# Dive into Deep Learning: A Comprehensive Introduction

## From AI Fundamentals to Cutting-Edge Deep Learning Techniques

**Alexandre Vérine,**
**PhD Student at LAMSADE**

Executive Master IASD
Université Paris-Dauphine, PSL

March 18, 2024

# AI 101: From Fundamentals to Deep Learning

**1 Introduction to Artificial Intelligence . . . 5**

1.1 Deep Learning in the AI family . . . 5

1.2 Representation Learning . . . 10

**2 Neural Networks Fundamentals . . . 15**

2.1 Neurons . . . 16

2.2 Layers . . . 18

2.3 Activation Functions . . . 20

**3 The Multi-layer Perceptron (MLP) . . . 29**

3.1 The first Deep Learning Model . . . 30

3.2 Stochastic Gradient Descent . . . 31

3.3 Back-propagation . . . 33

3.4 Example : Image classification of handwritten digits from A to Z . . . 51

# Deep Learning in Action: From Neural Networks to Transformer Models


**1 Convolutional Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . 58**

1.1 The Two dimensional Convolution . . . . . . . . . . . . . . . . . . . . . . . . . 59

1.2 CNN : Convolutional in a network Networks . . . . . . . . . . . . . . . . . . . . 67

1.3 CNN in practice: CIFAR 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

**2 Recurrent Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94**

2.1 Recurrent Block . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95

2.2 LSTM and GRU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97

**3 Transformer and Attention Mechanism . . . . . . . . . . . . . . . . . . . . . . 107**

3.1 Self-Attention Mechanism . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

3.2 Transformers Model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

# Advanced Deep Learning Techniques: Reinforcement Learning, GANs and Beyond


**1 Deep Reinforcement Learning . . . 114**

1.1 Deep Q-Learning . . . 114

1.2 The Cheese Game . . . 116

**2 Generative Adversarial Networks . . . 121**

2.1 GANS Models . . . 121

2.2 MNIST Generation . . . 123

**3 Sentiment Analysis . . . 126**

3.1 Bert . . . 126

3.2 Sentiment Analysis . . . 129

# 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, ...

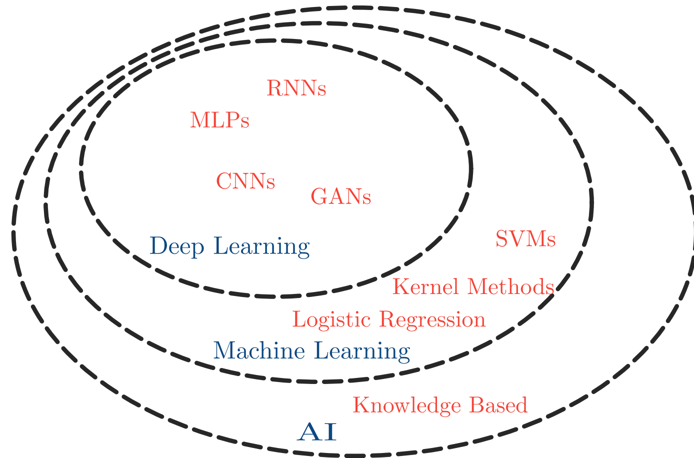
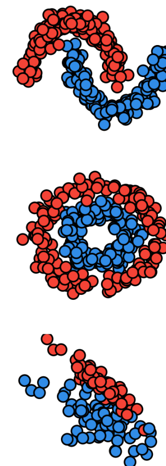



**Figure.** Subsets of Artificial Intelligence

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.



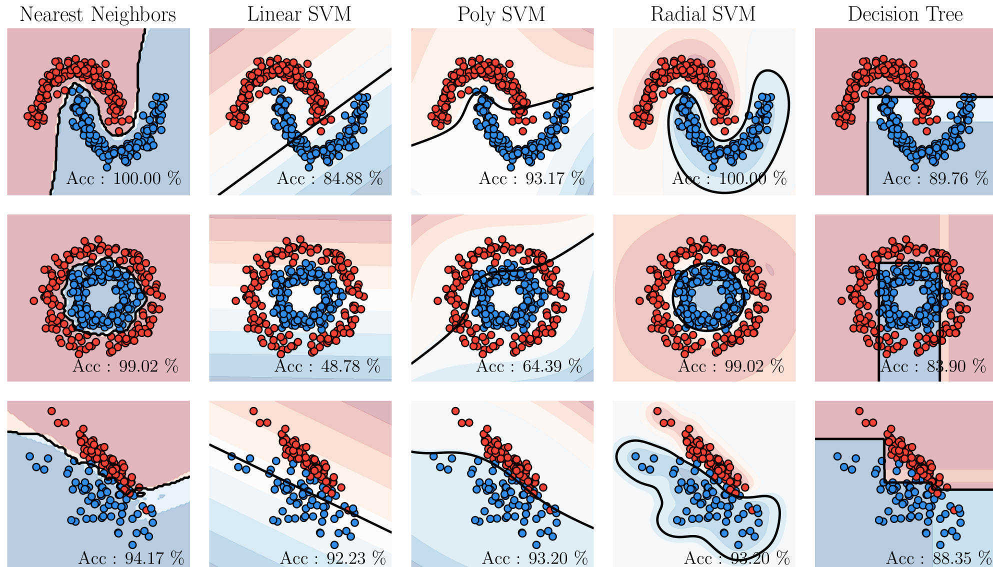



**Figure.** 2D classification for different AI models.

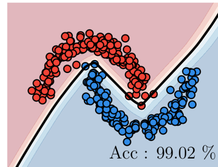
The Universal Approximation Theorem is a fundamental result in the field of artificial neural networks. It states that a deep learning model can approximate any function.

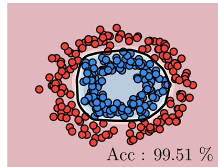
### Theorem 1 (Universal Approximation Theorem)

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



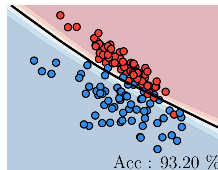
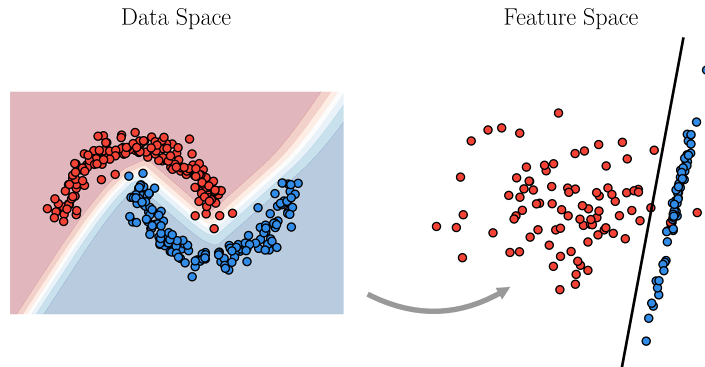




**Figure.** 2D classification for small Neural Network.

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.



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.

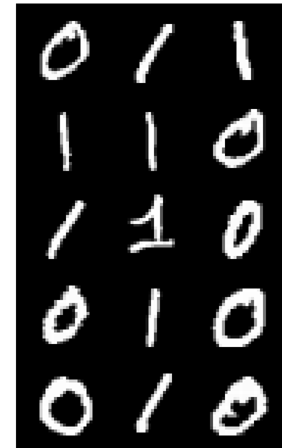


**Figure.** MNIST : Data

# Introduction to Artificial Intelligence

## Example of Representation Learning

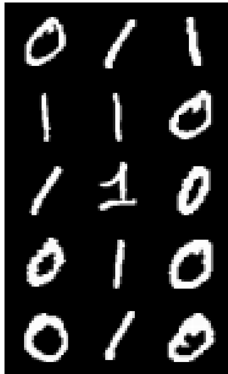


**Figure.** MNIST : Layer 0

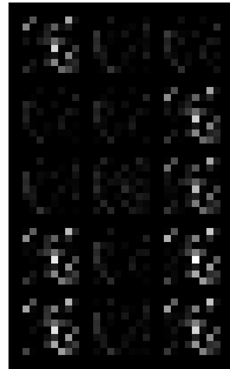


**Figure.** MNIST : Layer 1

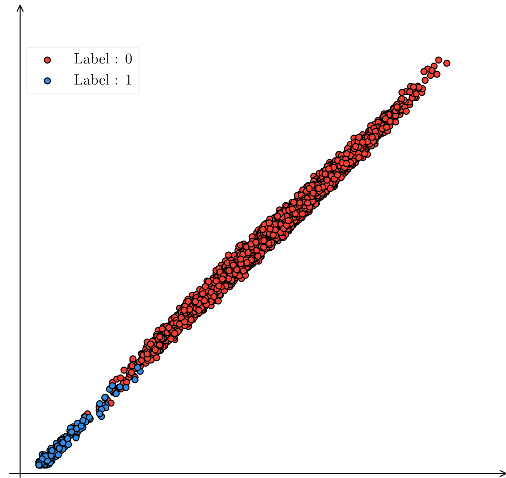



**Figure.** MNIST : Layer 2

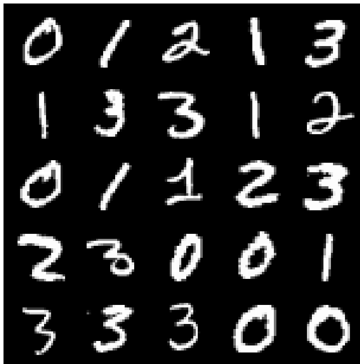


**Figure.** MNIST : Layer 0

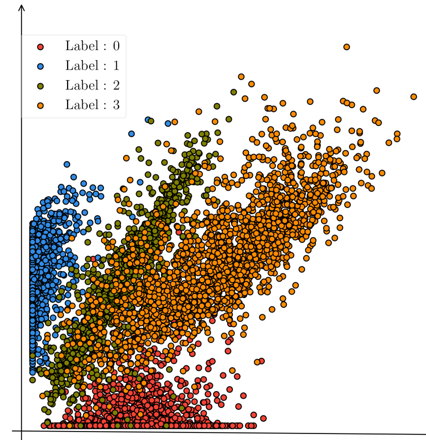


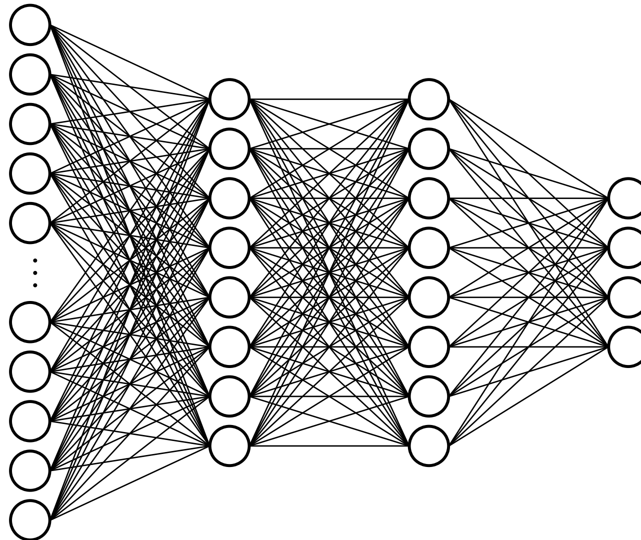

**Figure.** MNIST : Layer 2

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.

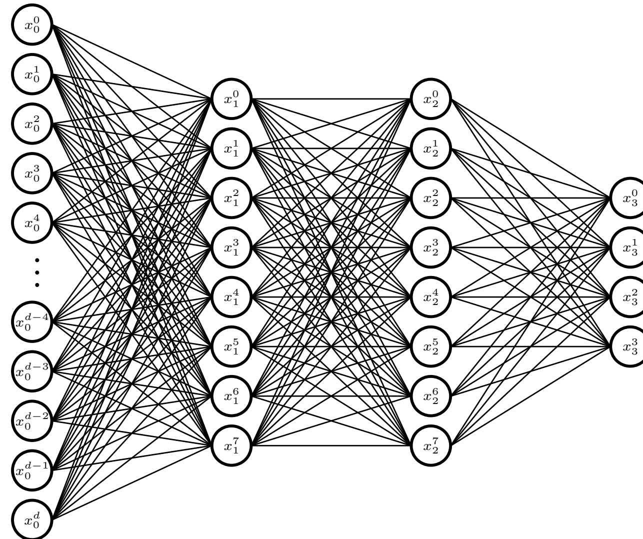


Let's see what the interconnected nodes, the layers and the transformations are.

