Dive into Deep Learning: A Comprehensive Introduction

From AI Fundamentals to Cutting-Edge Deep Learning Techniques

Alexandre Vérine,
Research Fellow, ENS-PSL
Université PSL

Executive Master IASD
Université Paris-Dauphine, PSL

January 20, 2025

AI 101: From Fundamentals to Deep Learning

Part I

AI 101: From Fundamentals to Deep Learning

Introduction to Artificial Intelligence

Deep Learning in the AI family

In general, among all the class of AI algorithms, we make the difference between 3 sub-categories :

- Artificial Intelligence : human designed program and...
- Machine Learning : human designed features with learned mapping such as Support Vector Machine, Kernels methods, Logistic Regression and ...
- Deep Learning: Learned features with learned mapping such as Multilayer Perceptron, Convolutional Networks, ...

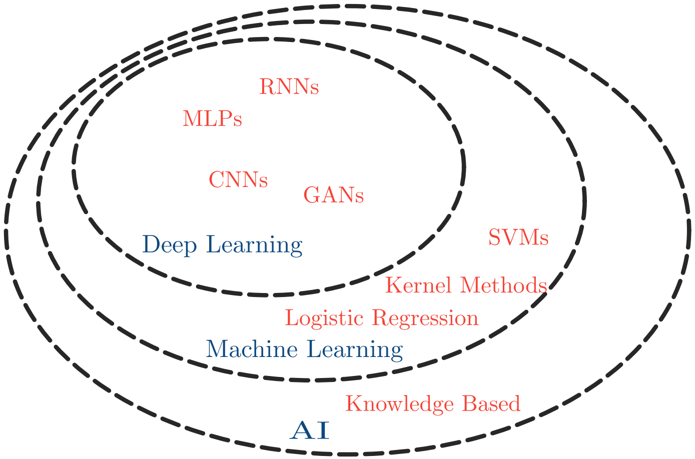


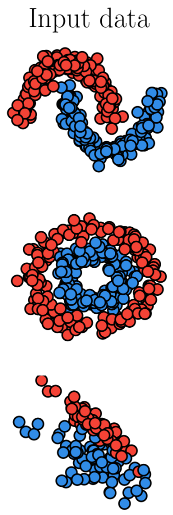
Figure. Subsets of Artificial Intelligence

INTRODUCTION TO ARTIFICIAL INTELLIGENCE
DEEP LEARNING IN THE AI FAMILY

In the field of Artificial Intelligence, the fundamental objective is to find a function f that can perform a desired task. This function can either be set by a human or can be learned through training.

For example, in the context of a binary classification task, the goal is to determine $f(x)$ such that $f(x) = 0$ when the label of x is 0 and $f(x) = 1$ when its label is 1. The choice of AI model impacts the expressivity of the function f.

For example, a logistic regression model uses a linear function to make decisions, where $f(x) = \text{sgn}(Ax + b)$. The expressivity of the model can be increased by using more complex functions, such as polynomials or radial basis functions.



Introduction to Artificial Intelligence

Classification task

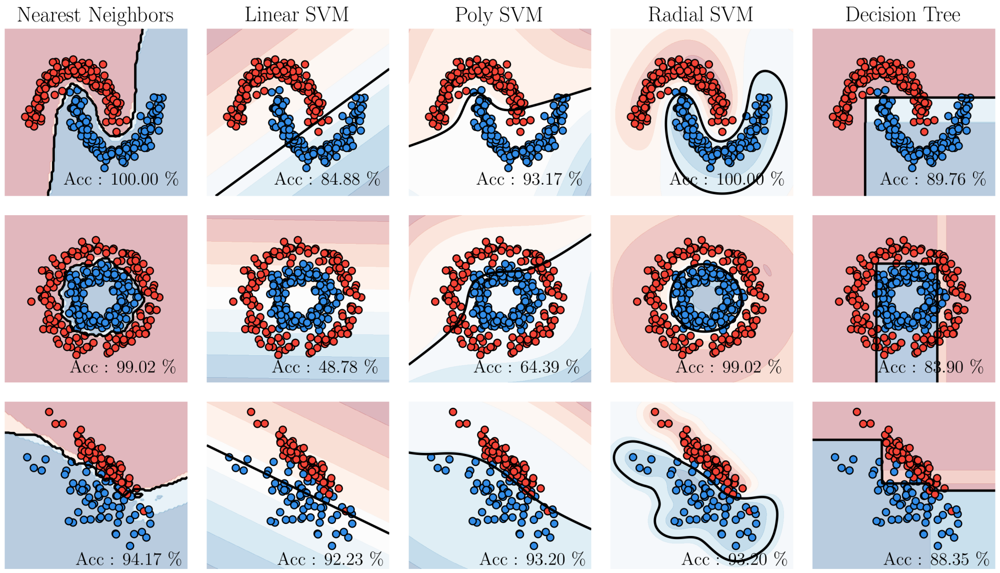


Figure. 2D classification for different AI models.

Introduction to Artificial Intelligence

The Universal Approximation Theorem

The Universal Approximation Theorem is a fundamental result in the field of artificial neural networks. It states that a deep learning model can approximate any function.

Theorem 1 (Universal Approximation Theorem)

Let $\mathcal{X} \subset \mathbb{R}^d$ be compact, $\mathcal{Y} \subset \mathbb{R}^m$, $f : \mathcal{X} \to \mathcal{Y}$ be a continuous function and $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous real function.
Then σ is not polynomial if and only if for every $\epsilon > 0$, there exist $k \in \mathbb{N}$, $A \in \mathbb{R}^{k \times d}$, $b \in \mathbb{R}^k$ and $C \in \mathbb{R}^{m \times k}$ such that

$$\sup_{x \in \mathcal{X}} \|f(x) - g(x)\| \leq \epsilon$$

where $g(x) = C \times \sigma(Ax + b)$.

Introduction to Artificial Intelligence

Classification task

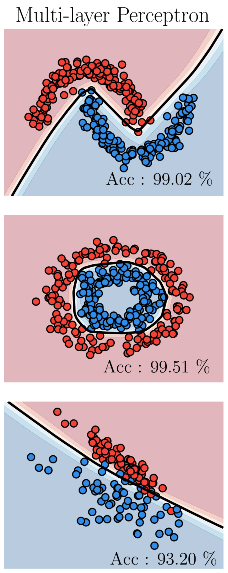


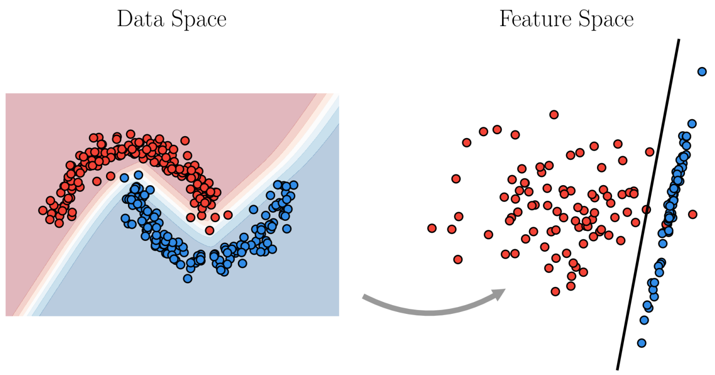
Figure. 2D classification for small Neural Network.

Introduction to Artificial Intelligence

Representation Learning

How does deep learning work in practice ?

Deep learning is a subset of representation learning that uses deep neural networks to learn meaningful representations of data. In deep learning, representations are learned through a hierarchy of nonlinear transformations, where each layer of the network builds upon the previous one to extract increasingly abstract and higher-level features from the input data.



