	1
Instance 2		1	1	0
Instance 3		0	1	1
Instance 4		0	1	0

***Sparse Categorical** Cross-Entropy*

	Features	Class 1	Class 2	Class 3
Instance 1	…	0	0	1
Instance 2		1	0	0
Instance 3		0	0	1
Instance 4		0	1	0

Classification Losses (3)

Binary Cross-Entropy

Binary Cross-entropy is used for binary classification problems, where there are only **two output classes**. The loss is defined as the negative logarithm of the predicted probability of the true class:

$$H_{binary} = -\sum_{i}^{n} (y_i log(p_i) + (1 - y_i) log(1 - p_i))$$

When $y = 0$, then H_{binary} is equal to $log(1-p_i)$ for the instance i

When $y = 1$, then H_{binary} is equal to $log(p_i)$ for the instance i

Classification Losses (4)

Hinge Loss

Hinge loss is a loss function commonly used in machine learning, including for classification tasks in DNN. It is useful in classification tasks where the emphasis is on correctly classifying the “hard” examples.

$$L = \frac{1}{n}\sum_{i}^{n} max(0, 1 - y_i\hat{y}_i)$$

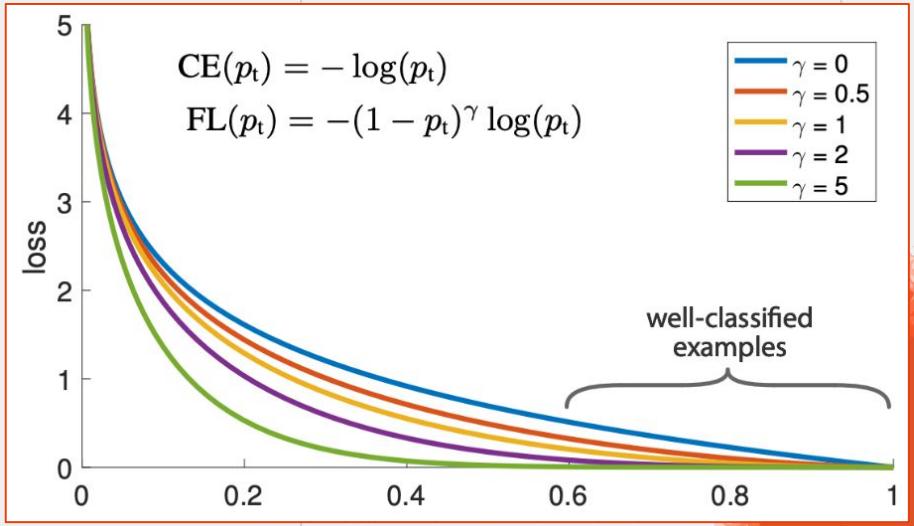
For the Hinge Loss, the true output y_i must be either -1 or 1, not 0 or 1.

Classification Losses (5)

Focal Loss

Focal loss is a loss function that is designed to **address class imbalance problems** in multi-class classification tasks by reducing the loss contribution of the well-classified instances.

$$L = -\sum_{i}^{n}\left(\sum_{c}^{M}(1-p_{i,c})^{\gamma}y_{i,c}log(p_{i,c})\right)$$



Source : Focal Loss for Dense Object Detection, Tsung-Yi 2017

Classification Losses (6)

Pros & Cons

	Pros	Cons	Usage
Cross-Entropy	• Simple/fast to compute	• Sensitive to class distribution • Sensitive to outliers	Useful for classification problems with balanced class distributions
Hinge	• Less sensitive to outliers • Less sensitive to class distribution	• Difficult to optimize due to its non-differentiability at zero	Useful for binary classification problems with outliers or unbalanced classes
Focal	• Less sensitive to class distribution • Handle well the hard instances	• Computationally expensive • Sensitive to the choice of γ	Useful for multi-class classification problems with imbalanced class distributions or hard examples

Specialized Losses

Specialized losses for neural networks refer to loss functions that are **specifically designed for certain types of tasks or data**.

These losses can be useful in improving the performance of neural networks in specific domains, and are often **developed through empirical observation and experimentation**. Some of these losses includes the Dice Loss (for image segmentation), the Triplet Loss (for Few Shot Learning), the Wasserstein loss (for GANs) …

Additionally, you can create your own loss functions. However, you should **never reinvent the wheel** as some researchers have probably already found great specialized loss functions for your task.

MLP

Optimization Algorithm

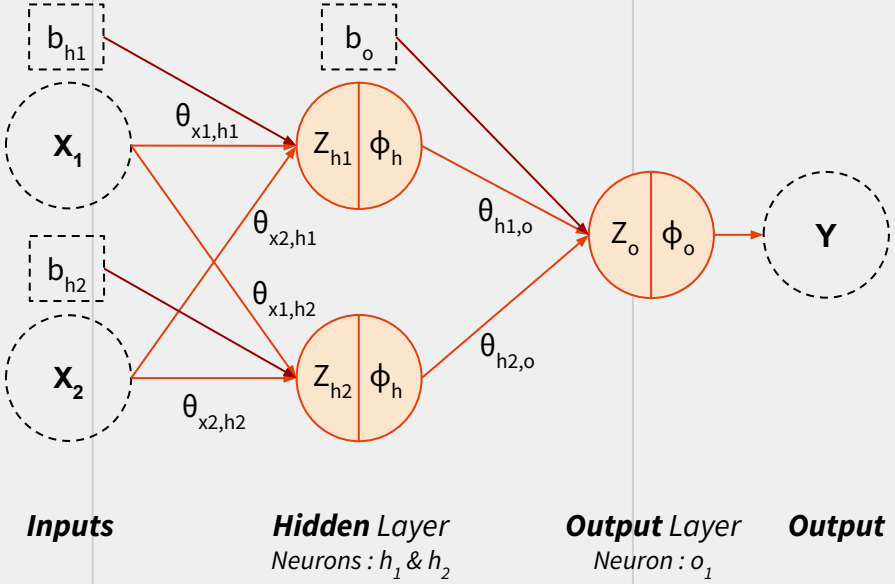
Backpropagation

Backpropagation is a widely used **algorithm** for training neural networks. It involves computing the gradient of the network's loss function with respect to the weights of the network using the **chain rule of calculus**. The gradient is then used to update the weights of the network through an **optimization algorithm such as the Gradient Descent**.

It follows 4 steps:

1. Forward Pass
2. Error measurement
3. Backward pass
4. Weight update (*e.g. Gradient Descent*)

Example



***Hidden** Layer*
Activation Function

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

***Output** Layer*
Activation Function

$$\Phi(z) = z$$

***Loss** Function*

$$MSE = \frac{1}{n}\sum_{i}^{n}(y_i - \hat{y}_i)^2$$

Gradient Descent

Gradient descent is an **optimization algorithm** that can be used to **update the parameters** of a neural network in order to minimize the loss function.

$$\theta_{i,t+1} = \theta_{i,t} - \eta \frac{\partial L}{\partial \theta_{i,t}}$$

It iteratively computes and subtracts the **gradient of the loss function** with respect to the weights. This process continues until a minimum of the loss function is reached, which corresponds to the optimal set of parameters for the model. The hyperparameter η, also known as the **learning rate**, controls the amount of gradient used to update the weights.

Forward Pass

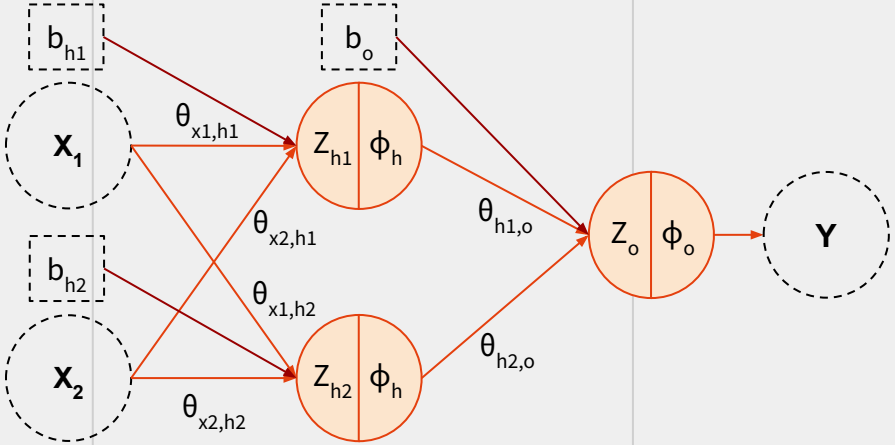
Definition

The forward pass of backpropagation refers to the **computation of the output of a neural network for a given input**. During the forward pass, the input is fed into the network, and a series of matrix multiplications and non-linear transformations are performed to produce the final output.

It is exactly like a prediction step except that **all the intermediate results are preserved** for the backward pass.