# Neural Networks Fundamentals

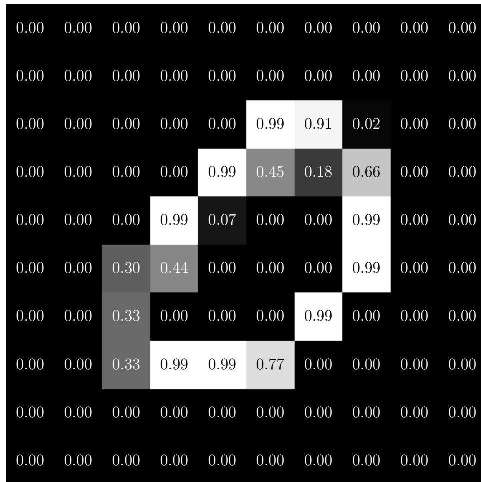
## Neurons

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 $x_i^k$.

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




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

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

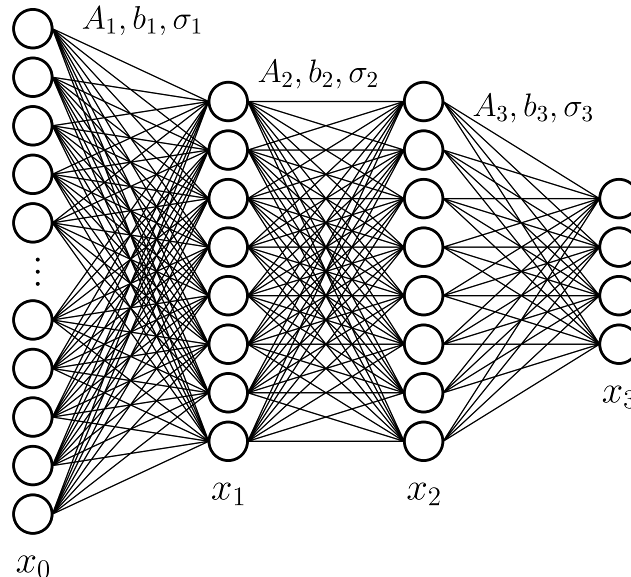
# Neural Networks Fundamentals

## 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 $\sigma_i$ the activation function.

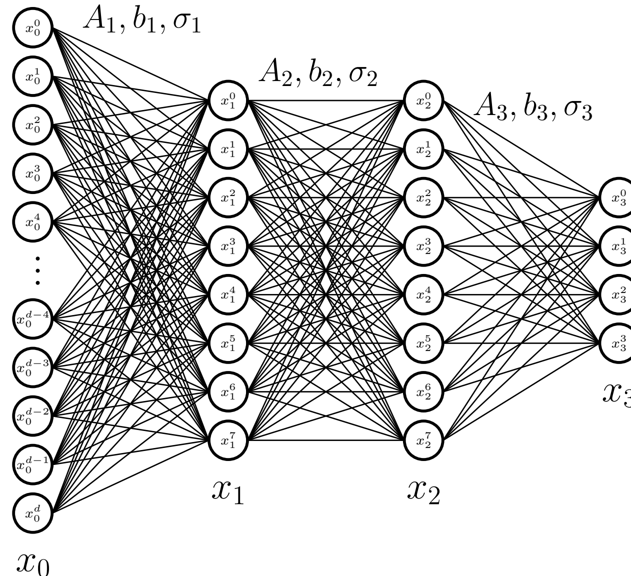


# Neural Networks Fundamentals
## Layers

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 $\sigma_i$ 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

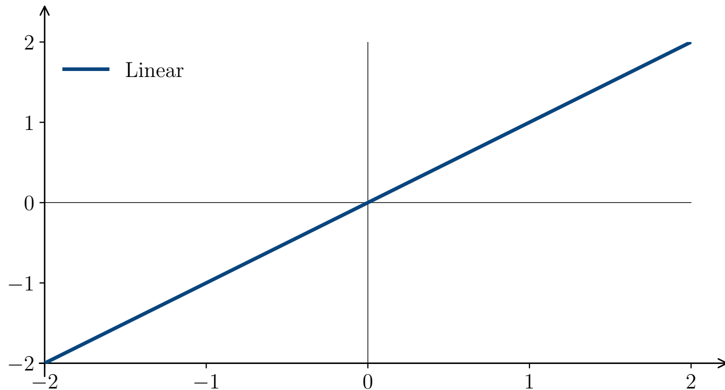
# Neural Networks Fundamentals

## Linear

- Linear activation Function:

$$\sigma(x) = x$$

- Final activation
- Use case : Regression



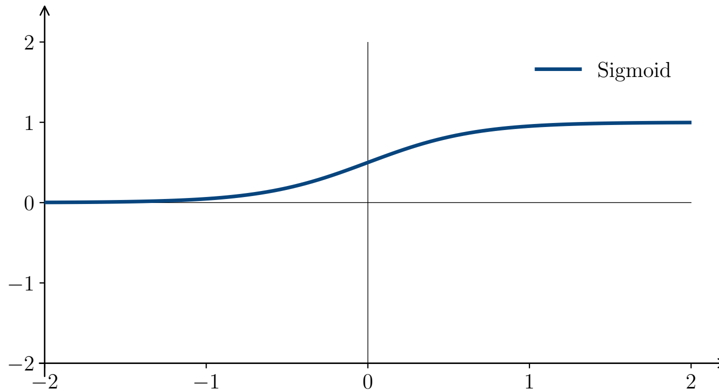
# Neural Networks Fundamentals

## Sigmoid

- Sigmoid Function:

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

- Final activation
- Use case : Classification

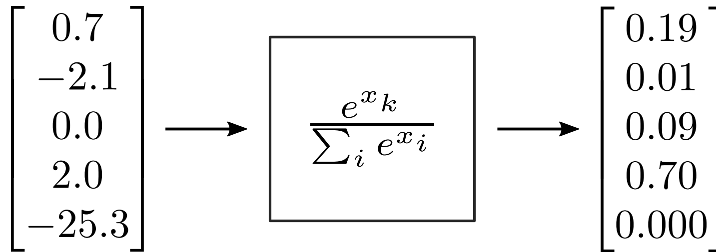


# Neural Networks Fundamentals
## Softmax

- Softmax Function:

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

- Final activation
- Use case : Multi-class Classification



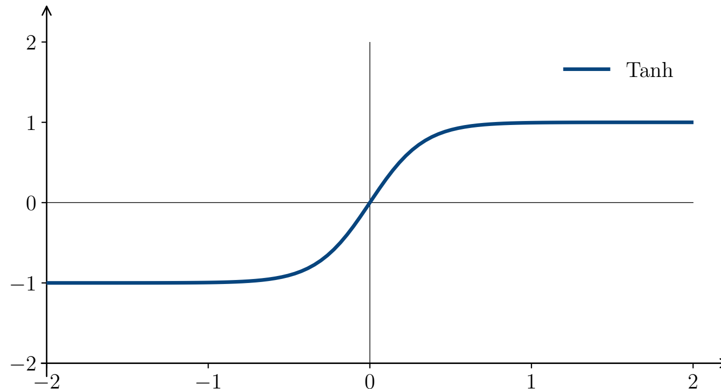
# Neural Networks Fundamentals

## Hyperbolic Tangent

- Hyperbolic Tangent

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

- Final activation
- Use case : Generative task

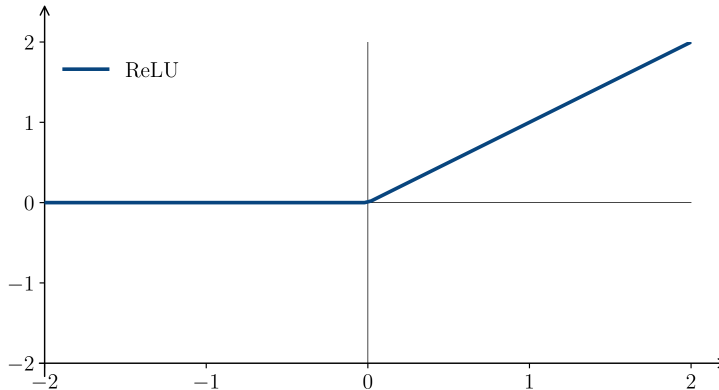


# Neural Networks Fundamentals
## ReLU

- Rectified Linear Unit (ReLU):

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

- Intermediate activation



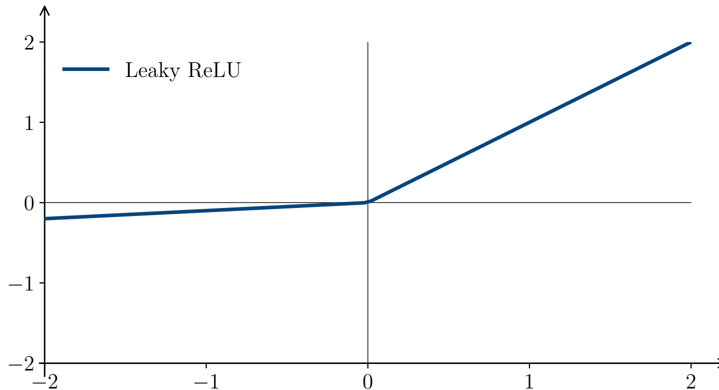
# Neural Networks Fundamentals

## Leaky ReLU

- Leaky Rectified Linear Unit (Leaky ReLU):

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

- Intermediate activation

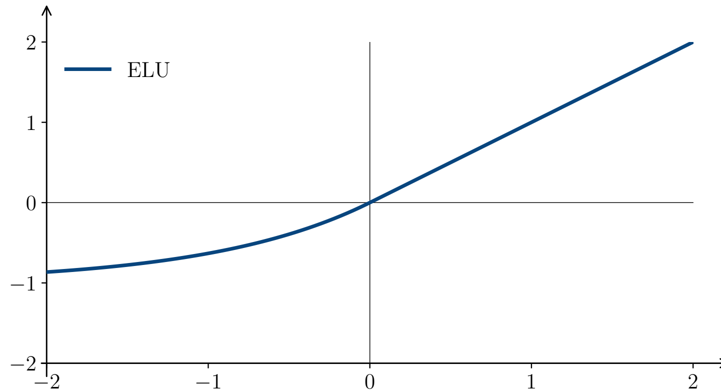


# Neural Networks Fundamentals
## ELU

- 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



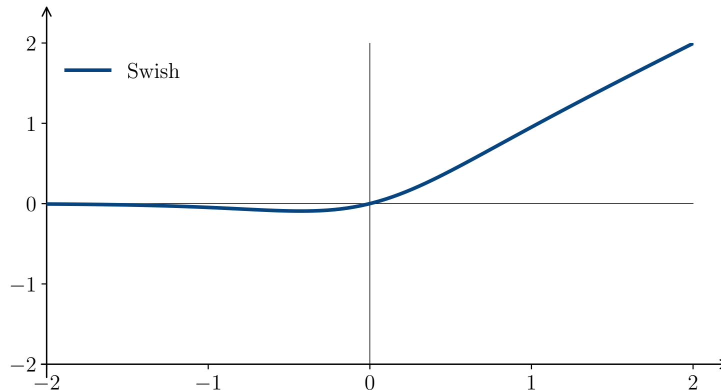
# Neural Networks Fundamentals

## Swish

- Sigmoid-Weighted Linear Unit (Swish):

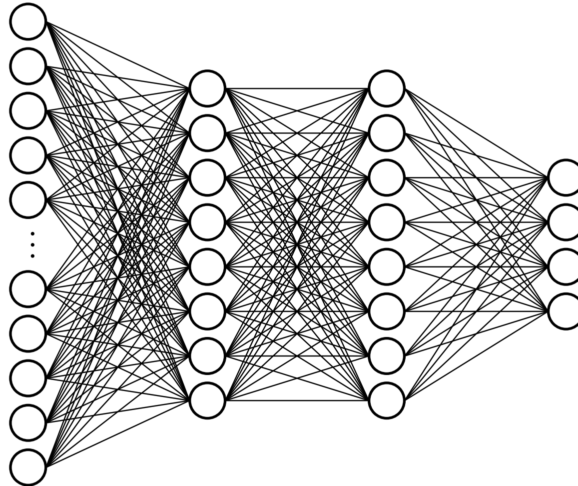
$$\sigma(x) = \frac{x}{1 + e^{-x}}$$

- 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.