Introduction to Artificial Intelligence
Example of Representation Learning

Consider the task of recognizing objects in images. A traditional approach would be to hand-engineer features such as edge detectors and color histograms that can be fed into a classifier.
However, with deep learning representation learning, the model learns to automatically discover these features from the data. The network might start by learning simple features such as edges and color blobs in the first layer, then build upon these to learn more complex features such as parts of objects in subsequent layers, until finally, the final layer outputs a probability distribution over classes of objects.
In this way, deep learning of representation enables the model to automatically learn a rich and meaningful representation of the data, without the need for manual feature engineering.

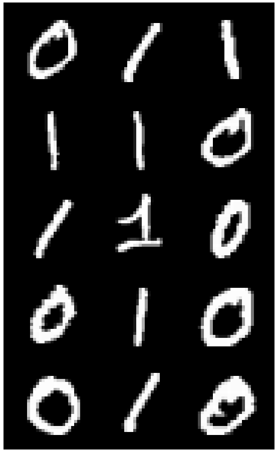


Figure. MNIST

Introduction to Artificial Intelligence

Example of Representation Learning

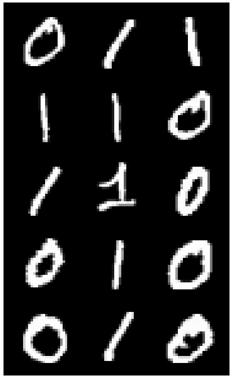


Figure. MNIST : Layer 0

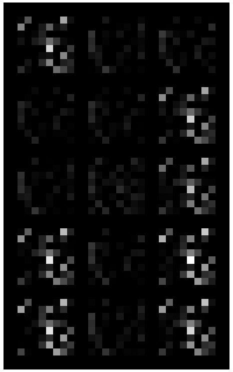


Figure. MNIST : Layer 1

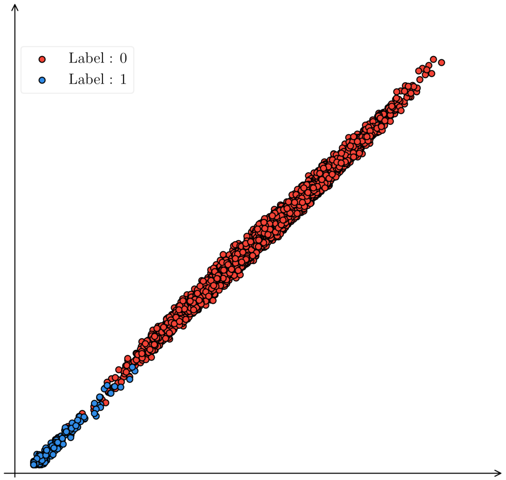


Figure. MNIST : Layer 2

Introduction to Artificial Intelligence
Example of Representation Learning

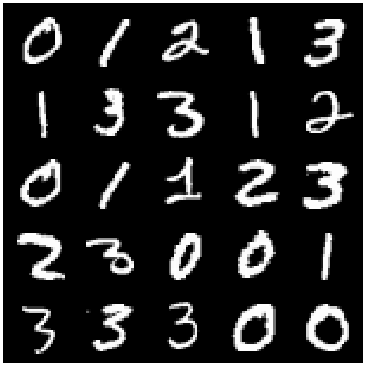


Figure. MNIST : Layer 0

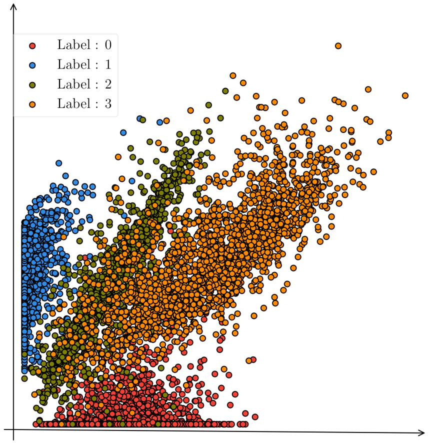


Figure. MNIST : Layer 2

Introduction to Artificial Intelligence

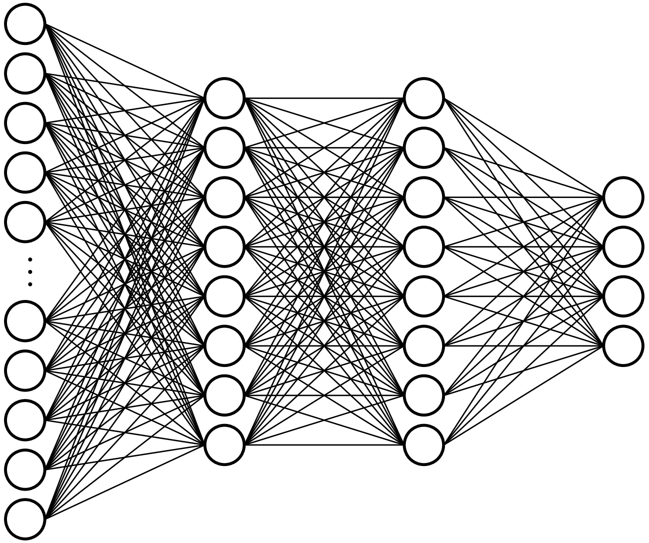
Deep Learning and Neural Networks

Ok, Deep Learning is a model that learns a good representation of the feature. But how?

- How does it work ?
- How can we build a model ?
- How does it learn ?

Neural Networks Fundamentals

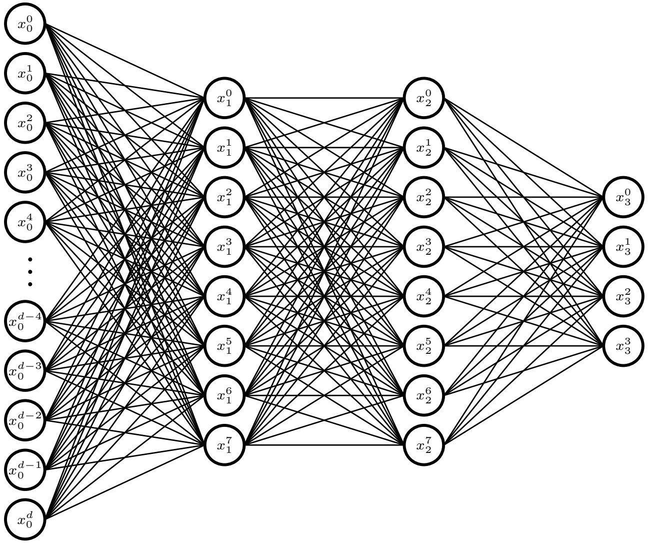
Typically, a neural network is defined as a computational model composed of interconnected nodes, organised into layers, that perform transformations on input data.



Let's see what the interconnected nodes, the layers and the transformations are.

Neural Networks Fundamentals
Neurons

If we consider that the Neural Network is a function $f : \mathbb{R}^d \to \mathbb{R}^m$:

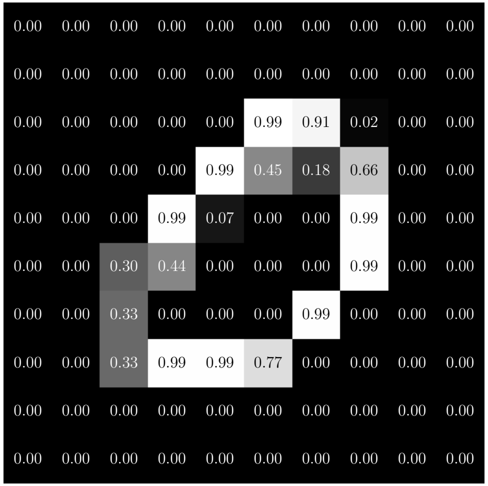


A Neuron is a processing unit that receives input, performs a computation, and produces an output. Here, the inputs are x_{i-1} and the output is x_i^k.