Forward Pass (2)

An example



$$h_1 = \Phi_h(X_1\theta_{x_1,h_1} + X_2\theta_{x_2,h_1} + b_{h1}) = \Phi_h(Z_{h_1})$$

$$h_2 = \Phi_h(X_1\theta_{x_1,h_2} + X_2\theta_{x_2,h_2} + b_{h2}) = \Phi_h(Z_{h_2})$$

$$o = \Phi_o(h_1\theta_{h_1,o} + h_2\theta_{h_2,o} + b_o) = \Phi_o(Z_o) = Y$$

Loss Function

The **error measurement** is done using the loss function and the output of the network calculated during the forward pass. In the following example, the loss function is the Mean Squared Error.

$$
\begin{aligned}
L(y, \hat{y}) &= \frac{1}{n} \sum_{i}^{n} (y_i - \hat{y}_i)^2 \\
&= \frac{1}{n} \sum_{i}^{n} (y_i - o_i)^2 \\
&= \frac{1}{n} \sum_{i}^{n} (y_i - \Phi_o(Z_o))^2
\end{aligned}
$$

Backward Pass

Definition

The **backward pass** of backpropagation refers to the process of **computing the gradients of the weights** of a neural network with respect to the loss function. This is done by **propagating the error from the output layer back through the network to the input layer**.

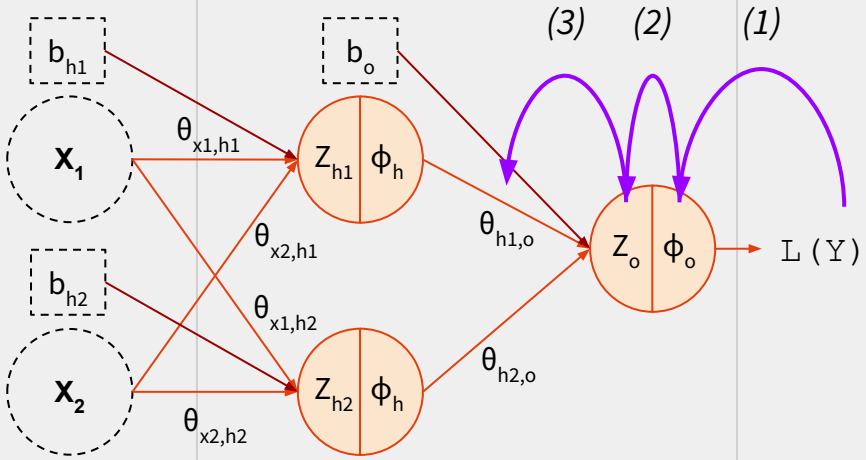
This step is based on the **chain rule** property of the partial derivatives:

$$\frac{\partial FoGoH(z)}{\partial z} = \frac{\partial FoGoH(z)}{\partial GoH(z)} \frac{\partial GoH(z)}{\partial H(z)} \frac{\partial H(z)}{\partial z}$$

with `F`, `G` and `H` three differentiable functions

Backward Pass (2)

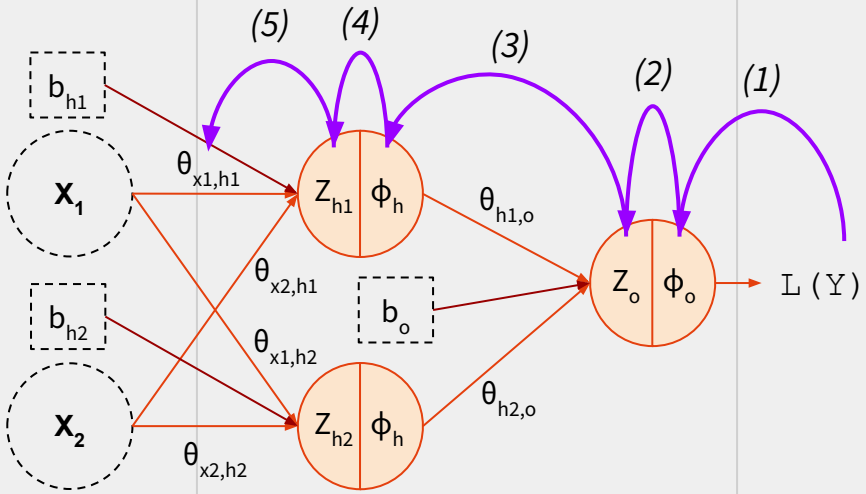
An example



$$\frac{\partial L}{\partial \theta_{h_1,o}} = \overset{(1)}{\frac{\partial L}{\partial \Phi_o(Z_o)}} \overset{(2)}{\frac{\partial \Phi_o(Z_o)}{\partial Z_o}} \overset{(3)}{\frac{\partial Z_o}{\partial \theta_{h_1,o}}}$$

Backward Pass (3)

Another example



$$\frac{\partial L}{\partial \theta_{x_1,h_1}} = \underbrace{\frac{\partial L}{\partial \Phi_o(Z_o)}}_{(1)} \underbrace{\frac{\partial \Phi_o(Z_o)}{\partial Z_o}}_{(2)} \underbrace{\frac{\partial Z_o}{\partial \Phi_h(Z_{h_1})}}_{(3)} \underbrace{\frac{\partial \Phi_h(Z_{h_1})}{\partial Z_{h_1}}}_{(4)} \underbrace{\frac{\partial Z_{h_1}}{\partial \theta_{x_1,h_1}}}_{(5)}$$

Backward Pass (4)

Another example

(1)
$$\frac{\partial L}{\partial \Phi_o(Z_o)} = \frac{\partial (y - \Phi_o(Z_o))^2}{\partial \Phi_o(Z_o)} = -2(y - \Phi_o(Z_o))$$

(2)
$$\frac{\partial \Phi_o(Z_o)}{\partial Z_o} = \frac{\partial Z_o}{\partial Z_o} = 1$$

(3)
$$\frac{\partial Z_o}{\partial \Phi_h(Z_{h_1})} = \frac{\partial(\theta_{h_1,o} h_1 + \theta_{h_2,o} h_2 + b_o)}{\partial h_1} = \theta_{h_1,o}$$

(4)
$$\frac{\partial \Phi_h(Z_{h_1})}{\partial Z_{h_1}} = \frac{\partial \left(\frac{1}{1+e^{-Z_{h_1}}}\right)}{\partial Z_{h_1}} = \frac{1}{1+e^{-Z_{h_1}}}\left(1 - \frac{1}{1+e^{-Z_{h_1}}}\right)$$

(5)
$$\frac{\partial Z_{h_1}}{\partial \theta_{x_1,h_1}} = \frac{\partial \left(\theta_{x_1,h_1} X_1 + \theta_{x_2,h_1} X_2 + b_{h_1}\right)}{\partial \theta_{x_1,h_1}} = X_1$$

Memoization

Similarly to the forward pass, the **backward pass can reuse the calculated gradients** (see the two previous examples) to compute the next ones. This makes the backpropagation very computationally efficient.

$$\frac{\partial L}{\partial \theta_{h_1,o}} = \frac{\partial L}{\partial \Phi_o(Z_o)} \frac{\partial \Phi_o(Z_o)}{\partial Z_o} \frac{\partial Z_o}{\partial \theta_{h_1,o}}$$

$$\frac{\partial L}{\partial \theta_{x_1,h_1}} = \frac{\partial L}{\partial \Phi_o(Z_o)} \frac{\partial \Phi_o(Z_o)}{\partial Z_o} \frac{\partial Z_o}{\partial \Phi_h(Z_{h_1})} \frac{\partial \Phi_h(Z_{h_1})}{\partial Z_{h_1}} \frac{\partial Z_{h_1}}{\partial \theta_{x_1,h_1}}$$

Vanishing-Exploding Gradient problem

Definition

The **vanishing/exploding gradient problem** is a common issue that can occur when training deep neural networks. It arises from the fact that **during the backpropagation algorithm**, the gradients of the loss function with respect to the weights of the network are computed and used to update the weights.

These **gradients can become very small (vanishing gradient) or very large (exploding gradient) as they are propagated back through the layers of the network**, making it difficult to train the model effectively.

Cause of the problem

The **vanishing gradient** problem occurs **when the absolute value of the activation function derivative has regions close to zero**.

Similarly, the **exploding gradient** descent problem **occurs when the activation function has regions "high" value regions**.

$$\frac{\partial L}{\partial \theta} = \frac{\partial L}{\partial \Phi_n(Z_n)} \frac{\partial \Phi_n(Z_n)}{\partial Z_n} \frac{\partial Z_n}{\partial \Phi_{n-1}(Z_{n-1})} \frac{\partial \Phi_{n-1}(Z_{n-1})}{\partial Z_{n-1}} \frac{\partial Z_{n-1}}{\partial \Phi_{n-2}(Z_{n-2})} \frac{\partial \Phi_{n-2}(Z_{n-2})}{\partial Z_{n-2}} \dots$$

The partial derivatives of the activation functions are chained which can lead to the exploding/vanishing gradient problem when the network is large.

Solutions

Fortunately, researchers found several solutions to the vanishing and exploding gradient problem in deep neural networks:

- Using a better **weight initialization** method
- Using **alternative activation functions**
- Using **gradient clipping** methods
- Using **batch normalization**
- Building networks with **skip connections**
- Creating new types of neurons

Weight Initialization

In their 2010 paper titled *"Understanding the Difficulty of Training Deep Feedforward Neural Networks"*, Glorot and Bengio showed that the choice of weight initialization can have a significant impact on the vanishing/exploding gradient problem in deep neural networks.

They observed that:

- if the weights of a layer are initialized with a **variance that is too high**, the activations of the neurons in the layer can quickly become very large, causing the **gradient to explode**.
- if the weights of a layer are initialized with a **variance that is too low**, the activations can become very small, causing the **gradient to vanish**.