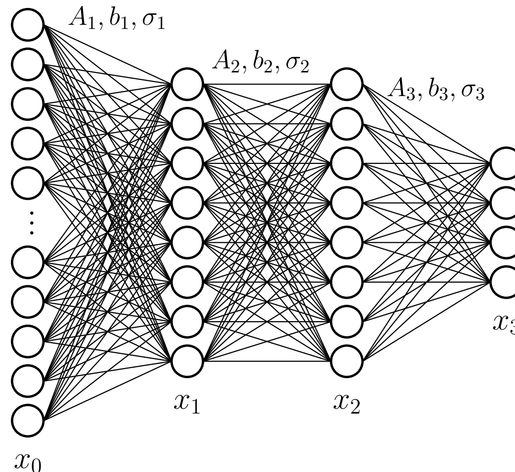


# The Multi-layer Perceptron (MLP)

## 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 $f_\theta$ 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

In practice, we

1: Given a loss function $l$ and a dataset $\mathcal{D} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}$
2: Initialize parameters $\theta$
3: Repeat until convergence:
4: **for** $i = 1$ to $N$ **do**
5: Randomly select $x_i$ from the dataset
6: Compute gradient of the loss with respect to $\theta$: $\nabla_\theta l(f(x_i), y_i)$
7: Update parameters $\theta = \theta - \lambda \nabla_\theta l(f(x_i), y_i)$
8: **end for**

Back Propagation is a widely used algorithm for training neural networks that leverages gradient descent optimization. It operates by computing the gradient of the loss function with respect to the model's parameters and then updating the parameters in the opposite direction of the gradient to reduce the loss. The algorithm utilizes the chain rule of differentiation to efficiently calculate the gradients through the network's many layers, making it well-suited for deep networks.

At every step $t$ of the gradient descent, setting a learning rate $\lambda$, the parameter $\theta$ is updated as:

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

# 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:

$$f(x) = x_3 = \sigma_3(A_3x_2 + b_3)$$

Therefore:

$$l(f(x)) = l\left(\sigma_3\left(A_3x_2 + b_3\right)\right)$$

To reduce 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 $A_3$:

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

and therefore:

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

We need to keep in memory the latent values of $x$.

# 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 $\frac{\partial l}{\partial x_2}$.

# The Multi-layer Perceptron (MLP)

## Back-propagation

We have to compute the gradient with respect to $x_2$:

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

Therefore:

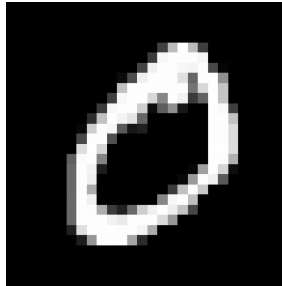
$$A_2 \leftarrow A_2 - \lambda\left[l'(x_3)A_3^T\sigma_3'\left(A_3 x_2 + b_3\right)\times\sigma_2'\left(A_2 x_1 + b_2\right)x_1^T\right]$$

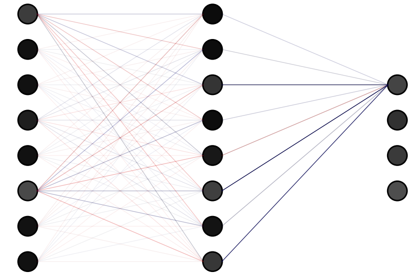
# The Multi-layer Perceptron (MLP)

## 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.





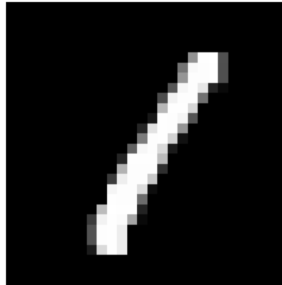
$$x_3^1 = \sigma_3 \left( A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

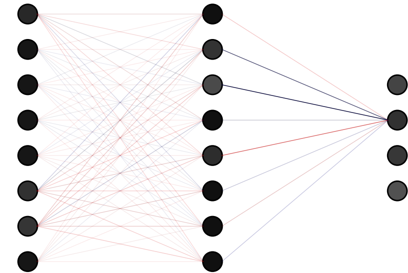
# The Multi-layer Perceptron (MLP)

## 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

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 :

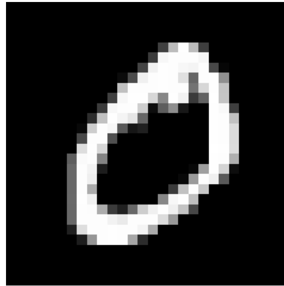
$$b_1 \leftarrow b_1 - \lambda \frac{\partial l}{\partial x_1}\sigma'\left(A_1 x_0 + b_1\right)$$

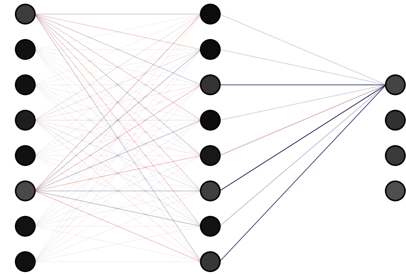
We need to back-propagate the term $\frac{\partial l}{\partial x_1}$ computed for the second layer which has been computed with $\frac{\partial l}{\partial x_2}$ back-propagated from the first layer.

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



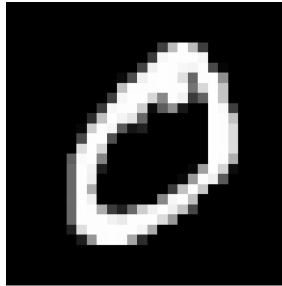


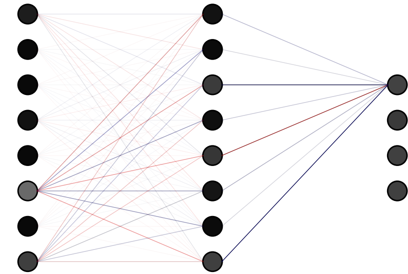
$$x_3^1 = \sigma_3 \left( A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



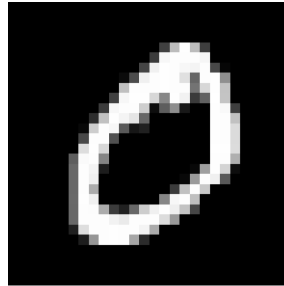


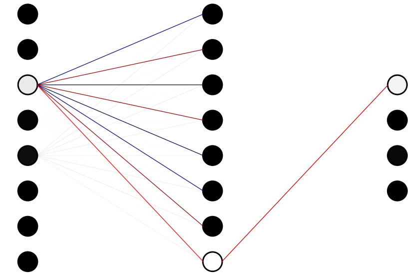
$$x_3^1 = \sigma_3 \left( A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



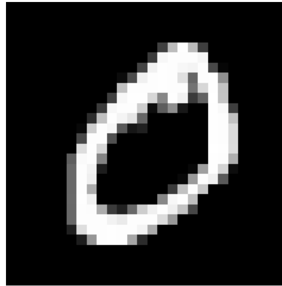


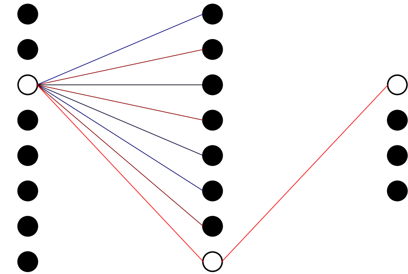
$$x_3^1 = \sigma_3 \left( A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



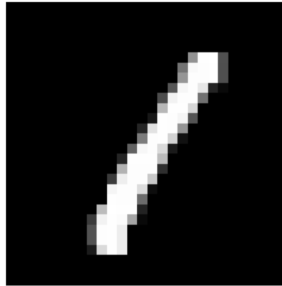


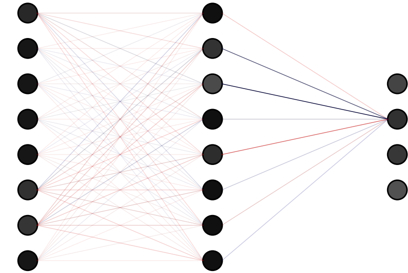
$$x_3^1 = \sigma_3 \left( A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



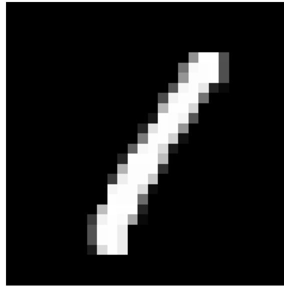


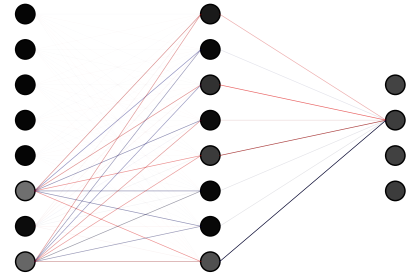
$$x_3^2 = \sigma_3 \left( A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



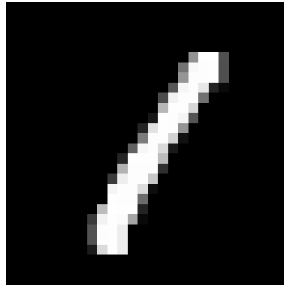


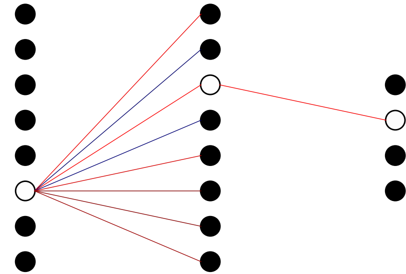
$$x_3^2 = \sigma_3 \left( A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



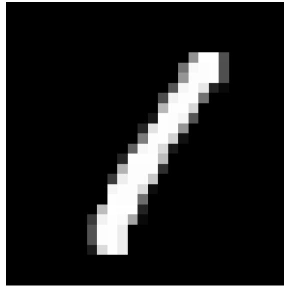


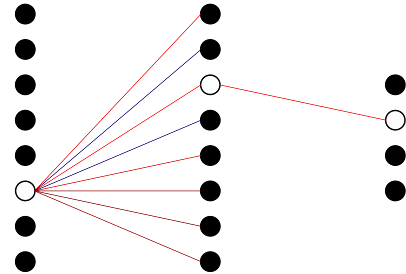
$$x_3^2 = \sigma_3 \left( A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

# The Multi-layer Perceptron (MLP)

## Back-propagation

Iteratively, the neural networks improves its performance.