Neural Networks Fundamentals

Neurons

For example, with an image dataset, the image can be flattened:



0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.99	0.91	0.02	0.00	0.00
0.00	0.00	0.00	0.00	0.99	0.45	0.18	0.66	0.00	0.00
0.00	0.00	0.00	0.99	0.07	0.00	0.00	0.99	0.00	0.00
0.00	0.00	0.30	0.44	0.00	0.00	0.00	0.99	0.00	0.00
0.00	0.00	0.33	0.00	0.00	0.00	0.99	0.00	0.00	0.00
0.00	0.00	0.33	0.99	0.99	0.77	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

$\in [0,1]^{d/2 \times d/2}$

$$x_0 = [0.00, 0.00, \dots, 0.00, 0.99, 0.07 \dots, 0.00, 0.00] \in [0,1]^d$$

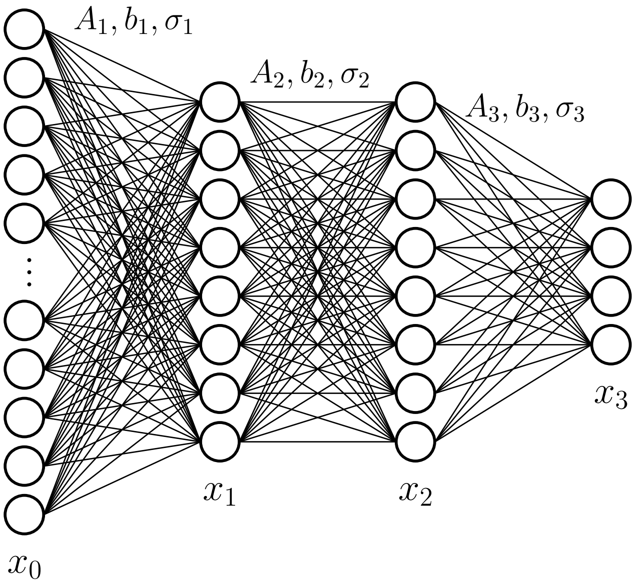
Neural Networks Fundamentals

Layers

A layer i is defined by a matrix $A_i \in \mathbb{R}^{k_{i-1} \times k_i}$, a vector $b_i \in \mathbb{R}^{k_i}$ and a nonlinear function $\sigma_i : \mathbb{R} \mapsto \mathbb{R}$. The transformation made by a layer is:

$$x_i = \sigma_i \left(A_i x_{i-1} + b_i \right) .$$

The non-linear function σ_i the activation function.



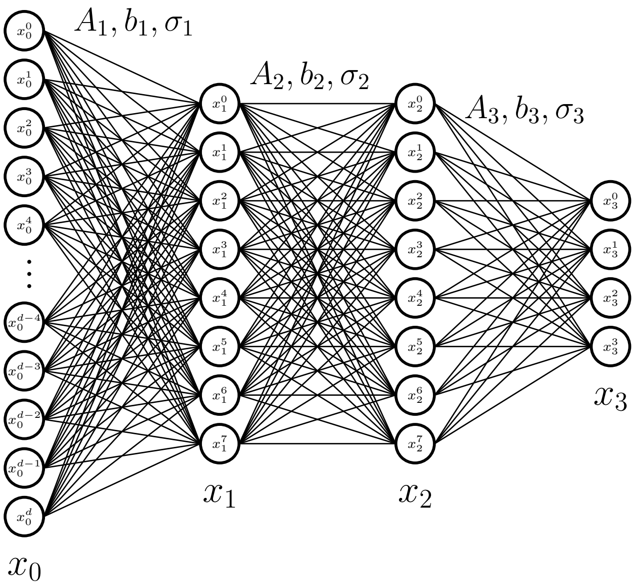
Neural Networks Fundamentals

Layers

A layer i is defined as a matrix $A_i \in \mathbb{R}^{k_{i-1} \times k_i}$, a vector $b_i \in \mathbb{R}^{k_i}$ and a nonlinear function $\sigma_i : \mathbb{R} \mapsto \mathbb{R}$. The transformation made by a layer is:

$$x_i^k = \sigma_i \left(\sum_{l=1}^{k_i} [A_i]_{l,k} \, x_{i-1} + [b_i]_k \right) .$$

The non-linear function σ_i the activation function.



Neural Networks Fundamentals

Activation Functions

The activation functions play a crucial role in the implementation of deep neural networks, as they allow them to approximate any continuous function, as stated by the Universal Approximation Theorem. We can list some activation function that are commonly used :

- Linear
- Sigmoid
- Hyperbolic Tangent
- Rectified Linear Unit (ReLU)
- Leaky Rectified Linear Unit (Leaky ReLU)
- Exponential Linear Unit (ELU)
- Sigmoid-Weighted Linear Unit (Swish)
- Softmax

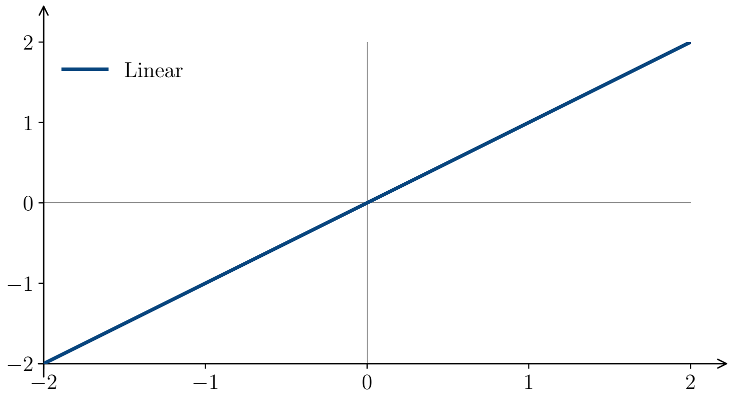
Neural Networks Fundamentals

Linear

- Linear activation Function:

$$\sigma(x) = x$$

- Final activation
- Use case : Regression



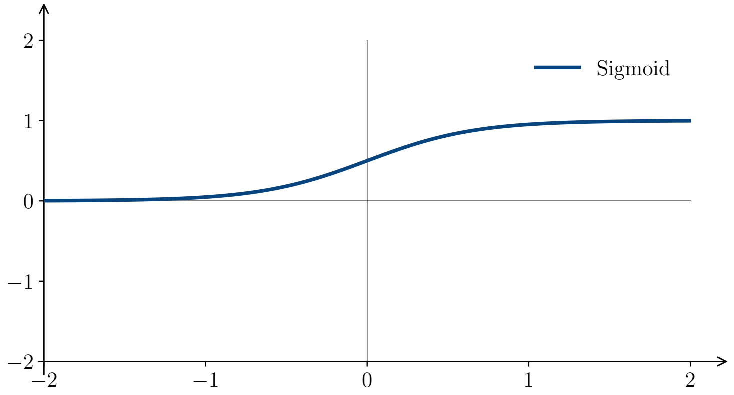
Neural Networks Fundamentals

Sigmoid

- Sigmoid Function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Final activation
- Use case : Classification