Glorot Initialization

The **Glorot initialization** is a popular weight initialization method that sets the weights to be sampled from a **Normal distribution** with zero mean and variance that depends on the number of inputs and outputs of the layer:

$$\sigma^2 = \frac{2}{fan_{in} + fan_{out}}$$

where $\texttt{fan}_{\texttt{in}}$ and $\texttt{fan}_{\texttt{out}}$ are the number of input and output neurons.

The Glorot initialization also exists in a **Uniform distribution**:

$$\left[-\sqrt{\frac{6}{fan_{in} + fan_{out}}}, +\sqrt{\frac{6}{fan_{in} + fan_{out}}}\right]$$

Glorot Initialization (2)

The Glorot initialization is generally a **good choice for neural networks with sigmoidal or hyperbolic tangent activation functions** but it may not be a good choice for ReLU and its variants because these activation functions are highly nonlinear and have a large variance.

When the Glorot/Xavier initialization is used with ReLU and its variants, the weights may be too small, which can cause the network to have a smaller than optimal capacity. This can result in underfitting, where the network is not able to capture the complexity of the data.

He Initialization

To mitigate the problem of the Glorot initialization with the ReLU activation function, the **He initialization** was introduced. Its **Normal distribution** form uses a larger variance than the Glorot initialization:

$$\sigma^2 = \frac{2}{fan_{in}}$$

The He initialization also exists in a **Uniform distribution**:

$$\left[-\sqrt{\frac{6}{fan_{in}}}, \sqrt{\frac{6}{fan_{in}}}\right]$$

LeCun Initialization

The LeCun Initialization is a special case of the Glorot initialization where:

$$fan_{in} = fan_{out}$$

This initialization is generally used in convolutional neural networks and paired with the SELU* (Scaled Exponential Linear Unit).

** see later in this chapter*

Initialization Usages

	Glorot	He	LeCun
Sigmoid	Yes	-	-
Softmax	Yes	-	-
TanH	Yes	-	-
ReLU and its variants *excepted SELU*	-	Yes	-
SELU	-	-	Yes

Activation Functions

The ReLU has a derivative that is either zero or one, depending on the input value. This means that the gradients do not become small for positive inputs, and the **vanishing gradient problem is mitigated**. However, the derivative of ReLU is zero for negative inputs, which can cause the **dying ReLU problem**, where some neurons "die" and do not fire again during training.

To solve this problem, multiple variants have been created including the **Leaky ReLU** that was later improved into the **ELU** which was also later improved by the **SELU** variant.

Activation Functions (2)

Leaky ReLU

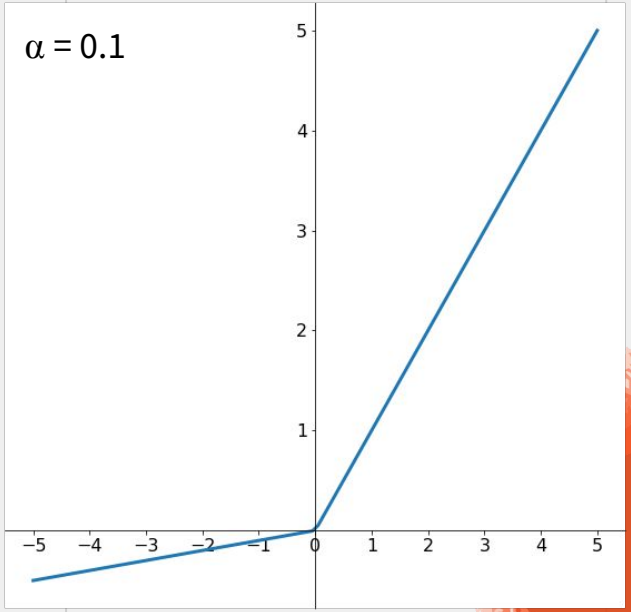
$$\Phi(z) = max(\alpha z, z)$$

Leaky ReLU is a variant of the ReLU activation function that is designed to **address the dying ReLU problem**.

The value of α is typically set to a small value, such as 0.1, to ensure that the slope is small enough that it does not negatively affect the performance of the network.

Its derivative is:

$$\frac{\partial \Phi(z)}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ \alpha & \text{if } z \leq 0 \end{cases}$$



Activation Functions (3)

Leaky ReLU variants

The Leaky ReLU has two main variants:

- The **Randomized Leaky ReLU** (RReLU) - the value of α is randomly sampled from a uniform distribution (*generally between 0 and 1*) during training and fixed to the median of the distribution during testing.
- The **Parametric Leaky ReLU** (PReLU) - the value of α is learned during the training. It is computationally more expensive than the Leaky ReLU and the RReLU.

Activation Functions (4)

ELU

In 2015, Clervert and his team introduced a new variant: the **Exponential Linear Unit** activation function, or ELU. It has a few advantages over the ReLU and Leaky ReLU activation functions:

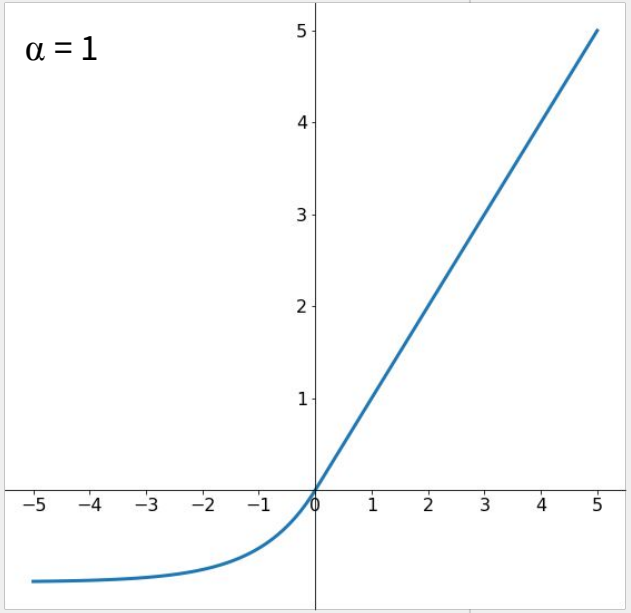
- It is **smooth everywhere**, which makes it easier to optimize with gradient-based methods.
- It is **not equal to 0 when it takes negative values**, which helps to avoid the dying ReLU problem.
- It has a **mean activation that is close to zero**, which can help to speed up convergence and improve the performance of the network.

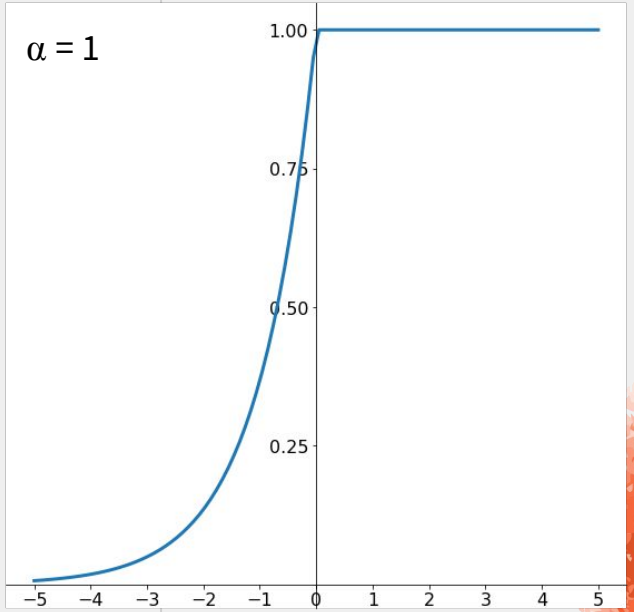
Activation Functions (5)

ELU - Formula & derivative

$$\Phi(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha\left(e^z - 1\right) & \text{if } z \leq 0 \end{cases}$$

$$\frac{\partial \Phi(z)}{\partial z} = \begin{cases} 1 & \text{if } z > 0 \\ \alpha e^z & \text{if } z \leq 0 \end{cases}$$





Activation Functions (6)

SELU

In 2017, Klambauer and his team introduced a variant of the ELU: the **Scaled Exponential Linear Unit** activation function, or SELU. It has a **self-normalizing property**, which means that the output of each layer in the network preserves a mean of 0 and a standard deviation of 1 throughout training, even in deep networks with many layers. This property completely **solves the vanishing and exploding gradient problem**.