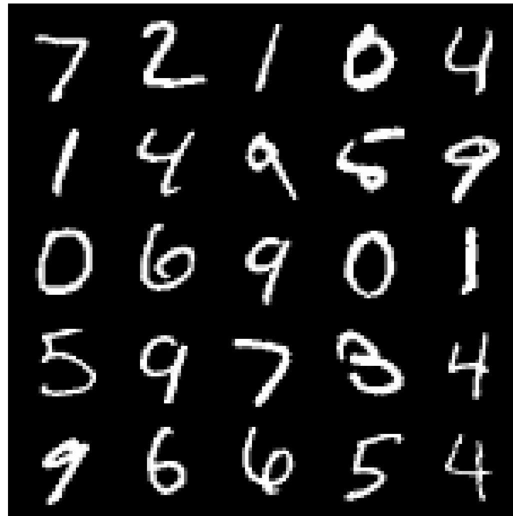


$$x_3^2 = \sigma_3 \left( A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

# 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 :

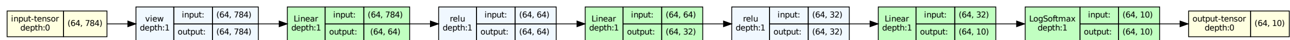
- $\hat{y} \in \mathbb{R}^{N \times K}$ is the predicted probability distribution over $K$ classes for $N$ samples,
- $y \in 0, 1^{N \times K}$ is the ground-truth one-hot encoded label matrix,

# The Multi-layer Perceptron (MLP)

## Example : Image classification of handwritten digits from A to Z

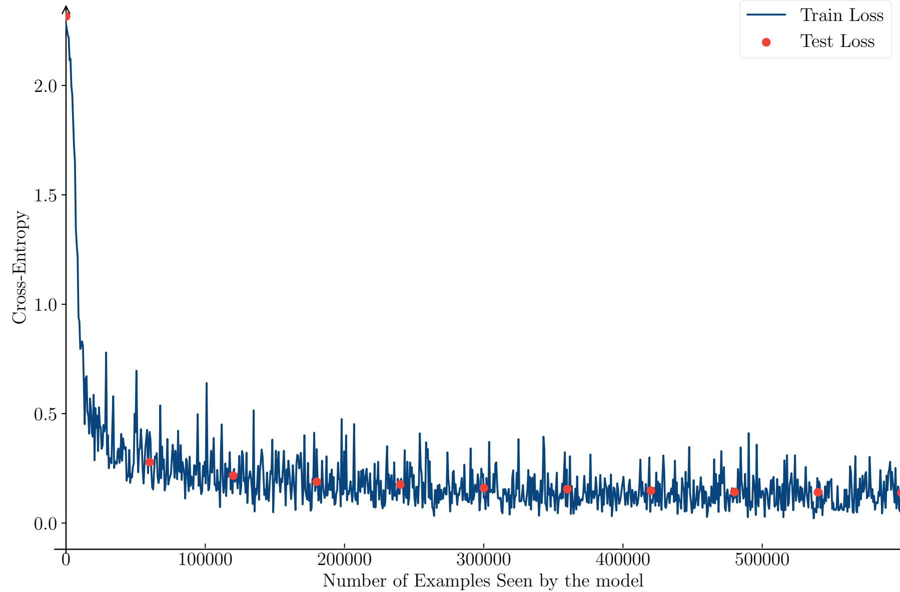
We build a 3 layers network.

- Batch size : 64
- Learning rate : 0.01
- Scheduler : we divide $\lambda$ by 2 every 3 epochs.
- Number of trained parameters: 52.6k



# The Multi-layer Perceptron (MLP)

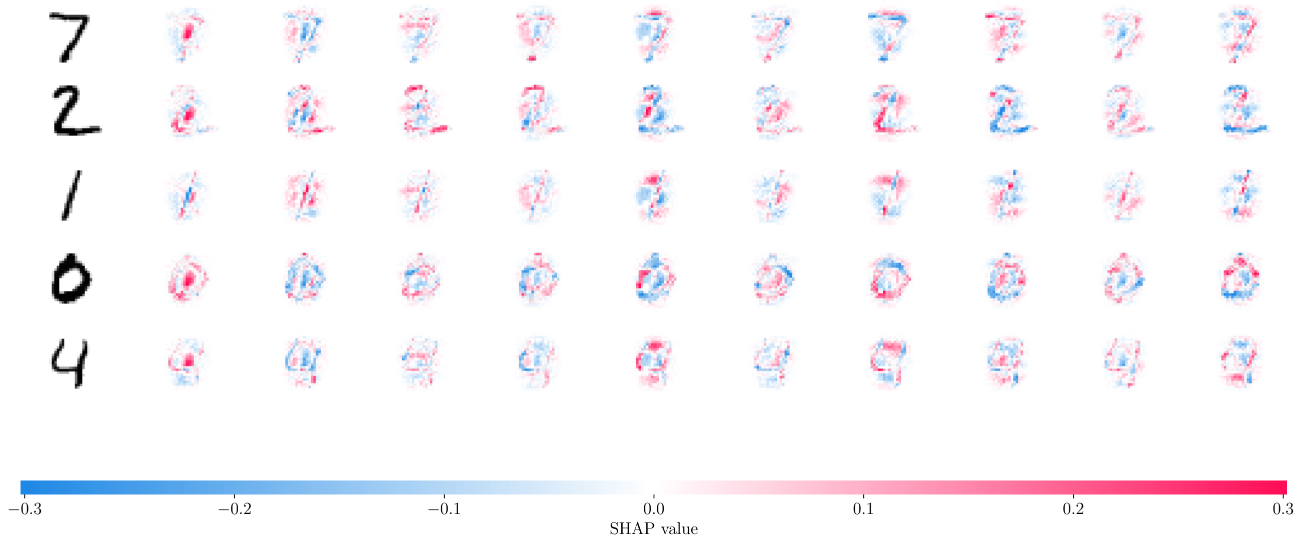
## Example : Image classification of handwritten digits from A to Z



# The Multi-layer Perceptron (MLP)

## Example : Image classification of handwritten digits from A to Z

With a interpretation tool such as SHAP:



# Part II

# Deep Learning in Action: From Neural Networks to Transformer Models

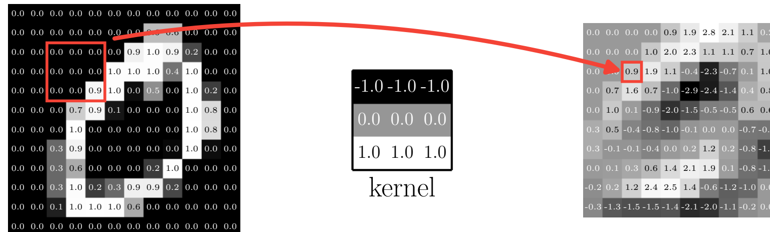
Now that we have an understanding of the training procedure for Artificial Neural Networks, we shall examine several widely-utilized structures within the literature of Neural Networks, including Convolutional Neural Networks (CNN),Resdiual Networks (ResNet), Recurrent Neural Networks (RNN), and Transformers.

# Convolutional Neural Networks

In the field of image processing, the Convolution Operators are widely considered as the most favoured approach. While it has been demonstrated that Dense blocks, or Linear blocks, are capable of accurately classifying images in the case of the MNIST dataset, the need for convolutional transformations arises when addressing wider and more intricate datasets.

A 2D convolution in a neural network context can be mathematically represented as a sliding window operation where a filter (also called kernel) $w$ of size $k \times k$ is applied to each $k \times k$ sub-matrix of the input matrix $x$. The operation can be defined as the element-wise multiplication of the filter $w$ and the sub-matrix followed by summing the results, i.e.

$$y_{i,j} = \sum_{m=1}^{k} \sum_{n}^{k} w_{m,n} \cdot x_{i+m,j+n}$$

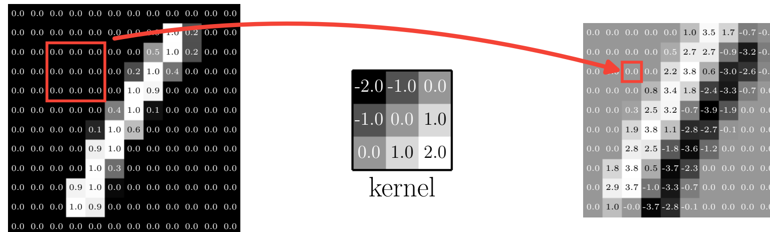


# Convolutional Neural Networks

## The Two dimensional Convolution

A 2D convolution in a neural network context can be mathematically represented as a sliding window operation where a filter (also called kernel) $w$ of size $k \times k$ is applied to each $k \times k$ submatrix of the input matrix $x$. The operation can be defined as the element-wise multiplication of the filter $w$ and the submatrix followed by summing the results, i.e.

$$y_{i,j} = \sum_{m=1}^{k} \sum_{n}^{k} w_{m,n} \cdot x_{i+m,j+n}$$



# Convolutional Neural Networks
## Kernel Size, Padding and Stride

In every Deep Learning library, the Conv2D block takes three parameters in argument:

- the Kernel's size,
- the Stride,
- the Padding.

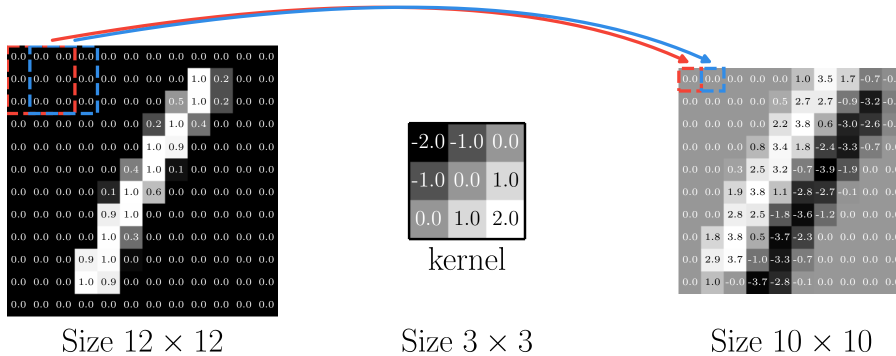
The size out the output is :

$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

# Convolutional Neural Networks

## Kernel Size, Padding and Stride

Kernel size: 3, Padding: 0, Stride: 1

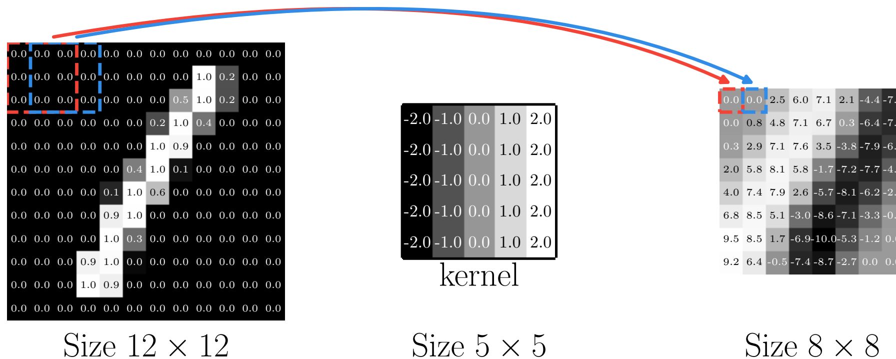



$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

# Convolutional Neural Networks

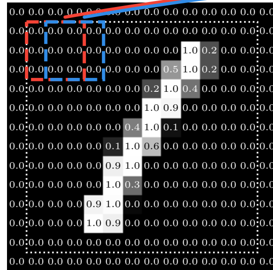
## Kernel Size, Padding and Stride

Kernel size: 5, Padding: 0, Stride: 1

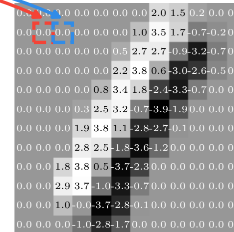



$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

Kernel size: 3, Padding: 1, Stride: 1. Padding mode can be 'zeros', 'reflect', 'replicate' or 'circular'.