Neural Networks Fundamentals
Softmax

- Softmax Function:

$$\sigma(x_k) = \frac{e^{x_k}}{\sum_{i=1}^{k_i} e^{x_i}}$$

- Final activation
- Use case : Multi-class Classification

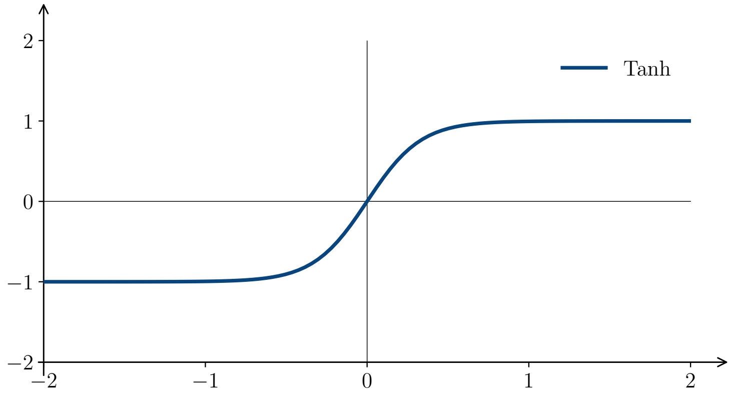
$$\begin{bmatrix} 0.7 \\ -2.1 \\ 0.0 \\ 2.0 \\ -25.3 \end{bmatrix} \longrightarrow \boxed{\frac{e^{x_k}}{\sum_i e^{x_i}}} \longrightarrow \begin{bmatrix} 0.19 \\ 0.01 \\ 0.09 \\ 0.70 \\ 0.000 \end{bmatrix}$$

Neural Networks Fundamentals
Hyperbolic Tangent

- Hyperbolic Tangent

$$\sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Final activation
- Use case : Generative task

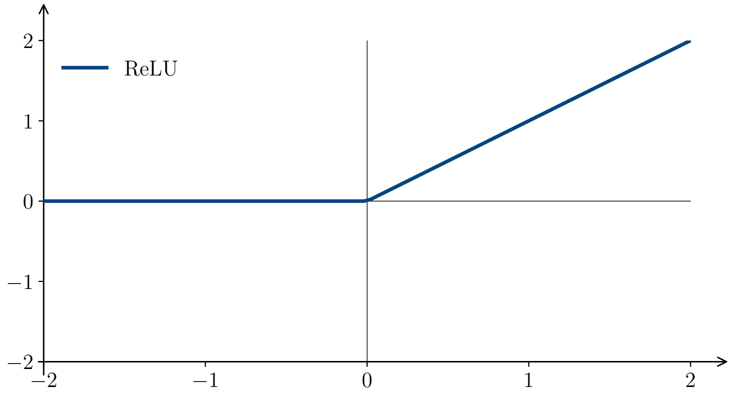


Neural Networks Fundamentals
ReLU

- Rectified Linear Unit (ReLU):

$$\sigma(x) = \max\{0, x\}$$

- Intermediate activation



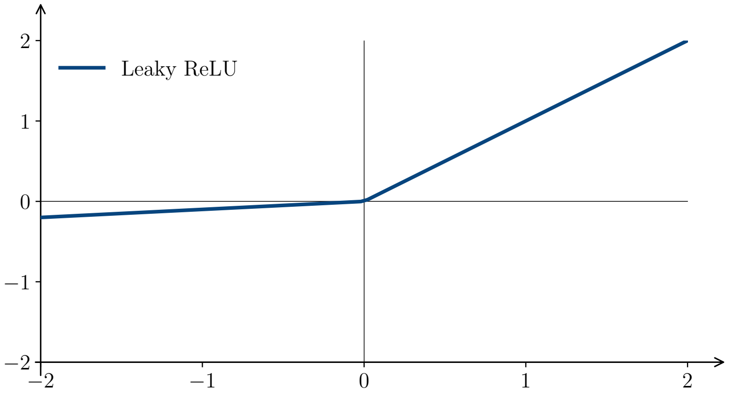
Neural Networks Fundamentals

Leaky ReLU

- Leaky Rectified Linear Unit (Leaky ReLU):

$$\sigma(x) = \max\{\alpha x, x\}$$

- Intermediate activation



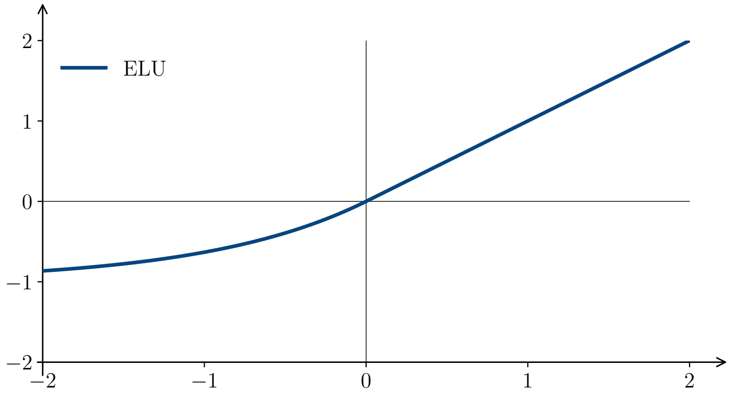
Neural Networks Fundamentals

ELU

- Exponential Linear Unit (ELU):

$$\sigma(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x < 0, \\ x & \text{if } x \geq 0. \end{cases}$$

- Intermediate activation

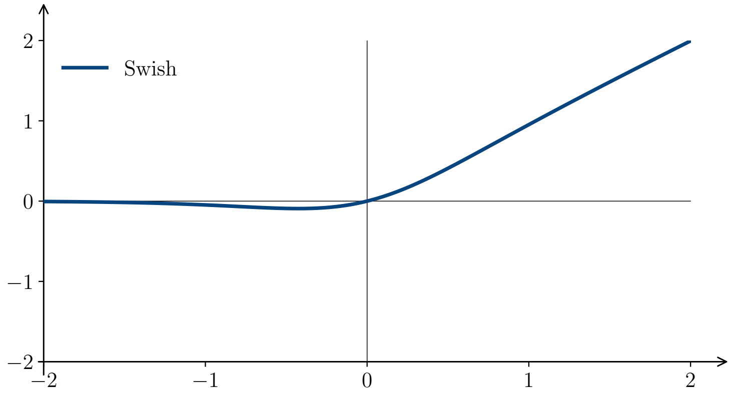


Neural Networks Fundamentals
Swish

- Sigmoid-Weighted Linear Unit (Swish):

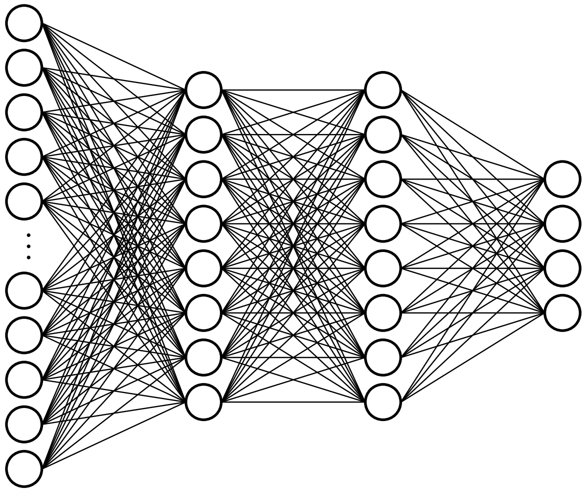
$$\sigma(x) = \frac{x}{1 + e^{-x}}$$

- Intermediate activation



The Multi-layer Perceptron (MLP)

Having discussed the structure of a neural network, we will proceed to examine the process of training a model for a specific task. As an illustration, we will consider the example of a Multilayer Perceptron.The two intermediate activation functions are ReLUs and the final activation is a softmax to perform multi-class classification on MNIST. We will consider only 4 classes.