The SELU often significantly outperforms the other activation functions but it has a few constraints:

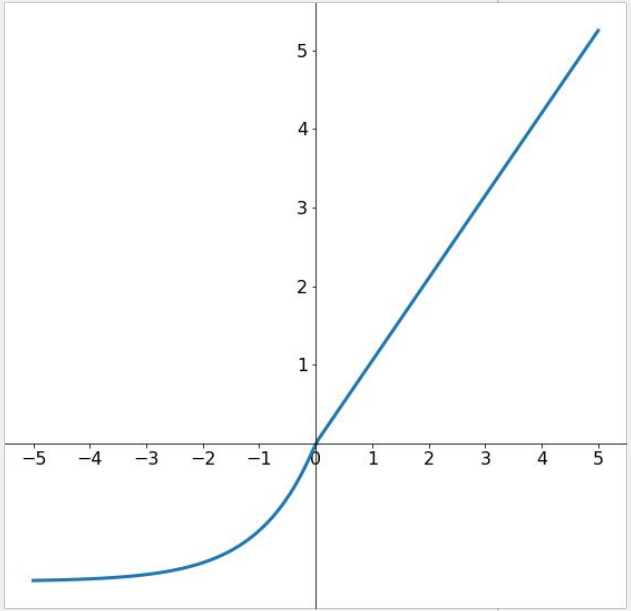
- The inputs must be standardized
- The LeCun weight initialization must be used
- The network must be sequential: it can’t be used in RNNs or in networks with skip connections

Activation Functions (7)

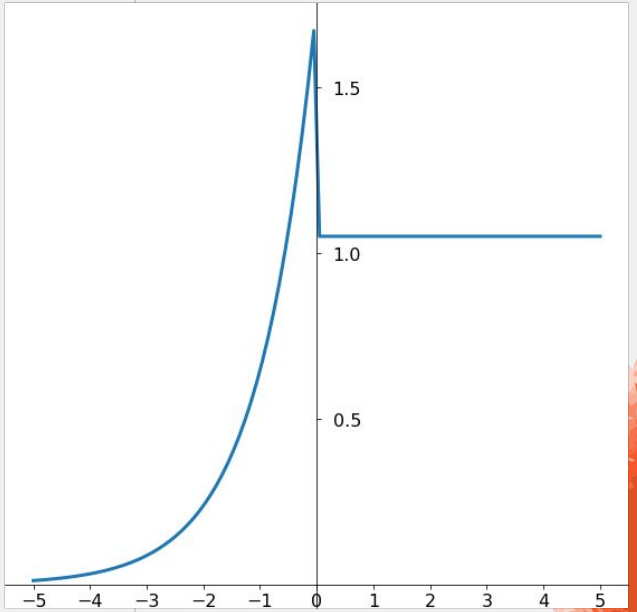
SELU - formula & derivative

$$\Phi(z) = \lambda \begin{cases} z & \text{if } z > 0 \\ \alpha\left(e^z - 1\right) & \text{if } z \leq 0 \end{cases}$$

$$\frac{\partial \Phi(z)}{\partial z} = \lambda \begin{cases} 1 & \text{if } z > 0 \\ \alpha e^z & \text{if } z \leq 0 \end{cases}$$



$\alpha = 1.67326324$
$\lambda = 1.05070098$



Activation Functions (8)

Comparison

	Dying ReLU	Vanishing Gradient	Exploding Gradient	Performances
ReLU	Sensitive	-	Sensitive	/
Leaky ReLU	-	-	Sensitive	Usually better than the ReLU
RReLU	-	-	Sensitive	Better than Leaky ReLU for overfitting problems
PReLU	-	-	Sensitive	Better than Leaky ReLU if there is a lot of data
ELU	-	-	Sensitive	Better but slower than ReLU, Leaky ReLU, RReLU and PReLU
SELU	-	-	-	Better than ELU but it is still slow and has more constraints.

Gradient Clipping

Gradient clipping is a technique used during training of neural networks to **prevent the exploding gradient problem**. It limits the maximum value of the gradients to a specified threshold, which prevents the gradients from becoming too large and helps to stabilize the training process.

However, gradient clipping can also have some negative effects on the training process, such as **slowing down the convergence** of the network or introducing bias in the parameter updates.

Batch Normalization

Batch normalization is a technique used in neural networks to improve the training process and the generalization performance of the network. The technique works by **normalizing the input to each layer of the network**, which helps to stabilize the training process and **reduce the impact of the vanishing/exploding gradient problem**.

Each input of each layer is normalized with the mean μ and the standard deviation σ, and then rescaled with two parameters ɣ (scale) and β (offset):

The inputs of the layer

$$\hat{x} = \frac{x - \mu}{\sigma}$$

$$z = \gamma \hat{x} + \beta$$

Batch Normalization (2)

Training vs Testing

The parameters ɣ (scale) and β (offset) are initialized to 1 and 0, respectively, and are learned during training.

At test time, batch normalization operates slightly differently than during training. Indeed, there is typically only a single input example, which means that it is not possible to compute the mean and variance over a batch.

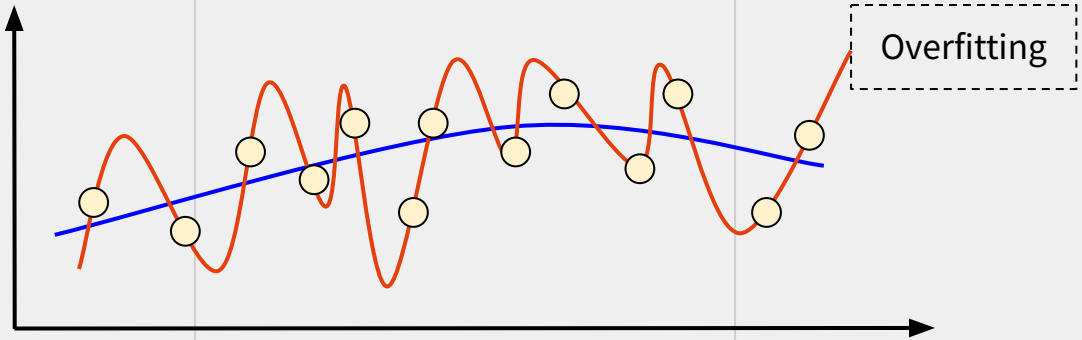
To address this issue, batch normalization typically **uses the population statistics of the mean and variance** that were computed during training. At the end of training, the population statistics of the mean and variance are computed by taking the average of these statistics over the training data.

Control Overfitting

Definition

Overfitting occurs when a model learns to **fit the training data too well**, to the extent that it starts to **memorize the noise in the data** instead of learning the underlying patterns.

This results in poor performance when the model is presented with new data that it has not seen before, as it is unable to generalize its learning beyond the training set.



Definition (2)

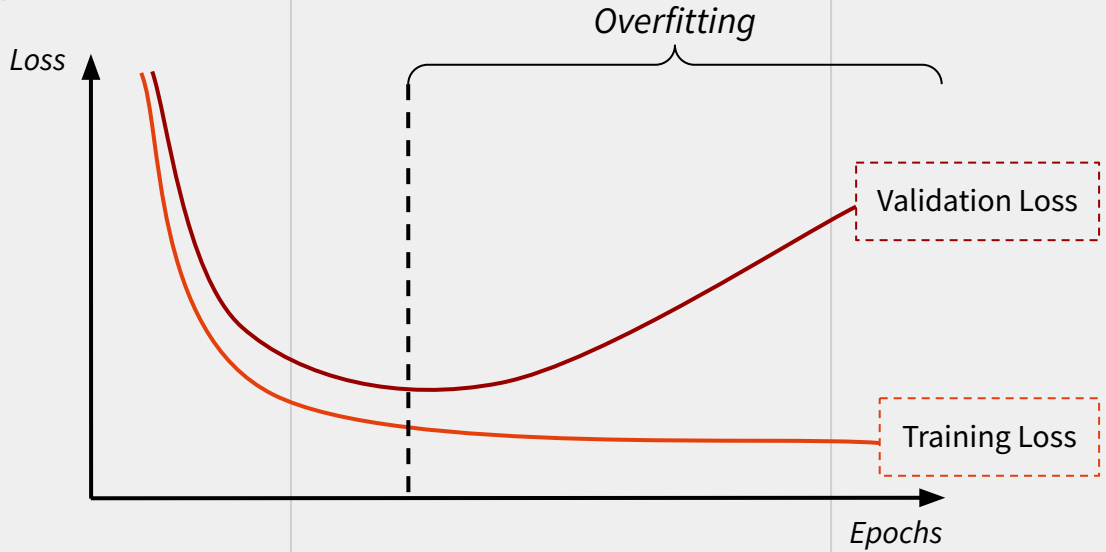
More specifically, overfitting in deep learning can occur **when the model has too many parameters relative to the amount of training data**, leading to a high variance in the model's performance. Overfitting can also occur **when the model is trained for too many epochs**, as it can continue to adjust its parameters to fit the training data even when it has already learned the underlying patterns.

To prevent overfitting in deep learning, various techniques can be used such as:

- Regularization
- Dropout layers
- Early stopping

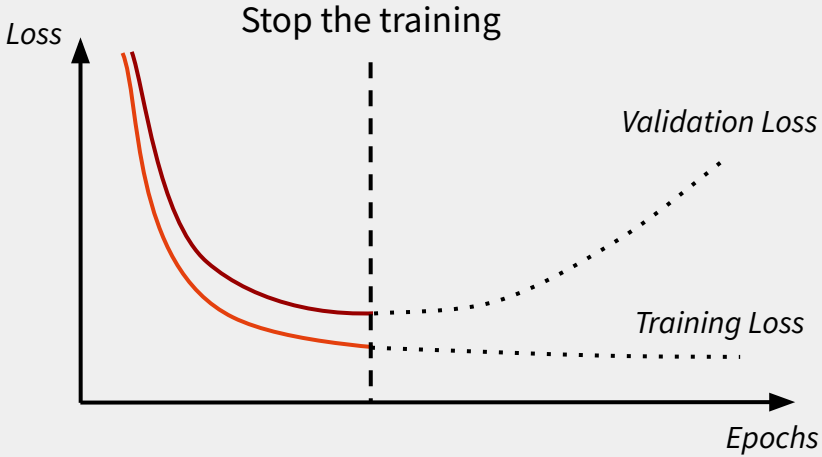
Detection

The overfitting can easily be detected by **analyzing the learning curves**. It is possible to identify overfitting by looking for a **large gap between the loss on the training and validation sets**.