Size 12 × 12

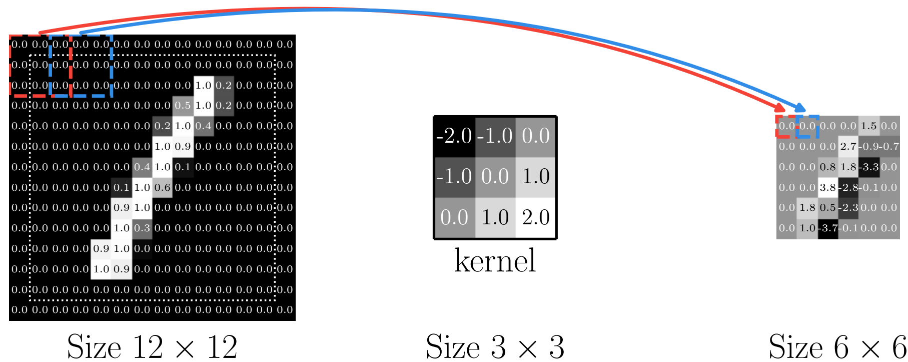
Size 3 × 3


Size 12 × 12

$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

# Convolutional Neural Networks
## Kernel Size, Padding and Stride

Kernel size: 3, Padding: 1, Stride: 2

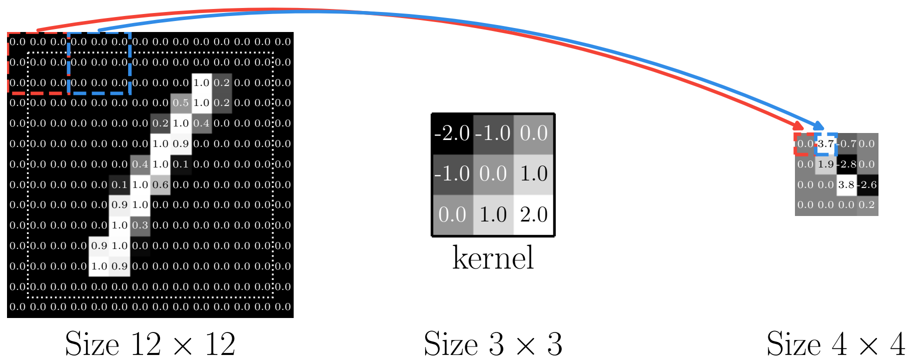



$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

# Convolutional Neural Networks

## Kernel Size, Padding and Stride

Kernel size: 3, Padding: 1, Stride: 3

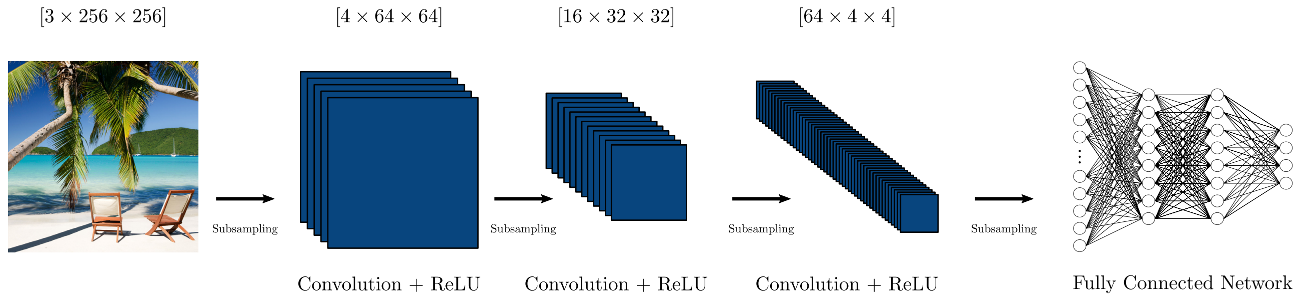



$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

# Convolutional Neural Networks

## CNN : Convolutional in a network Networks

We can represent a CNN as under this form:

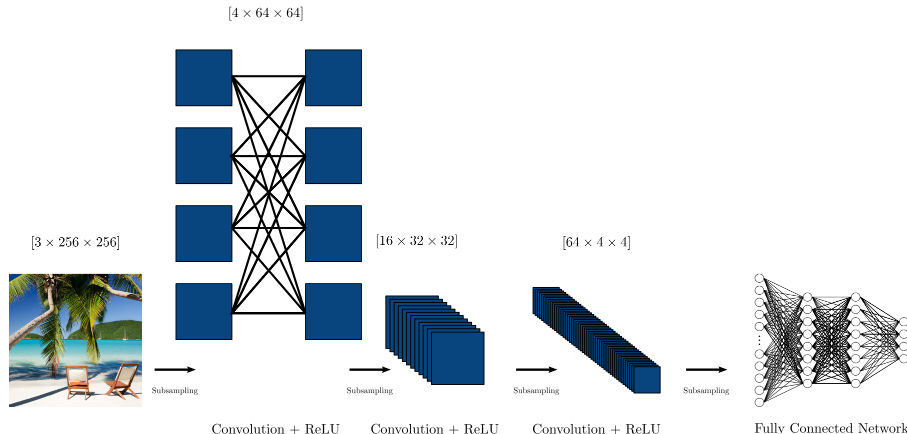


# Convolutional Neural Networks

## CNN : Convolutional in a network Networks

Usually, the output of a convolutional block is linear combination of the Convolutional output of every previous channels and a bias:

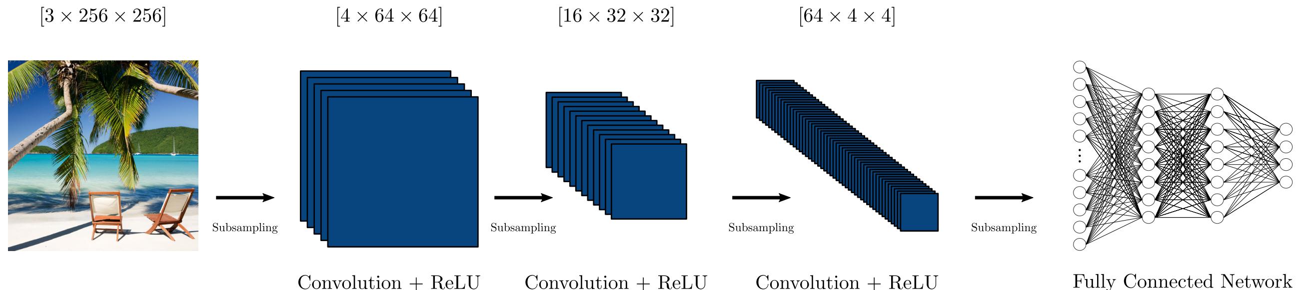
$$\text{out}_{i,j}(c_{\text{out}}) = \text{bias}(c_{\text{out}}) + \sum_{k=0}^{|c_{\text{in}}|-1} \text{Conv}(\text{input}(k), \text{kernel}_k)_{i,j}$$



# Convolutional Neural Networks

## CNN : Convolutional in a network Networks

In practice, we split the image into multiple channels : the three channels RGB to begin with. Then we apply convolutional operation on different scales and then we use a fully connected tail. To change the scale we can use different sub-sampling : Max pooling, Average pooling or Invertible pooling.



# Convolutional Neural Networks

## Max Pooling

Max Pooling take the maximum within a given sized sub-matrix. In practice, the matrix is size $2 \times 2$ in order to reduce the dimension by 4 and doubling the scale.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.2 | 1.0 | 0.3 | 0.8 | 0.1 | 0.8 | 0.6 | 0.9 |
| 0.7 | 0.3 | 0.3 | 0.5 | 0.4 | 0.3 | 0.3 | 0.4 |
| 0.2 | 0.6 | 0.7 | 0.9 | 0.9 | 0.1 | 0.3 | 0.5 |
| 0.8 | 0.7 | 0.1 | 0.3 | 0.3 | 0.6 | 0.9 | 0.5 |
| 0.7 | 0.1 | 0.1 | 0.2 | 0.8 | 0.4 | 0.9 | 0.7 |
| 0.9 | 0.1 | 0.8 | 0.9 | 0.6 | 0.3 | 0.1 | 0.9 |
| 0.8 | 0.7 | 0.2 | 0.7 | 0.0 | 0.6 | 0.9 | 0.5 |
| 0.9 | 0.5 | 0.1 | 0.2 | 0.0 | 0.1 | 0.1 | 0.4 |

Max Pooling → Subsampling

| | | | |
|---|---|---|---|
| 1.0 | 0.8 | 0.8 | 0.9 |
| 0.8 | 0.9 | 0.9 | 0.9 |
| 0.9 | 0.9 | 0.8 | 0.9 |
| 0.9 | 0.7 | 0.6 | 0.9 |

# Convolutional Neural Networks

## Average Pooling

The Average pooling takes the average value within the sub-matrix.

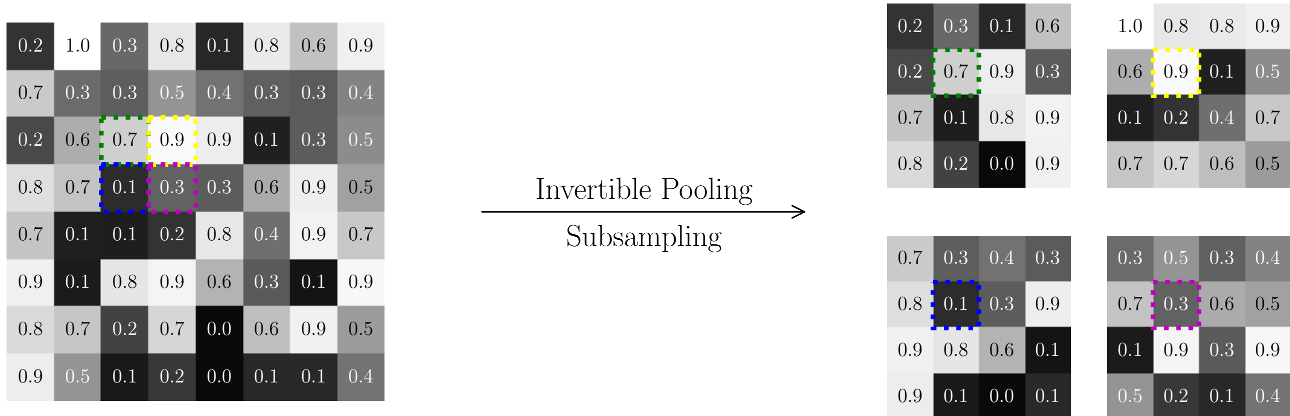
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0.2 | 1.0 | 0.3 | 0.8 | 0.1 | 0.8 | 0.6 | 0.9 |
| 0.7 | 0.3 | 0.3 | 0.5 | 0.4 | 0.3 | 0.3 | 0.4 |
| 0.2 | 0.6 | 0.7 | 0.9 | 0.9 | 0.1 | 0.3 | 0.5 |
| 0.8 | 0.7 | 0.1 | 0.3 | 0.3 | 0.6 | 0.9 | 0.5 |
| 0.7 | 0.1 | 0.1 | 0.2 | 0.8 | 0.4 | 0.9 | 0.7 |
| 0.9 | 0.1 | 0.8 | 0.9 | 0.6 | 0.3 | 0.1 | 0.9 |
| 0.8 | 0.7 | 0.2 | 0.7 | 0.0 | 0.6 | 0.9 | 0.5 |
| 0.9 | 0.5 | 0.1 | 0.2 | 0.0 | 0.1 | 0.1 | 0.4 |

Average Pooling →
Subsampling

| | | | |
|---|---|---|---|
| 0.5 | 0.5 | 0.4 | 0.6 |
| 0.6 | 0.5 | 0.5 | 0.5 |
| 0.4 | 0.5 | 0.5 | 0.6 |
| 0.7 | 0.3 | 0.2 | 0.5 |

# Convolutional Neural Networks
## Invertible Pooling

For Invertible Networks, we can use Invertible Pooling, aka Squeeze. It preserves the information contained in the channels and keeps the dimension constant.



Convolutional Neural Networks are more suitable for image processing compared to fully connected networks due to their ability to efficiently handle the spatial relationships between pixels in an image. This is achieved through the use of convolutional layers that apply filters to small portions of an image, rather than fully connected layers that process the entire image as a single vector. Additionally, the shared weights in convolutional layers allow for learning of hierarchical features, reducing the number of parameters in the network and increasing its ability to generalize to new images.

# Convolutional Neural Networks

## CNN in practice: CIFAR 10

- Input shape : $3 \times 32 \times 32$.
- Number of Classes : 10.
- Number of training samples $(x, y)$: 50000.
- Number of evaluating samples: 10000.

# Convolutional Neural Networks

## CNN in practice: CIFAR 10

We will compare three different models:

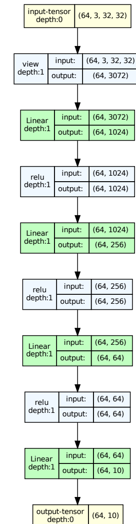
- Model 1 : Fully Connected Neural Network with 3.4 million parameters.
- Model 2 : CNN with 62 thousand parameters.
- Model 3 : Wider and longer CNN with 5.8 million parameters.