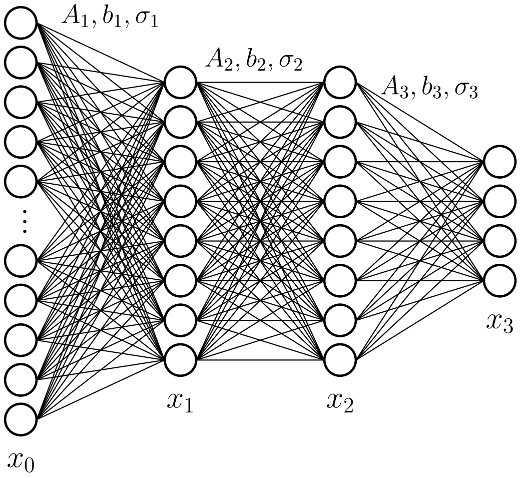


The Multi-layer Perceptron (MLP)

The first Deep Learning Model

To introduce the training process, we will consider a 3 layers MLP trained to minimise a loss $\mathcal{L}$ over a given a dataset $\mathcal{D}$. The model f_θ is parameterised by a vector $\theta = \{A_1, A_2, A_3, b_1, b_2, b_3\}$:

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta, \mathcal{D})$$



The Multi-layer Perceptron (MLP)
Stochastic Gradient Descent

Stochastic gradient descent (SGD) is widely used in deep learning instead of traditional gradient descent due to its efficiency and faster convergence rate. SGD updates the model parameters after computing the gradient of the loss function with respect to each parameter using only a single randomly selected sample. This leads to a faster convergence rate and improved optimization compared to traditional gradient descent, which uses the entire training dataset to compute the gradient at each iteration.

$$\theta^* = \arg\min_\theta \mathcal{L}(\theta, \mathcal{D}) = \arg\min_\theta \mathbb{E}_{x\sim\mathcal{D}}\left[l(x, f_\theta(x))\right]$$

The Multi-layer Perceptron (MLP)

Stochastic Gradient Descent

Theoretically the algorithm is the following:

```
Require: Given a loss function $l$, a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$ and a learning rate $\lambda$
1: Initialize parameters $\theta$
2: while $\theta$ has not converged do
3:     for $i = 1$ to $N$ do
4:         Randomly select $x_i$ from the dataset
5:         Compute gradient of the loss with respect to $\theta$: $\nabla_\theta l(x_i, f_\theta(x_i))$
6:         Update parameters $\theta = \theta - \lambda \nabla_\theta l(x_i, f(x_i))$
7:     end for
8: end while
9: return $\theta$
```

The Multi-layer Perceptron (MLP)

SGD in mini-batch

In practice the algorithm is modified to use mini-batches of data instead of single samples. This is done to improve the stability of the optimization process and reduce the variance of the gradient estimates. The algorithm is as follows:

```
Require: Given a loss function $l$, a dataset $\mathcal{D} = \{x_1, x_2, \dots, x_N\}$, a learning rate $\lambda$ and a batch size $b$
 1: Initialize parameters $\theta$
 2: Initialize the number of batches $B = \lfloor \frac{N}{b} \rfloor$
 3: while $\theta$ has not converged do
 4:     for $i = 1$ to $B$ do
 5:         Randomly select a mini-batch of $b$ samples from the dataset
 6:         Compute gradient of the loss with respect to $\theta$: $\frac{1}{B} \sum_{i=1}^{B} \nabla_\theta l(x_i, f_\theta(x_i))$
 7:         Update parameters $\theta = \theta - \lambda \frac{1}{B} \sum_{i=1}^{B} \nabla_\theta l(x_i, f(x_i))$
 8:     end for
 9: end while
10: return $\theta$
```

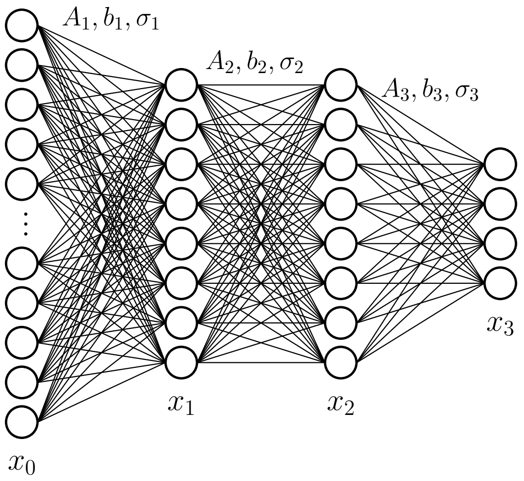
The Multi-layer Perceptron (MLP)

Back-propagation

At every step t of the gradient descent, setting a learning rate λ, the parameter θ is updated as:

$$\theta_{t+1} = \theta_t - \lambda \nabla_\theta l(f(x_i), y_i)$$

But $\theta = \{A_1, A_2, A_3, b_1, b_2, b_3\}$ and the gradient is computed with respect to each parameter.



The Multi-layer Perceptron (MLP)

Back-propagation

First we will consider a single data point x, the loss will depend on the output only: $l(f(x))$.

f is a layered composed function. Let us focus on the last layer:

$$f(x) = x_3 = \sigma_3(A_3 x_2 + b_3)$$

Therefore:

$$l(f(x)) = l\left(\sigma_3\left(A_3 x_2 + b_3\right)\right)$$

To minimise the loss, we have to act on A_3, b_3 and x_2.

The Multi-layer Perceptron (MLP)

Back-propagation

Let us look at the gradients with respect to A_3:

$$\frac{\partial l}{\partial A_3} = \frac{\partial l}{\partial x_3}\frac{\partial x_3}{\partial A_3} = l'(x_3)\frac{\partial \sigma_3\left(A_3x_2+b_3\right)}{\partial A_3} = l'(x_3)\sigma_3'\left(A_3x_2+b_3\right)\frac{\partial\left[A_3x_2+b_3\right]}{\partial A_3}$$
$$= \underbrace{l'(x_3)}_{\in\mathbb{R}}\ \underbrace{\sigma_3'\left(A_3x_2+b_3\right)}_{\in\mathbb{R}^{k_i\times 1}}\ \underbrace{x_2^T}_{\in\mathbb{R}^{1\times k_{i-1}}}$$

and therefore:

$$A_3 \leftarrow A_3 - \lambda l'(x_3)\sigma_3'\left(A_3x_2+b_3\right)x_2^T.$$

We need to keep in memory the latent values of x, i.e. x_2.

The Multi-layer Perceptron (MLP)

Back-propagation

Let us look at the gradients with respect to A_2:

$$\begin{aligned}\frac{\partial l}{\partial A_2} &= \frac{\partial l}{\partial x_2}\frac{\partial x_2}{\partial A_2} \\ &= \frac{\partial l}{\partial x_2}\frac{\partial \sigma_2\left(A_2 x_1 + b_2\right)}{\partial A_2} \\ &= \frac{\partial l}{\partial x_2}\sigma_2'\left(A_2 x_1 + b_2\right)\frac{\partial\left[A_2 x_1 + b_2\right]}{\partial A_2} \\ &= \frac{\partial l}{\partial x_2}\,\sigma_2'\left(A_2 x_1 + b_2\right) x_1^T\end{aligned}$$

which depends on $\frac{\partial l}{\partial x_2}$, we need to compute it.