Early Stopping

Early stopping is a regularization technique in deep learning that **monitors the performance of the model on a validation set** during training, and to stop the training process when the performance on the validation set starts to deteriorate.



Regularization

Regularization is a technique used in deep learning to prevent overfitting by **adding a penalty term to the loss function during training**. The penalty term encourages the model to learn simpler and more generalizable patterns by adding a cost to more complex models.

There are two main types of regularization used in deep learning:

- **L1 regularization**, adds a penalty term to the loss function that is proportional to the absolute value of the weights.
- **L2 regularization**, adds a penalty term to the loss function that is proportional to the square of the weights.

Regularization (2)

The regularization term is added to the overall loss function of the model and is controlled by a parameter λ:

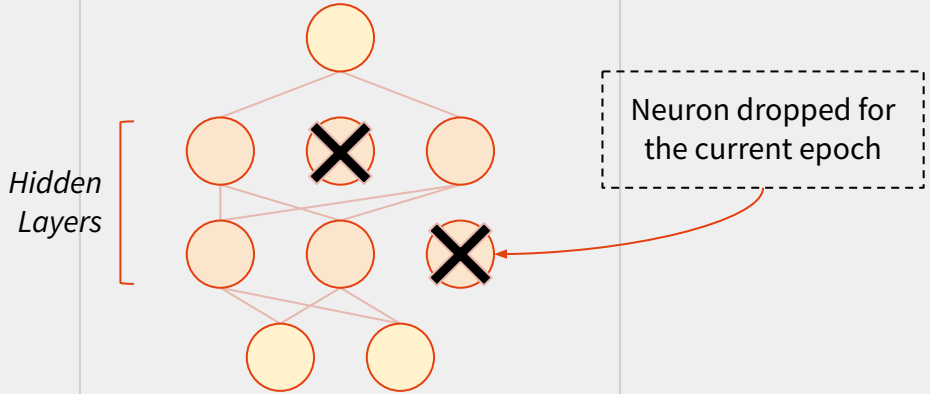
$$\hat{L}(y, \hat{y}) = L(y, \hat{y}) + \lambda R(\theta)$$

The regularization can be specifically added to some layers but a common approach is to apply the same regularizer to all layers.

$$R_{L1}(\theta) = \sum_{i}^{n} |\theta_i| \qquad R_{L2}(\theta) = \sum_{i}^{n} \theta_i^2$$

Dropout

Dropout is a regularization technique that is widely used in deep learning to prevent overfitting. It works by **randomly dropping out** (*i.e. setting to zero*) **a fraction of the neurons in the network during training**. By doing so, the network is forced to learn more robust and generalizable representations that do not rely on specific combinations of nodes being present.



Dropout (2)

Training vs Testing

During training, each neuron in the layer is disabled with a fixed probability, which is typically set to a value between 0.1 (*10% of the neurons are disabled at each epoch*) and 0.5 (*50% of the neurons are disabled at each epoch*).

At test time, the full network is used, but the output of each neuron is scaled by the probability that it was present during training.

Dropout (3)

Pros & cons

Dropout is an effective regularization technique in deep learning because:

- The **remaining neurons are forced to learn more robust and diverse representations** that are less sensitive to the precise configuration of the other neurons.
- The number of possible partial networks is so huge (2^{neurons}) that **it is almost guaranteed to have a different partial network** at each epoch.

However, the dropout **increases the training time** and choosing the wrong rate can lead to either underfitting or overfitting.

Dropout (4)

Guidelines

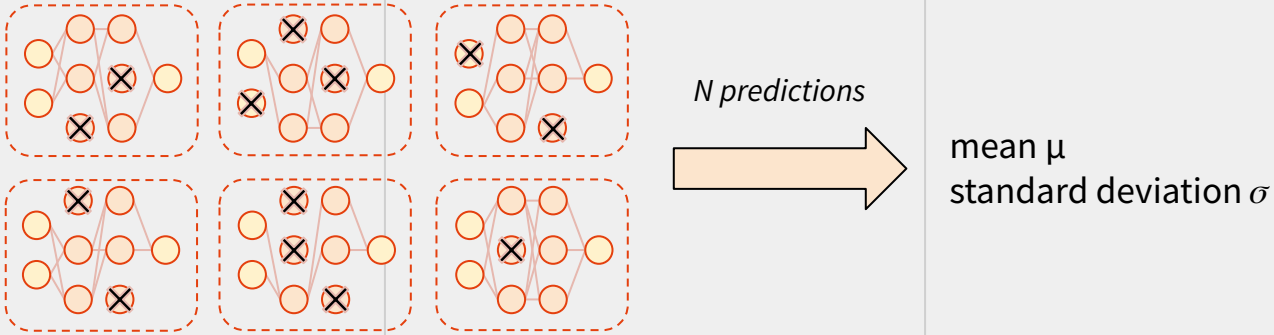
There is no one-size-fits-all rule for the number and position of dropout layers in a network. However, here are some guidelines:

- it is generally recommended to **apply dropout to the fully connected layers** (MLP) of the network
- it is often beneficial to **apply dropout after each layer of the network**, rather than only at the end
- it is recommended to **only use one dropout at the end of the network if the full dropout is too strong** (i.e. underfitting)

Monte Carlo Dropout

Monte Carlo dropout is a technique that extends the use of dropout in deep learning to provide a measure of uncertainty in the model's predictions.

Instead of using dropout only during training to prevent overfitting, **Monte Carlo dropout applies dropout at test time to generate a set of predictions** and estimates the uncertainty of the model's predictions based on the variance of the predictions across the set.



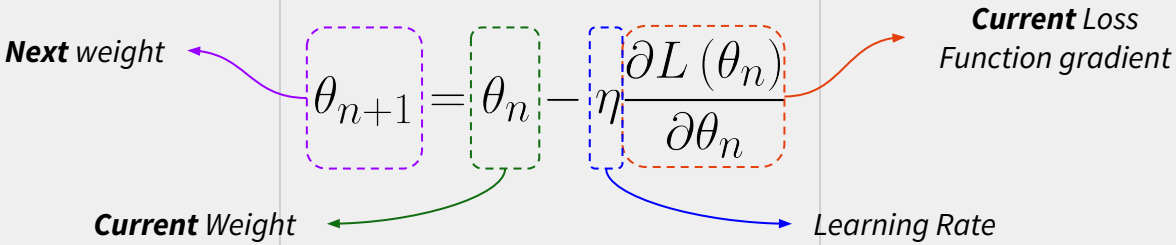
Model Optimization

Optimizers

Stochastic Gradient Descent

Optimizers are algorithms that are used to **minimize the loss function** of a neural network. There are various optimizers available, each with its own strengths and weaknesses.

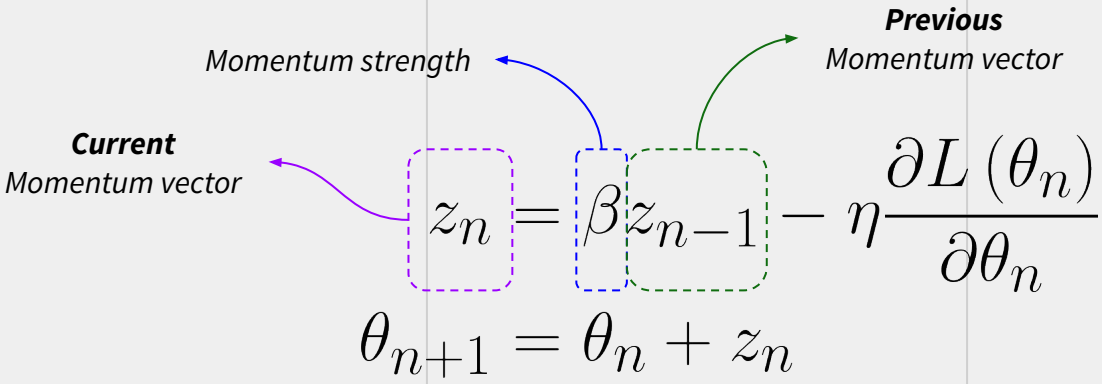
The most known optimizer is the **Stochastic Gradient Descent**. It is a **first-order optimization** algorithm that updates the weights of the neural network in the opposite direction of the gradient of the loss function for each training example (or batch of training examples).



Optimizers (2)

Momentum SGD

Momentum is a technique used in the Stochastic Gradient Descent algorithm to **accelerate the optimization process**, **improve the convergence rate** and lead to **better generalization** performance especially in situations where the loss function is noisy or has many local minima.



Optimizers (3)

Nesterov Accelerated Gradient

Nesterov Accelerated Gradient is a modification of the Stochastic Gradient Descent algorithm that aims to **improve the convergence rate** by taking into account the current momentum of the optimization process. It is particularly useful in situations where the loss function is highly non-linear or has many local minima.