# Convolutional Neural Networks

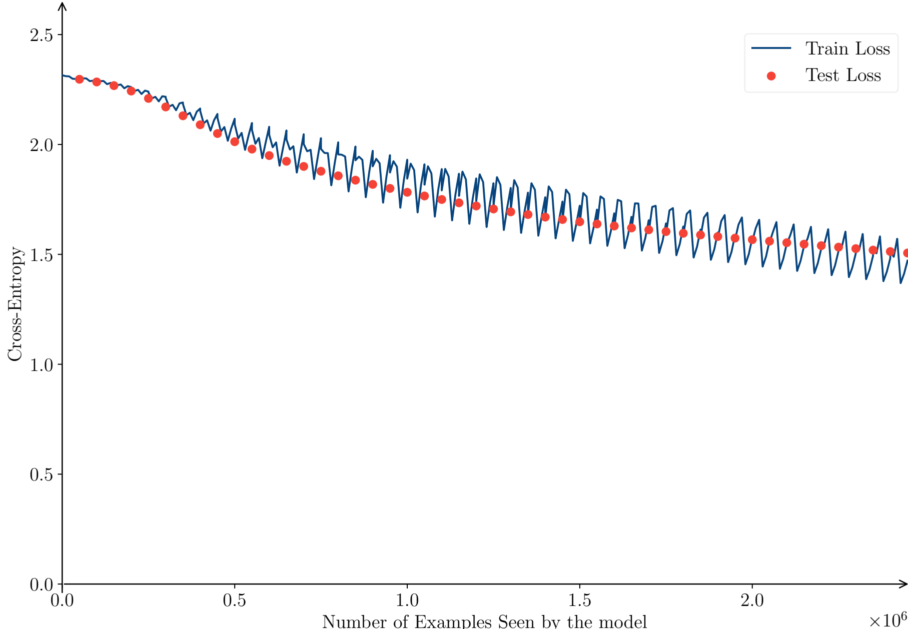
## Model 1

The Net in composed of 4 linear layers with ReLU activations:

- Linear 3072 $\mapsto$ 1024 + ReLU
- Linear 1024 $\mapsto$ 256 + ReLU
- Linear 256 $\mapsto$ 64 + ReLU
- Linear 64 $\mapsto$ 10 + SoftMax

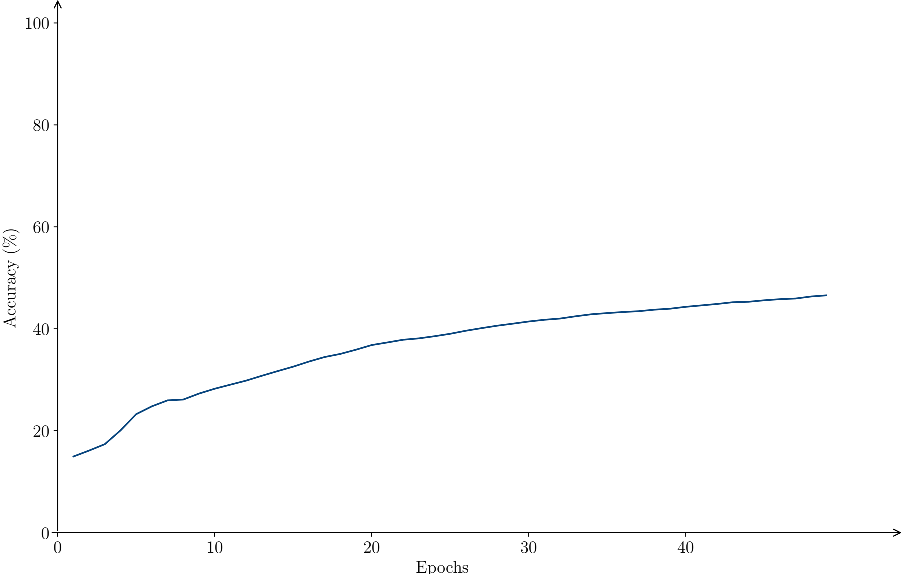


# Convolutional Neural Networks
## Model 1



# Convolutional Neural Networks

## Model 1

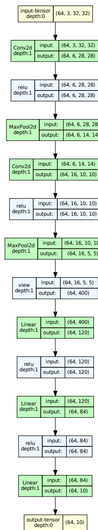


# Convolutional Neural Networks

## Model 2

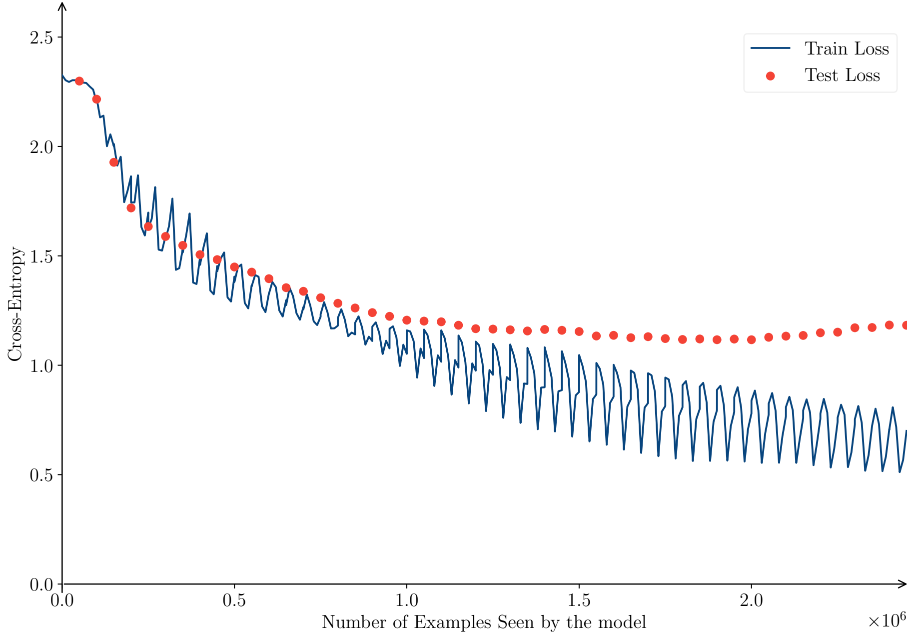
The Net is composed 2 convolutional layers and 2 linear layers:

- Conv $3 \times 32 \times 32 \mapsto 6 \times 28 \times 28$ + ReLU
- Max Pooling $6 \times 28 \times 28 \mapsto 6 \times 14 \times 14$
- Conv $6 \times 14 \times 14 \mapsto 16 \times 10 \times 10$ + ReLU
- Max Pooling $16 \times 10 \times 10 \mapsto 16 \times 5 \times 5$
- Linear $400 \mapsto 120$ + ReLU
- Linear $120 \mapsto 84$ + ReLU
- Linear $84 \mapsto 10$ + SoftMax



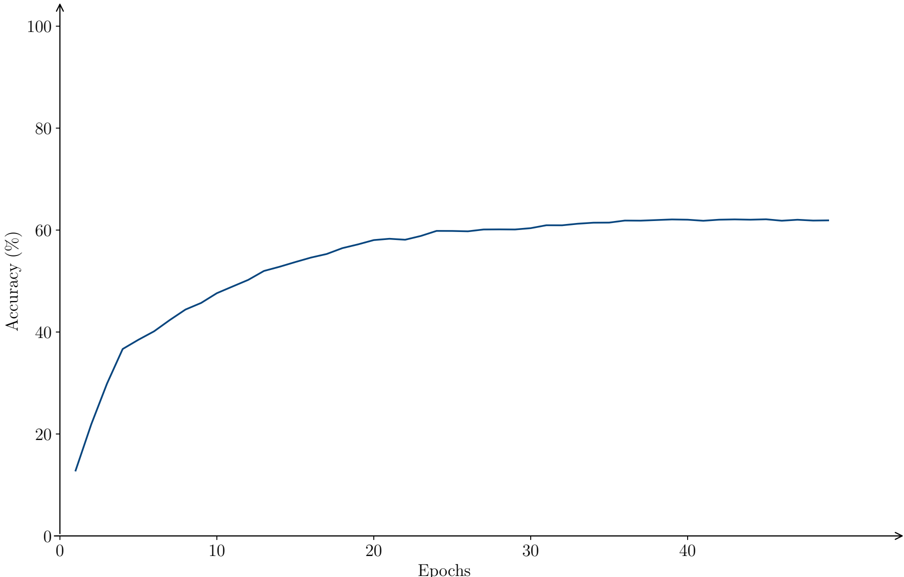
# Convolutional Neural Networks

## Model 2



# Convolutional Neural Networks

## Model 2

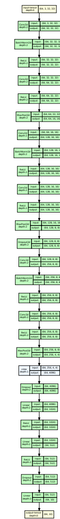


# Convolutional Neural Networks

## Model 3

The Net is composed 6 convolutional layers and 3 linear layers:

- Conv $3 \times 32 \times 32 \mapsto 32 \times 32 \times 32$ + BatchNorm2d + ReLU
- Conv $32 \times 32 \times 32 \mapsto 64 \times 32 \times 32$ + ReLU
- Max Pooling $64 \times 32 \times 32 \mapsto 64 \times 16 \times 16$
- Conv $64 \times 16 \times 16 \mapsto 128 \times 16 \times 16$ + BatchNorm2d + ReLU
- Conv $128 \times 16 \times 16 \mapsto 128 \times 16 \times 16$ + ReLU
- Max Pooling $128 \times 16 \times 16 \mapsto 128 \times 8 \times 8$
- Conv $128 \times 8 \times 8 \mapsto 256 \times 8 \times 8$ + BatchNorm2d + ReLU
- Conv $256 \times 8 \times 8 \mapsto 256 \times 8 \times 8$ + ReLU
- Max Pooling $256 \times 8 \times 8 \mapsto 256 \times 4 \times 4$ + DropOut $p = 0.05$
- Linear $4096 \mapsto 1024$ + ReLU
- Linear $1024 \mapsto 512$ + ReLU + DropOut $p = 0.05$
- Linear $512 \mapsto 10$ + SoftMax

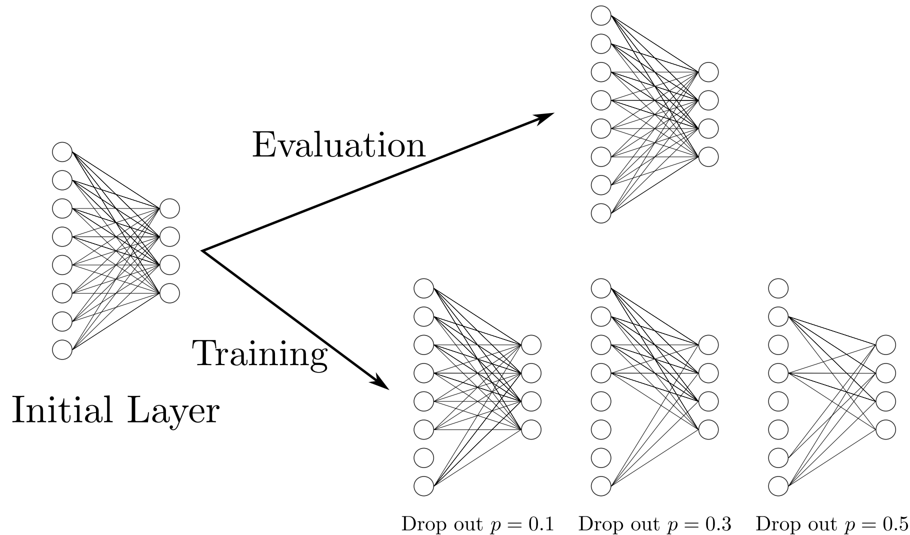


We have added Batch Normalization to improve the training stability and Drop Out to reduce overfitting.

# Convolutional Neural Networks

## Drop out

Dropout is a regularization technique in neural networks where during training, a portion of the nodes are randomly "dropped out" or ignored during each iteration. This helps prevent over-fitting by preventing the model from relying too heavily on any one node. The result is a more robust and generalizable model that can better handle unseen data.