The Multi-layer Perceptron (MLP)

Back-propagation

We have to compute the gradient with respect to x_2:

$$\begin{aligned}\frac{\partial l}{\partial x_2} &= \frac{\partial l}{\partial x_3}\frac{\partial x_3}{\partial x_2} = l'(x_3)\frac{\partial \sigma_3\left(A_3 x_2 + b_3\right)}{\partial x_2} = l'(x_3)\frac{\partial\left[A_3 x_2 + b_3\right]}{\partial x_2}\sigma_3'\left(A_3 x_2 + b_3\right)\\ &= l'(x_3)\, A_3^T \sigma_3'\left(A_3 x_2 + b_3\right)\end{aligned}$$

Therefore:

$$A_2 \leftarrow A_2 - \lambda\left[l'(x_3) A_3^T \sigma_3'\left(A_3 x_2 + b_3\right) \times \sigma_2'\left(A_2 x_1 + b_2\right) x_1^T\right]$$

The update of A_2 depends on $l'(x_3)$,

Back-propagation

We have to compute the gradient with respect to A_1:

$$\begin{aligned}\frac{\partial l}{\partial A_1} &= \frac{\partial l}{\partial x_1}\frac{\partial x_1}{\partial A_1}\\ &= \frac{\partial l}{\partial x_1}\frac{\partial \sigma_1\left(A_1 x_0 + b_1\right)}{\partial A_1}\\ &= \frac{\partial l}{\partial x_1}\sigma_1'\left(A_1 x_0 + b_0\right) x_0^T,\end{aligned}$$

which depends on $\frac{\partial l}{\partial x_1}$, we need to compute it.

Back-Propagation

Let us compute the gradient with respect to x_1:

$$\begin{aligned}\frac{\partial l}{\partial x_1} &= \frac{\partial l}{\partial x_2}\frac{\partial x_2}{\partial x_1} = \frac{\partial l}{\partial x_2}\frac{\partial \sigma_2\left(A_2x_1+b_2\right)}{\partial x_1} = \frac{\partial l}{\partial x_2}\frac{\partial\left[A_2x_1+b_2\right]}{\partial x_1}\sigma_2'\left(A_2x_1+b_2\right)\\ &= \frac{\partial l}{\partial x_2}A_2^T\sigma_2'\left(A_2x_1+b_2\right)\end{aligned}$$

Therefore:

$$A_1 \leftarrow A_1 - \lambda\left[l'(x_3)\,A_3^T\sigma_3'\left(A_3x_2+b_3\right)A_2^T\sigma_2'\left(A_2x_1+b_2\right)\times\sigma_1'\left(A_1x_0+b_1\right)x_0^T\right]$$

Back-Propagation

In other words, the update on the weights is:

$$
\begin{aligned}
A_3 &\leftarrow A_3 - \lambda l'(x_3)\sigma_3'\left(A_3x_2 + b_3\right)x_2^T \\
A_2 &\leftarrow A_2 - \lambda\left[l'(x_3)A_3^T\sigma_3'\left(A_3x_2 + b_3\right) \times \sigma_2'\left(A_2x_1 + b_2\right)x_1^T\right] \\
A_1 &\leftarrow A_1 - \lambda\left[l'(x_3)\,A_3^T\sigma_3'\left(A_3x_2 + b_3\right)A_2^T\sigma_2'\left(A_2x_1 + b_2\right) \times \sigma_1'\left(A_1x_0 + b_1\right)x_0^T\right]
\end{aligned}
$$

The Multi-layer Perceptron (MLP)

Back-propagation

If we look at the update of the different biases, we can easily compute the different gradient and see the updates. First, let us compute the gradient with respect to b_3:

$$
\begin{aligned}
\frac{\partial l}{\partial b_3} &= \frac{\partial l}{\partial x_3}\frac{\partial x_3}{\partial b_3} \\
&= l'(x_3)\frac{\partial \sigma_3\left(A_3x_2 + b_3\right)}{\partial b_3} \\
&= l'(x_3)\sigma_3'\left(A_3x_2 + b_3\right)\frac{\partial\left[A_3x_2 + b_3\right]}{\partial b_3} \\
&= \underbrace{l'(x_3)}_{\in\mathbb{R}}\ \underbrace{\sigma_3'\left(A_3x_2 + b_3\right)}_{\in\mathbb{R}^{k_i\times 1}}
\end{aligned}
$$

And thus :

$$b_3 \leftarrow b_3 - \lambda l'(x_3)\sigma'\left(A_3x_2 + b_3\right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Let's move on the second layer:

$$
\begin{aligned}
\frac{\partial l}{\partial b_2} &= \frac{\partial l}{\partial x_2}\frac{\partial x_2}{\partial b_2} \\
&= \frac{\partial l}{\partial x_2}\frac{\partial \sigma_2\left(A_2 x_1 + b_2\right)}{\partial b_2} \\
&= \frac{\partial l}{\partial x_2}\sigma_2'\left(A_2 x_1 + b_2\right)
\end{aligned}
$$

And thus :

$$
b_2 \leftarrow b_2 - \lambda \frac{\partial l}{\partial x_2}\sigma'\left(A_2 x_1 + b_2\right)
$$

We need to back-propagate the term $\frac{\partial l}{\partial x_2}$ computed for the first layer.

The Multi-layer Perceptron (MLP)

Back-propagation

For the first layer:

$$\begin{aligned}\frac{\partial l}{\partial b_1} &= \frac{\partial l}{\partial x_1}\frac{\partial x_1}{\partial b_1}\\ &= \frac{\partial l}{\partial x_1}\frac{\partial \sigma_1\left(A_1 x_0 + b_1\right)}{\partial b_1}\\ &= \frac{\partial l}{\partial x_1}\sigma_1'\left(A_1 x_0 + b_0\right)\end{aligned}$$

And thus :

$$b_1 \leftarrow b_1 - \lambda\frac{\partial l}{\partial x_1}\sigma'\left(A_1 x_0 + b_1\right)$$

We need to back-propagate the term $\frac{\partial l}{\partial x_1}$ computed for the second layer which has been computed with $\frac{\partial l}{\partial x_2}$ back-propagated from the first layer.

The Multi-layer Perceptron (MLP)

Back-Propagation

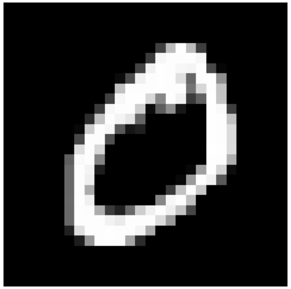
To update the weights, we need to compute the gradient of the loss with respect to the output of the network, and then **back-propagate** the gradient of the loss with respect to each activation, the $\frac{\partial l}{\partial x_i}$, through the network to compute the gradients with respect to the weights and biases of each layer.

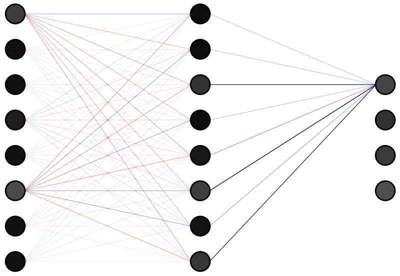
The Multi-layer Perceptron (MLP)

Last Layer

We can plot the current state of the network for a given input.

The red lines show positive values for A_i, the blue lines represent negative values for A_i. The level of transparency is proportional to the previous neurons.