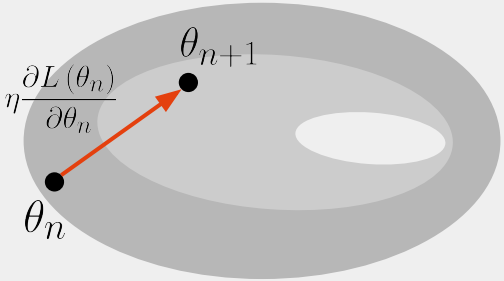
$$z_n = \beta z_{n-1} - \eta \frac{\partial L\left(\theta_n + \beta z_{n-1}\right)}{\partial \theta_n}$$

$$\theta_{n+1} = \theta_n + z_n$$

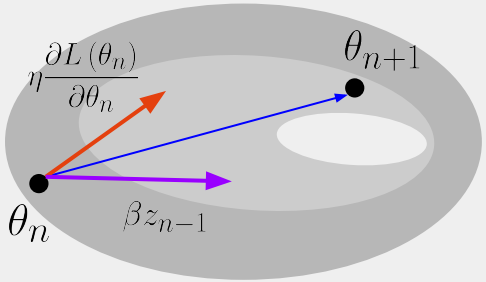
***Current** Loss Function gradient that takes the **current** momentum (i.e. the previous gradient) into account*

Optimizers (4)

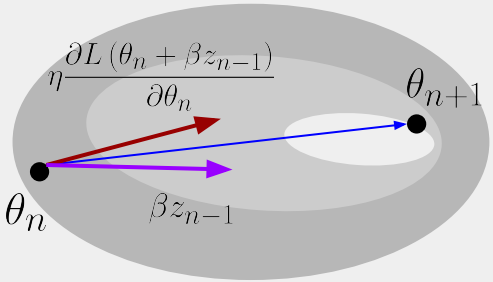
SGD vs Momentum vs NAG



Gradient Descent



Gradient Descent with ***momentum***



Gradient Descent with ***Nesterov*** and ***momentum***

Optimizers (5)

SGD disavantages

SGD, especially in combination with momentum and Nesterov, is a powerful optimization technique that has been used successfully in many deep learning applications.

However, it suffers from 4 main flaws:

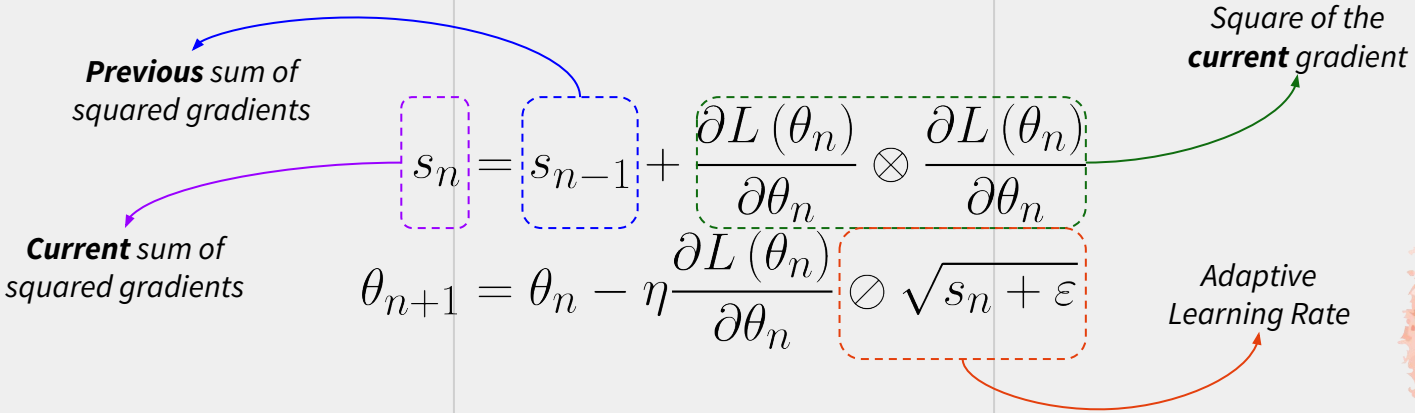
- It has a (very) **slow convergence**
- It is sensitive to the **choice of hyperparameters**
- It has a **fixed learning rate**
- It does not handle well **sparse data** and **noisy gradients**

Optimizers (6)

AdaGrad

Adagrad was introduced in 2011 by Duchi et al. as a way to **adaptively adjust the learning rate for each weight** in the neural network based on its historical gradients.

It is particularly **effective for sparse data**, and has been used successfully in many natural language processing applications.

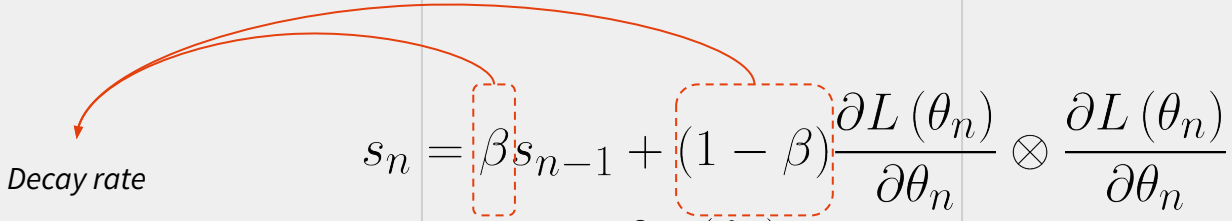


Optimizers (7)

RMSprop

RMSprop was introduced in 2012 by Geoff Hinton as a modification of Adagrad that addresses some of its limitations.

It uses a moving average of the squared gradients instead of the raw gradients, which makes it **more robust to noisy gradients** and can **help avoid the problem of the learning rate decaying too quickly**.



Optimizers (8)

Adam

Adam (Adaptive Moment Estimation) is an optimization algorithm used in deep learning that adapts the learning rate for each parameter in the neural network based on the historical gradients and weight updates. It was introduced in 2014 by Kingma and Ba as a **combination of RMSprop and Stochastic Gradient Descent with momentum**.

Current *momentum with a decaying rate beta*

$$z_n = \beta_z z_{n-1} - (1 - \beta_z)\frac{\partial L(\theta_n)}{\partial \theta_n}$$

Current *sum of squared gradients*

$$s_n = \beta_s s_{n-1} + (1 - \beta_s)\frac{\partial L(\theta_n)}{\partial \theta_n} \otimes \frac{\partial L(\theta_n)}{\partial \theta_n}$$

$$\theta_{n+1} = \theta_n - \eta \frac{z_n}{1 - \beta_z^t} \oslash \sqrt{\frac{s_n}{1 - \beta_s^t} + \varepsilon}$$

Iterations

Optimizers (9)

Nadam

Nadam (Nesterov-Adam) was introduced in 2016 by Dozat as a modification of Adam that incorporates the **Nesterov momentum** technique.

It uses Nesterov momentum to adjust the velocity of the optimization process based on the future position of the weight vector, which can further improve the convergence rate and performance of the algorithm.

Optimizers (10)

AdamW

AdamW is a variation of the Adam optimization algorithm that was introduced in 2019 by Loshchilov and Hutter.

The key modification in AdamW is the addition of a weight decay term to the loss function (*a regularized L2 norm*) during optimization, which helps **prevent overfitting** and **improve generalization** performance.

$$\theta_{n+1} = \theta_n - \eta \left(\frac{z_n}{1 - \beta_z^t} \oslash \sqrt{\frac{s_n}{1 - \beta_s^t} + \varepsilon} + \lambda_t \theta_n \right)$$

Weight decay term

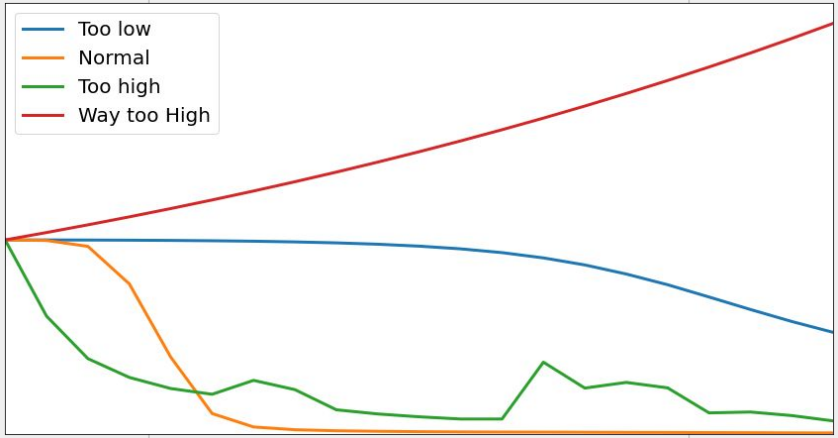
Optimizers (11)

Speed vs Quality

	Convergence Speed	Convergence Quality	Comments
SGD	Slow	Excellent	The learning rate needs to be carefully tuned.
SGD *with momentum*	Average	Excellent	Better than SGD.
SGD *with nesterov & momentum*	Average	Excellent	Better than SGD.
Adagrad	Fast	Poor	Do not use.
RMSprop	Fast	Good/Excellent	Very good optimizer.
Adam	Fast	Good/Excellent	Should be the default.
Nadam	Fast	Good/Excellent	More expensive than Adam.
AdamW	Fast	Good/Excellent	Effective against overfitting

Learning Rate

The learning rate is a **critical hyperparameter** that can greatly affect the performance of the neural network. If the learning rate is too high, the weights may oscillate and fail to converge, or may converge to a suboptimal solution. If the learning rate is too low, the optimization process may be slow and may get stuck in local minima.