Batch normalization is a technique in deep learning that is used to normalize the activations of a layer within a batch of data. This helps to prevent the problem of vanishing or exploding gradients and also speeds up the training process. By normalizing the activations, batch normalization helps to stabilize the distribution of the inputs to each layer, reducing the covariate shift and allowing the network to learn more effectively.

1: **for** each $x_i$ in a mini-batch $B$ of size $b$ **do**
2: Compute the mean $\mu_B$ and variance $\sigma_B^2$ of the features in the mini-batch $B$.

$$\mu_B = \frac{1}{b}\sum_i x_i \quad \text{and} \quad \sigma_B^2 = \frac{1}{m}\sum_i (x_i - \mu_B)^2$$

3: Normalize each feature $x_i$ in the mini-batch $B$ using $\mu_B$ and $\sigma_B^2$.

$$\bar{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$

4: Scale and shift each normalized feature $x_i$ using two learnable parameters $\gamma$ and $\beta$ respectively.

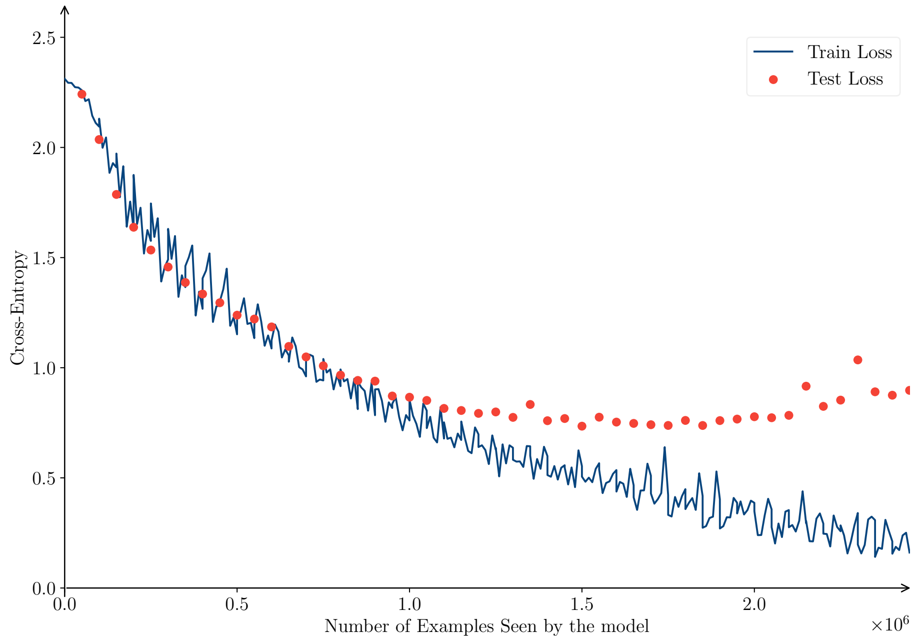
$$y_i = \gamma \bar{x}_i + \beta$$

5: **end for**

**Algorithm 1:** Batch Normalization

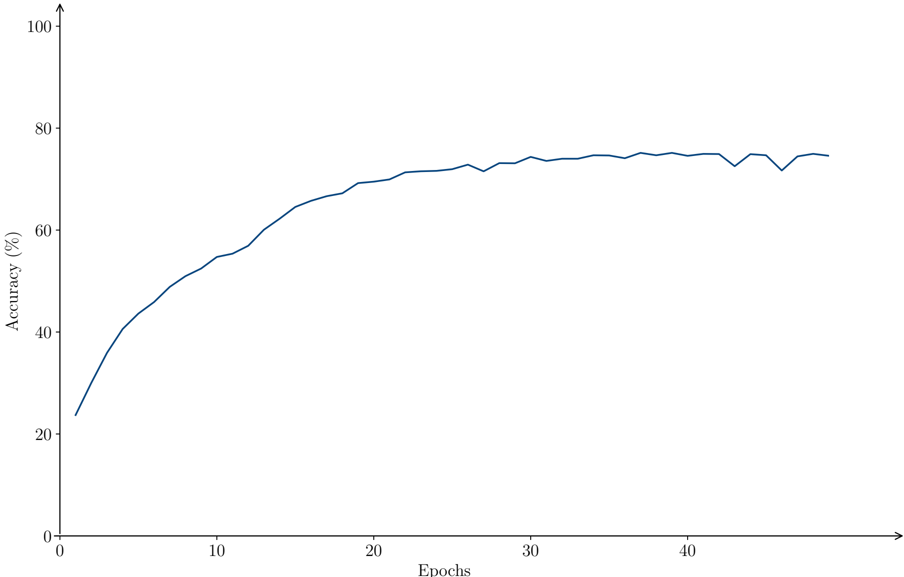
# Convolutional Neural Networks

## Model 3



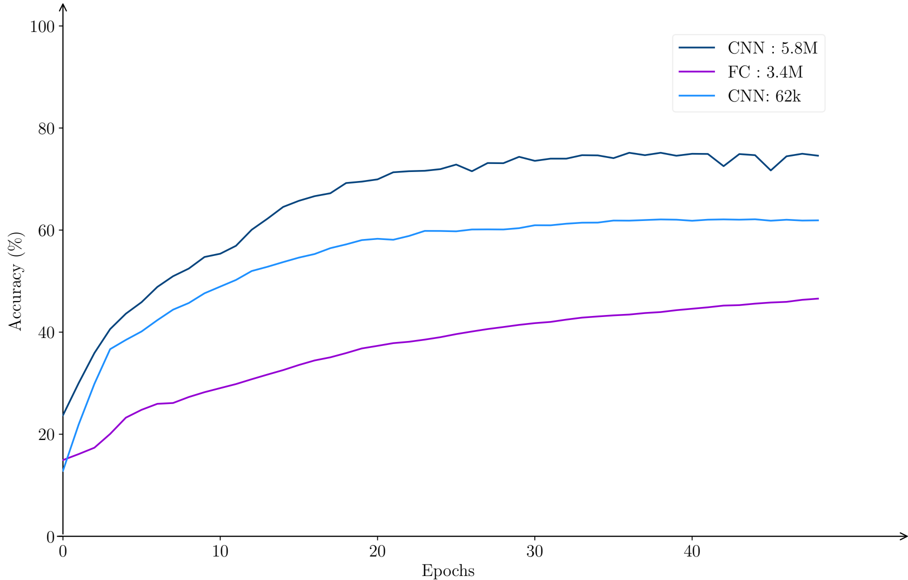
# Convolutional Neural Networks

## Model 3



# Convolutional Neural Networks

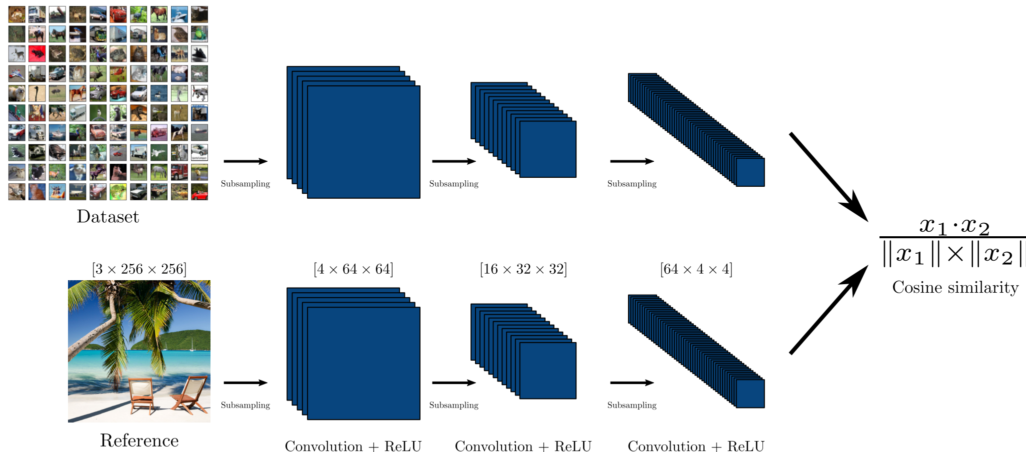
## CNN in practice: CIFAR 10



# Convolutional Neural Networks

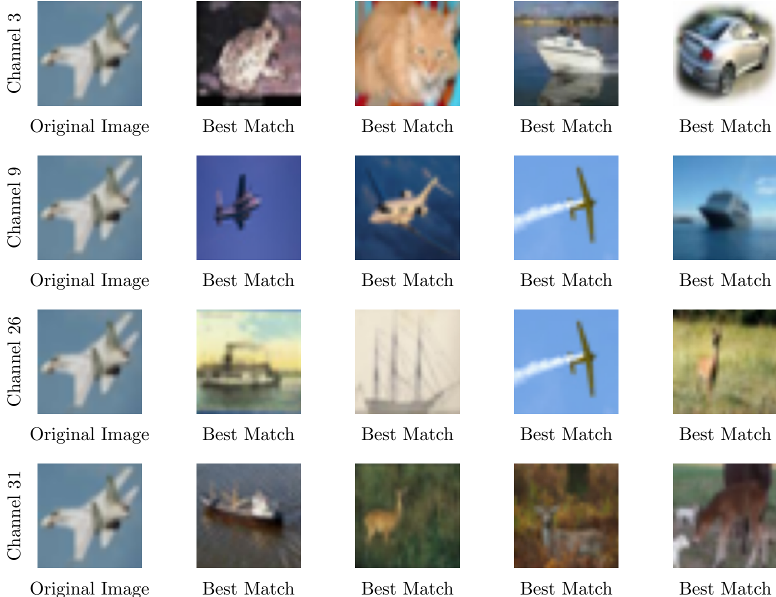
## Intuition behind channels

To examine the information captured by different channels in a Neural Network, we can compare their output on a dataset. For a given input $x$, we can compute the similarity between the output of a specific channel and the same channel for other images in the dataset.