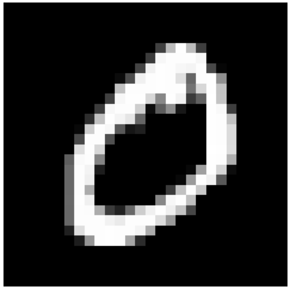


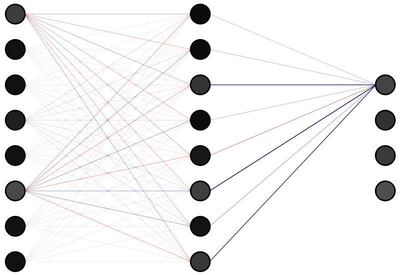
$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



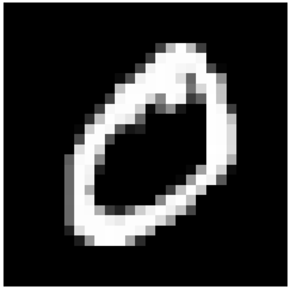


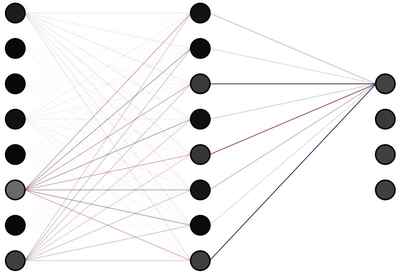
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



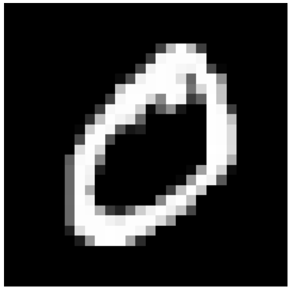


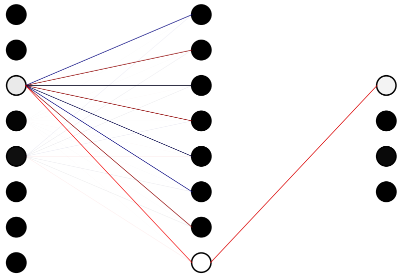
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



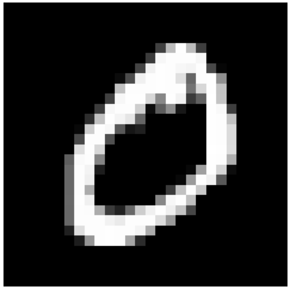


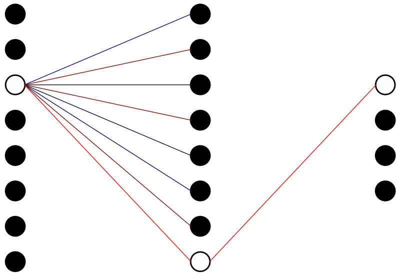
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.





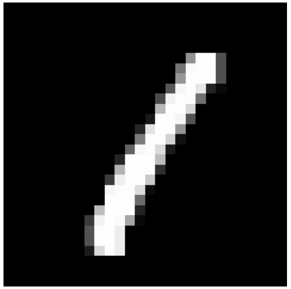
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

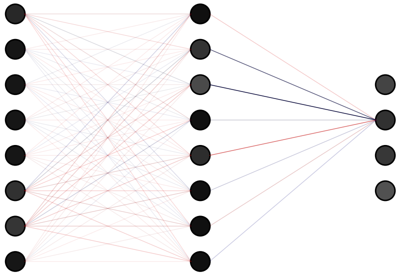
The Multi-layer Perceptron (MLP)

Last Layer

We can plot the current state of the network for a given input.

Red lines show positive values of A_i, Blue lines represent negative values of A_i. The level of transparency is proportional to the previous neurons.



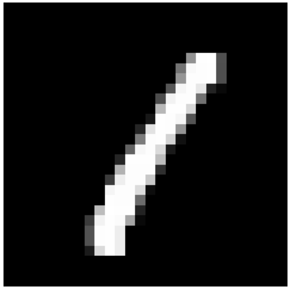


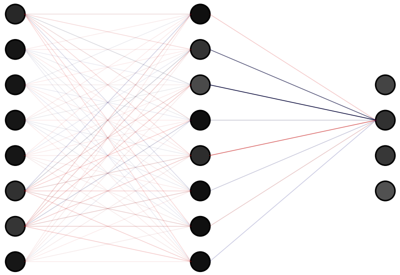
$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



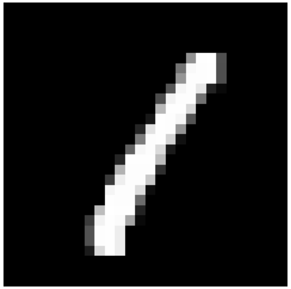


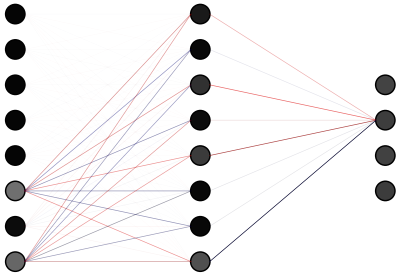
$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



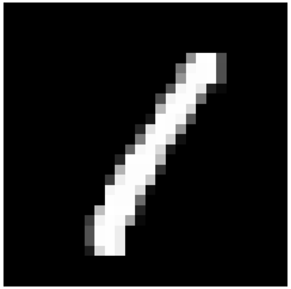


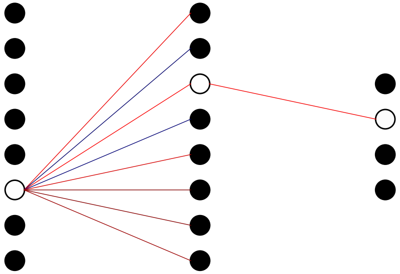
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



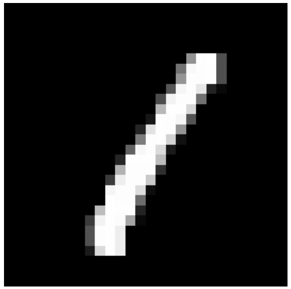


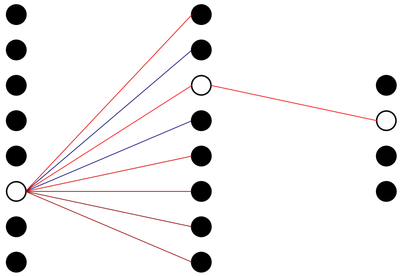
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



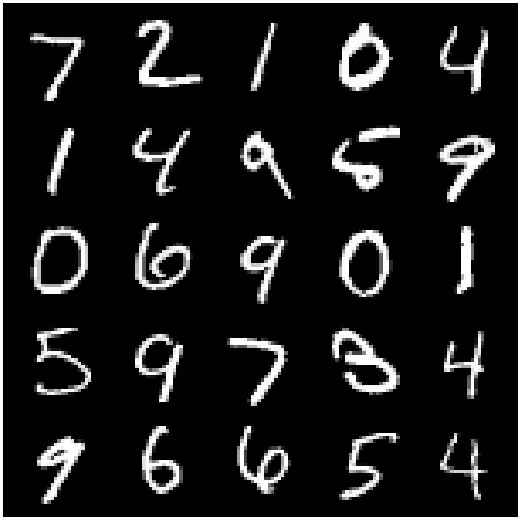


$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

Having discussed the theory behind Artificial Neural Networks and the training process, we will now proceed to demonstrate a comprehensive end-to-end example of image classification on MNIST.