Learning Rate (2)

There are several strategies for setting the learning rate in deep learning:

- **Fixed learning rate** - *The learning rate is fixed throughout training. This is a simple that requires careful tuning to find the optimal rate.*
- **Scheduled Learning rate** - *The learning rate is modified over time according to a predefined schedule. This can help the optimization process converge more effectively and avoid getting stuck in local minima.*
- **Adaptive learning rate** - *The learning rate is adjusted during training based on the historical gradients or weight updates. This can help the optimization process converge more effectively.*

Learning Rate (3)

Schedulers

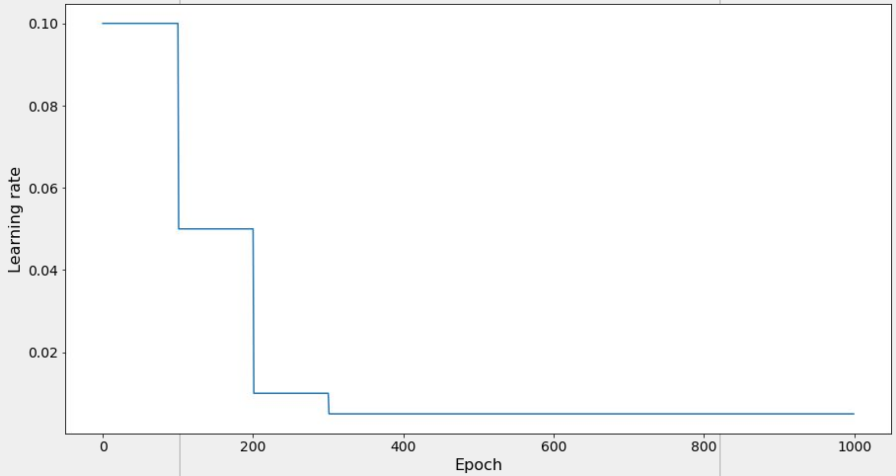
The goal of a learning rate scheduler is to improve the optimization process by **decreasing (or increasing) the learning rate over time**, which can help the model converge more effectively and avoid getting stuck in local minima.

There are several common types of learning rate schedulers including the Piecewise constant decay, the Exponential decay, the Polynomial decay, the Power decay, the Performance decay, the Cosine annealing and more complex schedulers such as One-Cycle.

Learning Rate (4)

Piecewise constant decay

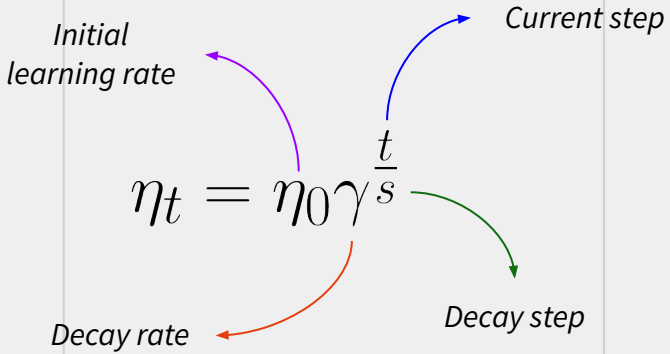
Piecewise constant decay decreases the learning rate at specific intervals during training. This schedule is implemented by defining a list of decay steps and a list of decay rates, and then decreasing the learning rate by the corresponding decay rate at each decay step.

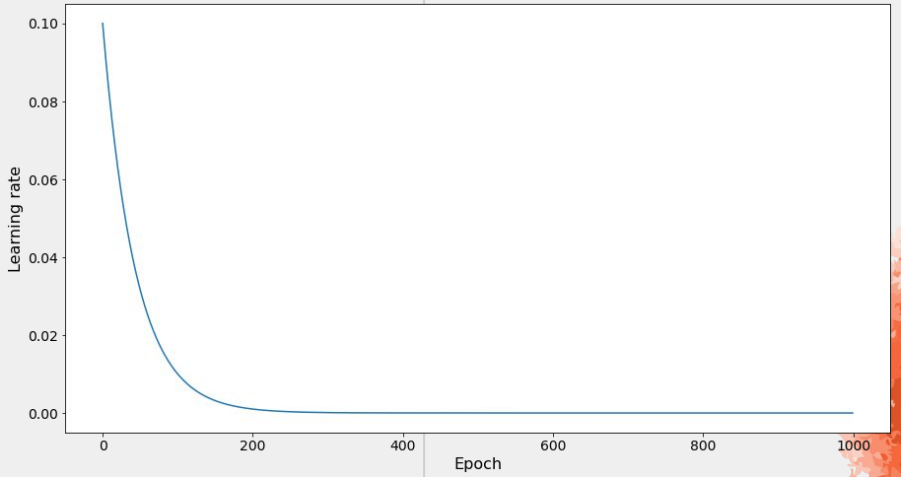


Learning Rate (5)

Exponential decay

Exponential decay decreases the learning rate over time during training at a exponential rate. This schedule is implemented by defining a decay rate and a decay step, and then decreasing the learning rate by a decay rate factor at each decay step.





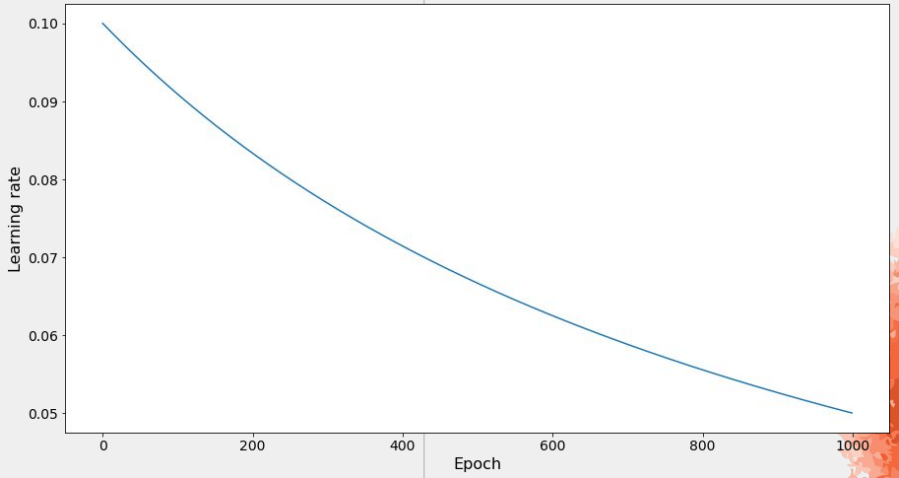
Learning Rate (6)

Power decay

Power decay gradually decreases the learning rate over time during training, according to an inverse time decay function. This schedule is implemented by defining the initial learning rate, the number of decay steps, a decay rate and a power for the inverse time decay function.

$$\eta_t = \frac{\eta_0}{\left(1 + \gamma \frac{t}{s}\right)^p}$$

Power



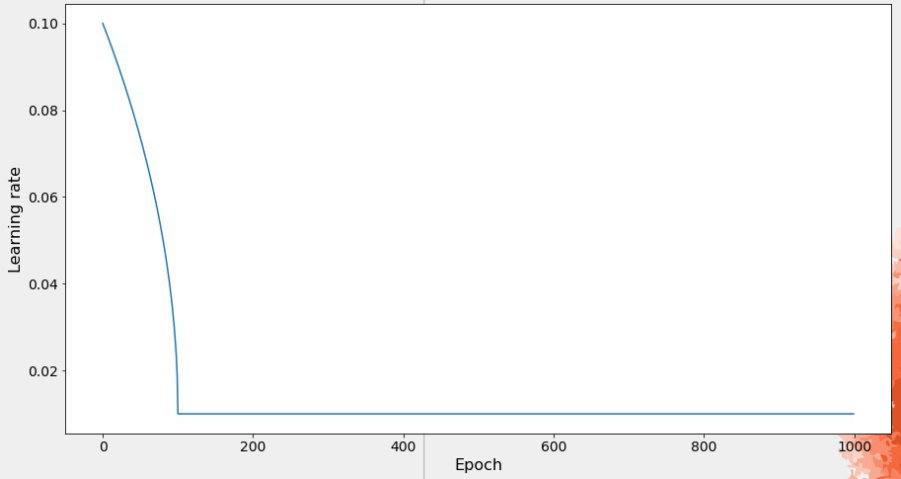
Learning Rate (7)

Polynomial decay

Polynomial decay gradually decreases the learning rate over time during training, according to a polynomial function. This schedule is implemented by defining the initial learning rate, the number of decay steps, the end learning rate, and a power factor for the polynomial function.

$$\eta_t = (\eta_0 - \eta_n)\left(1 - \frac{t}{s}\right)^p + \eta_n$$

End learning rate



Learning Rate (8)

Performance decay

Performance decay dynamically adjusts the learning rate based on the validation loss of the model during training. This schedule is implemented by monitoring the validation loss at regular intervals during training, and decreasing the learning rate if the validation loss has not improved by a specified factor for a specified number of epochs.