The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

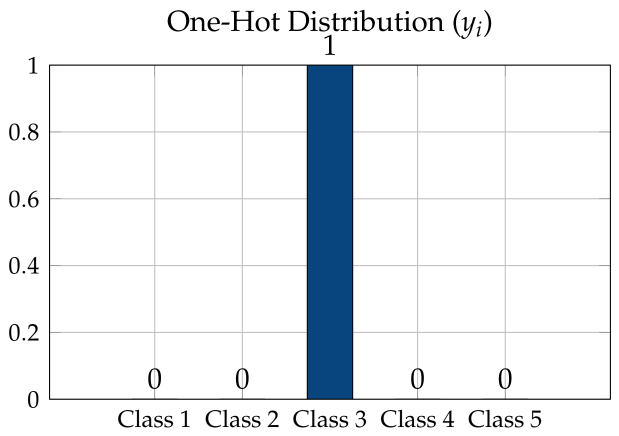
- Input shape : $1 \times 28 \times 28$.
- Number of Classes : 10.
- Number of training samples (x, y): 60000.
- Number of evaluating samples: 10000.
- Loss : cross-entropy

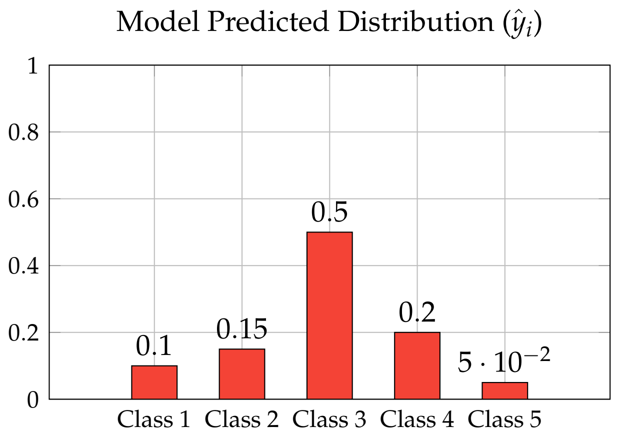
$$L(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij})$$

where :

- $\hat{y} \in \mathbb{R}^{N \times K}$ is the predicted probability distribution over K classes for N samples,
- $y \in 0, 1^{N \times K}$ is the ground-truth one-hot encoded label matrix,

Recap on the Cross-Entropy Loss





The cross-entropy loss for one sample is:

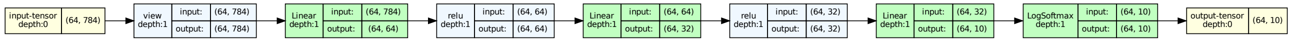
$$l(\hat{y}_i, y_i) = -\sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij}).$$

The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

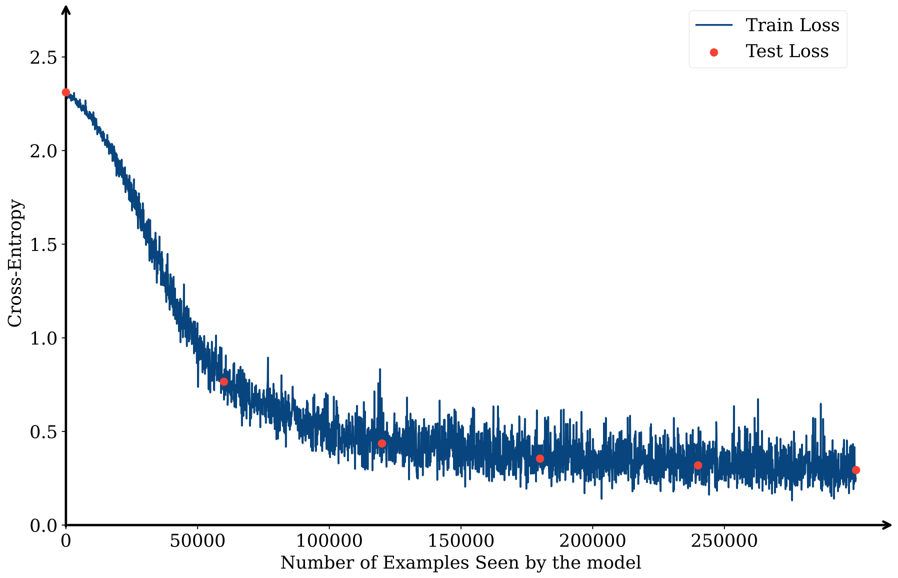
We build a 3 layers network.

- Batch size : 64
- Learning rate : 0.01
- Intermediate activation : ReLU
- Final activation : Softmax
- Number of epochs : 12
- Number of trained parameters: 52.6k


input-tensor depth:0 (64, 784)
view depth:1 input: (64, 784) output: (64, 784)
Linear depth:1 input: (64, 784) output: (64, 64)
relu depth:1 input: (64, 64) output: (64, 64)
Linear depth:1 input: (64, 64) output: (64, 32)
relu depth:1 input: (64, 32) output: (64, 32)
Linear depth:1 input: (64, 32) output: (64, 10)
LogSoftmax depth:1 input: (64, 10) output: (64, 10)
output-tensor depth:0 (64, 10)

The Multi-layer Perceptron (MLP)

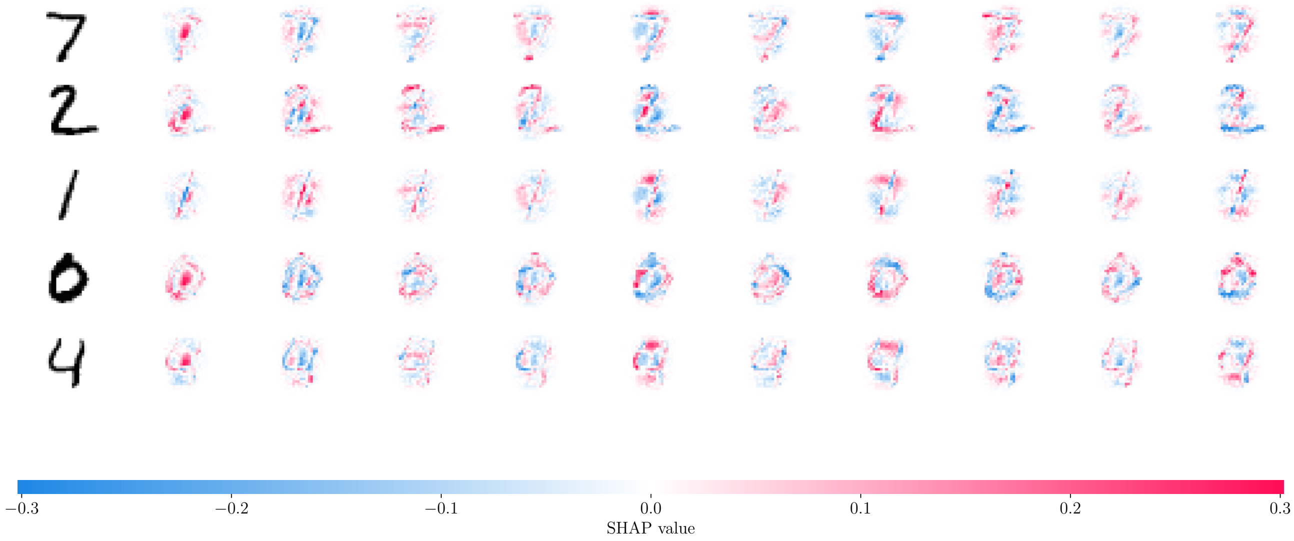
Example : Image classification of handwritten digits from A to Z



The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

With an interpretation tool such as SHAP:



TP1: The Multi-layer Perceptron (MLP)

The first Deep Learning Model

Link to the notebook (ipynb): TP1.ipynb
Link to the notebook (html): TP1.html