For example, consider a learning rate schedule that monitors the validation loss every 10 epochs, and decreases the learning rate by a factor of 0.1 if the validation loss has not improved by a factor of 0.01 for 3 consecutive epochs.

Learning Rate (9)

Cosine Annealing

Cosine annealing dynamically adjusts the learning rate based on a cosine function during training. The learning rate is maximum at the start of each cycle and decreases smoothly to the minimum learning rate at the end of the cycle. This process is then repeated as many times as needed.

Each cycle is characterized by a number of steps, an initial learning rate and an alpha parameter which is the minimum learning rate value as a fraction of initial learning rate.

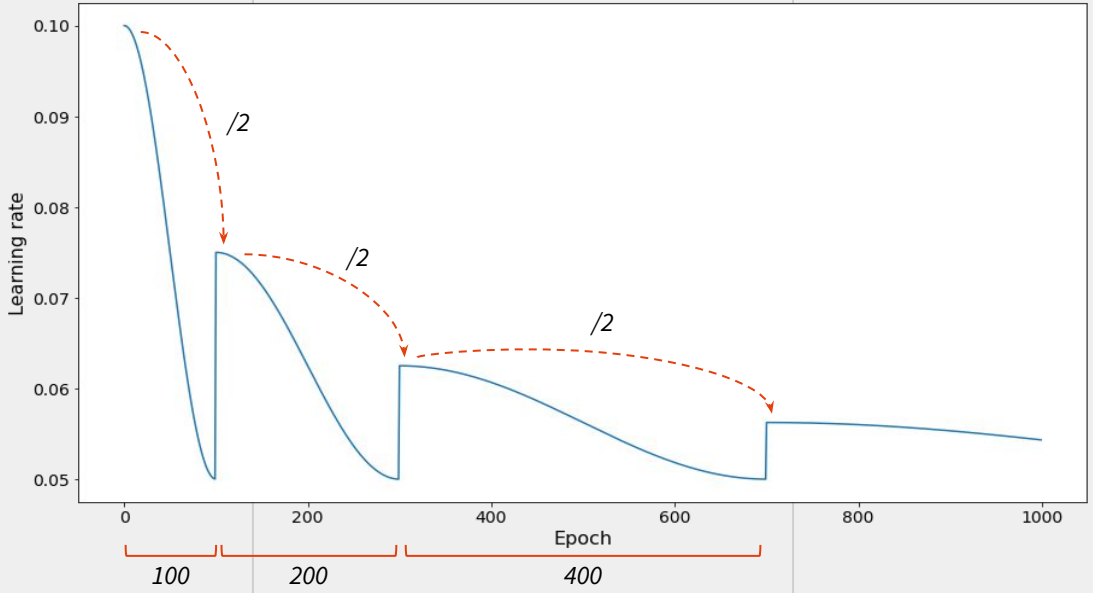
$$\eta_t = \eta_0 \frac{\left(1 + cos\left(\pi \frac{t}{s}\right)\right)}{2}(1 - \alpha) + \alpha$$

Minimum learning rate value as a fraction of initial learning rate

Learning Rate (10)

Cosine Annealing

In the following example, each subsequent cycle runs for two times more epochs with half the previous initial learning rate.



Learning Rate (11)

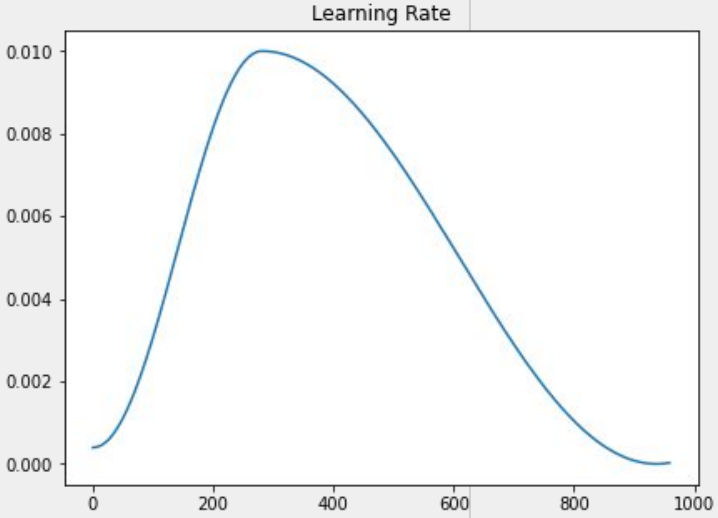
One-Cycle

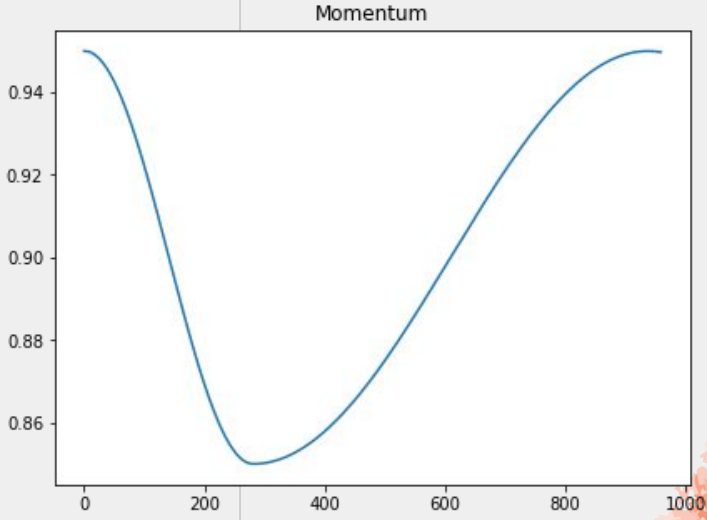
The **one-cycle scheduler** was introduced by Leslie Smith in 2018. The policy aims to **find the optimal learning rate and momentum** schedule for a given deep learning model in a single cycle of training. It consists of three phases:

- the **warm-up** - *the learning rate is gradually increased from a low initial value to a maximum value*
- the **annealing** - *the learning rate is gradually decreased from the maximum value to a low final value while the momentum is gradually increased from a low initial value to a maximum value*
- the **cool-down** - *the learning rate is further decreased to an even lower final value while the momentum is maintained at the maximum value.*

Learning Rate (12)

One-Cycle





Hyperparameters

The flexibility of neural networks is one of their main drawbacks. The number of layers, the number of neurons, the activation functions, the weight initialization, the optimizer, the loss function or even the regularization methods can all be tweaked.

	Defaults
Optimizer	Adam
Loss	MSE or CrossEntropy
Hidden layer activation functions	ELU
Hidden layer weight initialization	He Normal
Learning Rate Scheduler	Exponential decay, Performance decay or One-Cycle
Regularization	Early Stopping

Hyperparameters (2)

Number of hidden layers

For simple problems, a single hidden layer with a sufficient number of neurons can often achieve good performance. As the complexity of the problem increases, adding more hidden layers can allow the network to learn more complex and abstract features, leading to better performance.

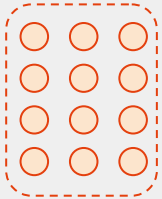
However, adding too many layers can lead to overfitting, where the model becomes too complex and memorizes the training data without generalizing well to new data.

A common approach is to **start with a single hidden layer** and **gradually increase the number of layers until the performance of the model no longer improves**.

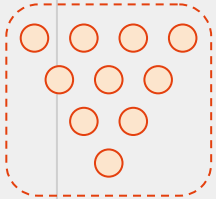
Hyperparameters (3)

Number of neurons

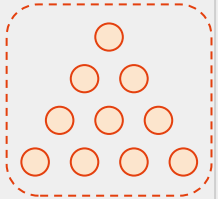
There is no one-size-fits-all to the number of neurons for the hidden layers of a deep neural network. A good default is to start with the **same low number of neurons for each hidden layer** and **gradually increase it until the performance of the model no longer improves**.



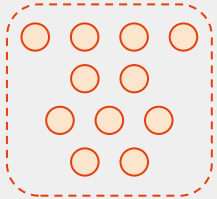
It simplifies the architecture of the network and is easier to train.



It reduces the risk of overfitting but increases the risk of underfitting.



It reduces the risk of underfitting but increases the risk of overfitting.



It can explore a larger space of possible architectures.

Hyperparameters (4)

Batch Size

The **batch size** is a hyperparameter that determines **the number of samples that are processed in each forward and backward pass** of a deep neural network (DNN). The choice of batch size can have a significant effect on the performance of the model:

- **Speed** - *Larger batch sizes can result in faster training times*
- **Generalization** - *Smaller batch sizes can lead to better generalization*
- **Stability** - *Smaller batch sizes can lead to training unstability*
- **Convergence** - *Larger batch sizes can lead to faster convergence, but smaller batch sizes can lead to higher performances*
- **Local minima** - *Larger batch sizes can help the model escape from local minima*

Hyperparameters (5)

Algorithms

Many Machine Learning hyperparameter search algorithms are also available for neural networks including:

- **Grid** & **random** searches
- **Bayesian optimization** such as Tree-structured Parzen Estimator or the Gaussian Process
- **Evolutionary algorithms** such as Genetic Algorithm, Particle Swarm Optimization or Population Based Training
- **Adaptive resource allocation algorithm** optimization such as Hyperband or Asynchronous Successive Halving Algorithm
- **Reinforcement** optimization