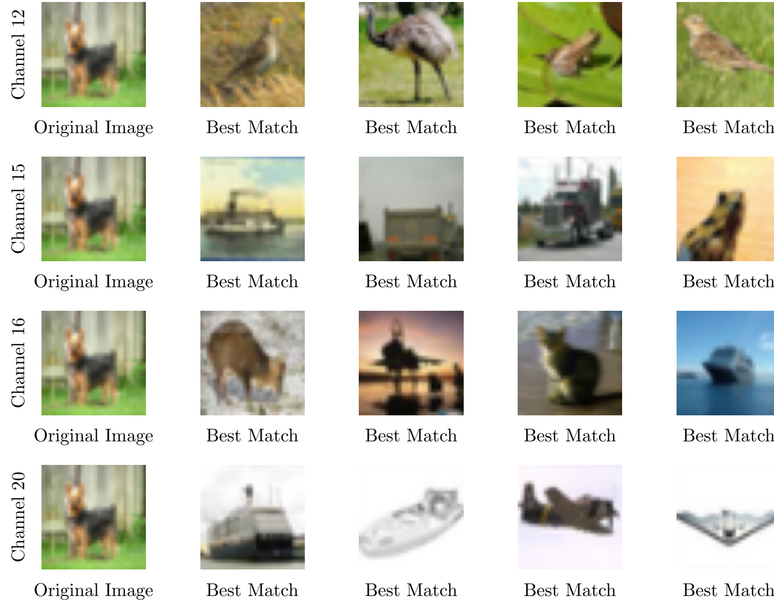
# Convolutional Neural Networks

## Intuition behind channels



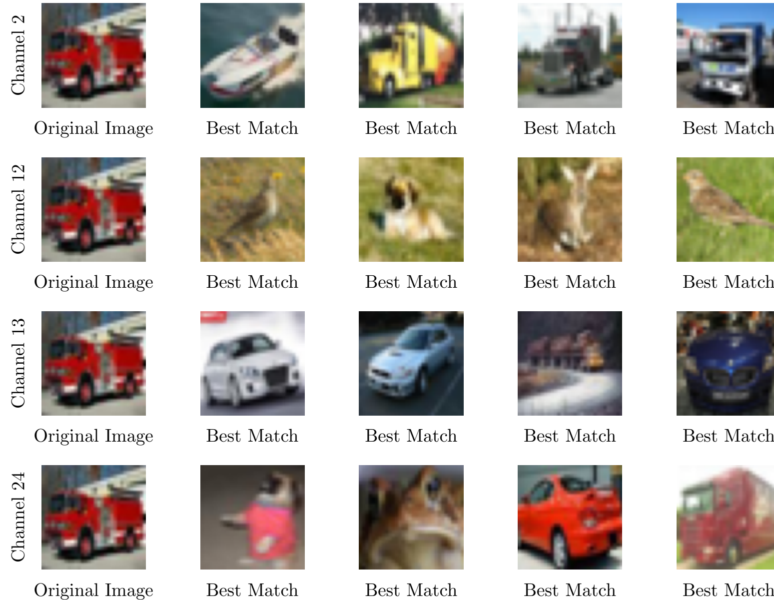
# Convolutional Neural Networks

## Intuition behind channels



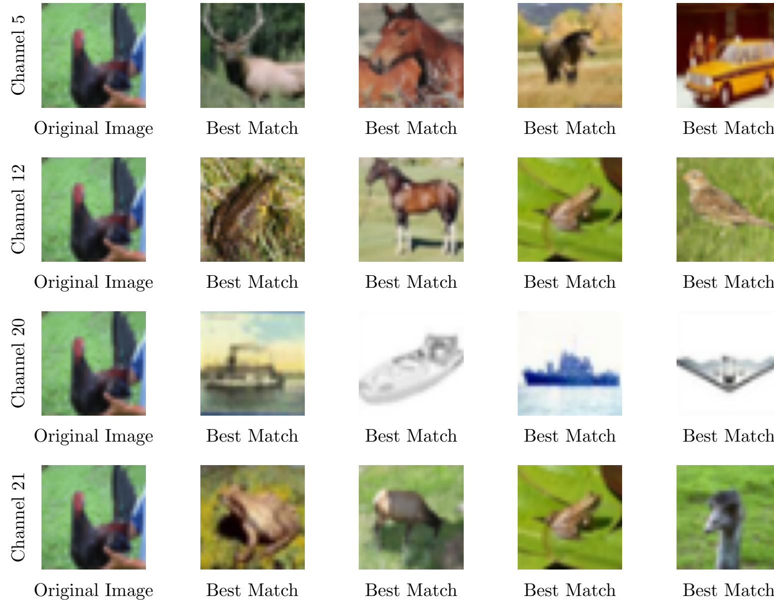
# Convolutional Neural Networks

## Intuition behind channels



# Convolutional Neural Networks

## Intuition behind channels



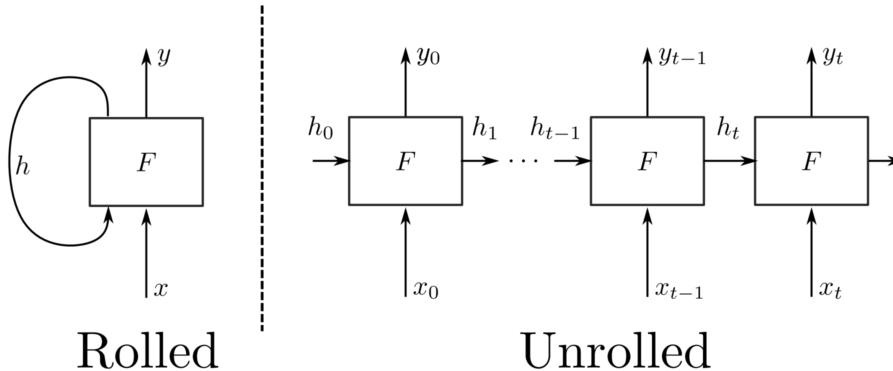
# Recurrent Neural Networks

Recurrent Networks (RNNs) are a type of neural network that are specifically designed to handle sequential data, whereas CNNs are more suited for image and grid-like data. The main difference between RNNs and CNNs lies in the way they process data, with RNNs considering the sequence of elements and their interdependencies, while CNNs focus on capturing local patterns within the input.

# Recurrent Neural Networks

## Recurrent Block

A Recurrent Network is a type of neural network that contains a loop mechanism, allowing previous outputs to be used as inputs for future computations. This creates a form of memory that allows the network to process sequential data with variable-length sequences.

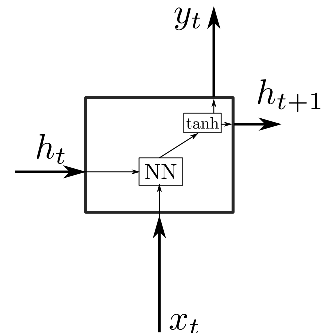


# Recurrent Neural Networks

## Recurrent Block

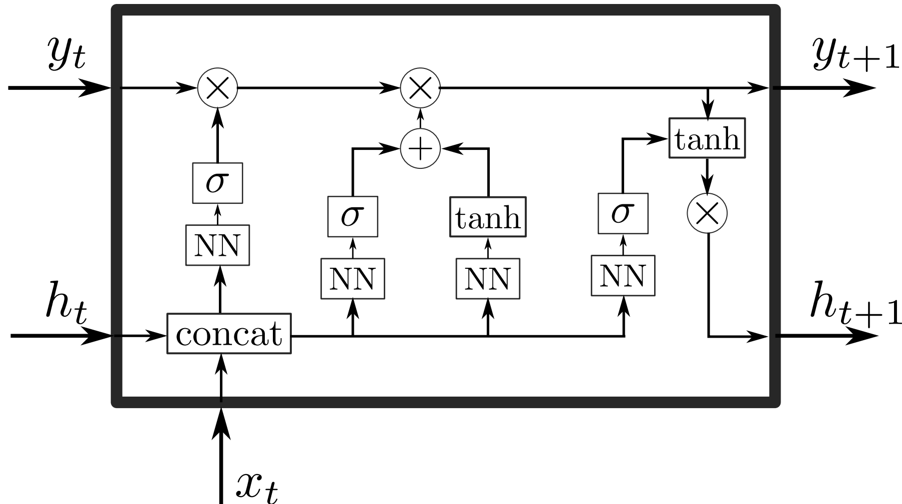
Some of the limitations of Vanilla RNNs:

- Vanishing gradient problem: The gradient signals used to update the weights during training can become very small, making it difficult to train RNNs effectively.
- Exploding gradient problem: On the other hand, gradients can become too large and cause numeric instability, making it difficult to train RNNs effectively.
- Short-term memory: Vanilla RNNs have difficulty retaining information over long periods of time, making them unsuitable for tasks that require remembering information from previous time steps.
- Computational limitations: RNNs can be computationally intensive, making it difficult to apply them to large sequences of data.
- Difficulty with parallelization: The sequential nature of RNNs can make it difficult to take advantage of parallel processing to speed up training and inference.



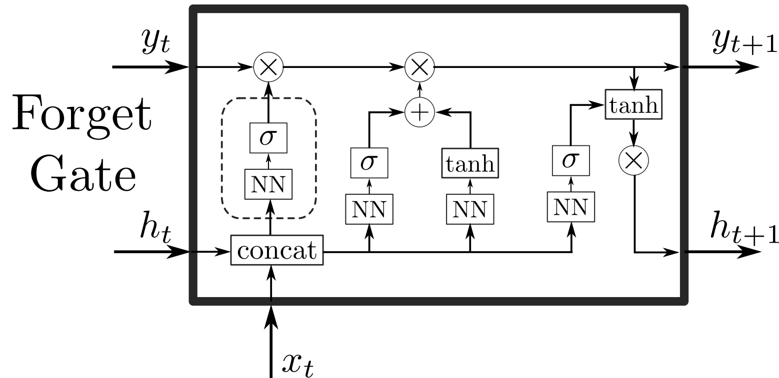
# Recurrent Neural Networks
## LSTM

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for updating cells, which allows them to selectively preserve information from previous time steps.



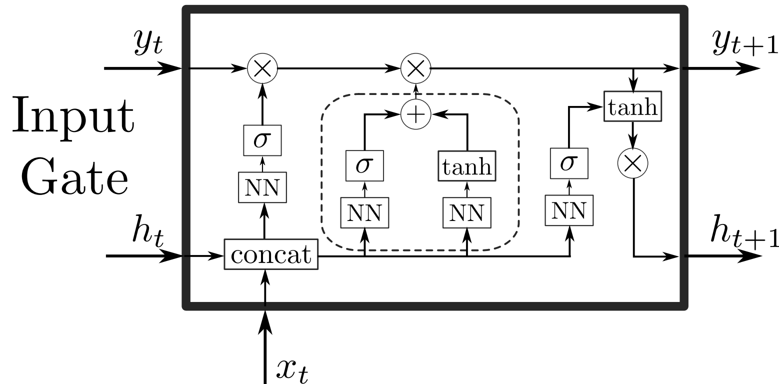
# Recurrent Neural Networks
## LSTM

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the problem of vanishing gradients and the difficulty of learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for updating cells, which allows them to selectively preserve information from previous time steps.



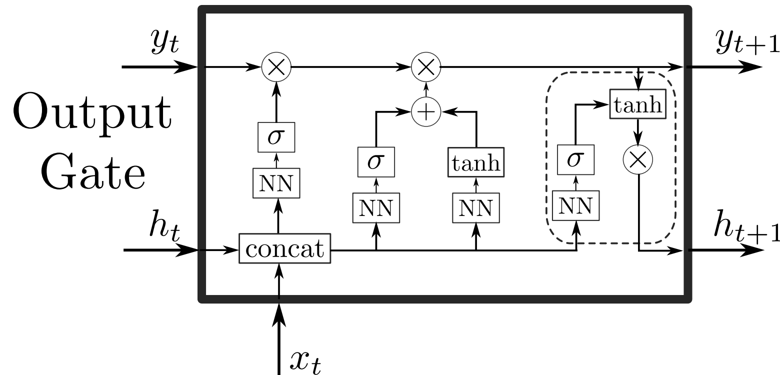
# Recurrent Neural Networks
## LSTM

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for updating the cells, which allows them to selectively preserve information from previous time steps.



# Recurrent Neural Networks
## LSTM

Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for updating the cells, which allows them to selectively preserve information from previous time steps.



# Recurrent Neural Networks

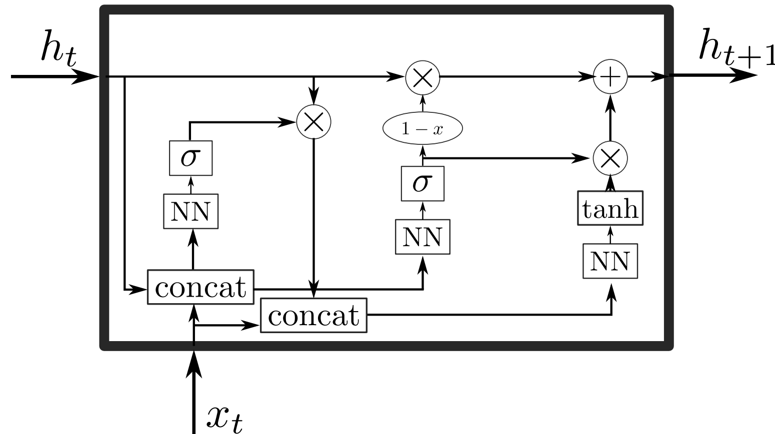
## Limits of LSTM

Limitations of LSTM RNNs:

- High computational cost: LSTMs are computationally more expensive compared to other traditional neural network models due to the presence of multiple gates and their sequential processing nature.
- Vanishing Gradient Problem: LSTMs, like any other RNNs, are prone to the vanishing gradient problem when the sequences are too long, making it difficult for the model to learn long-term dependencies.
- Overfitting: LSTMs are complex models and are more susceptible to overfitting compared to simple feedforward networks.
- Difficult to parallelize: Due to the sequential nature of LSTMs, they are difficult to parallelize and can take longer to train.
- Gradient Explosion: LSTMs can also suffer from the gradient explosion problem, where the gradients can become too large and cause numerical instability during training.

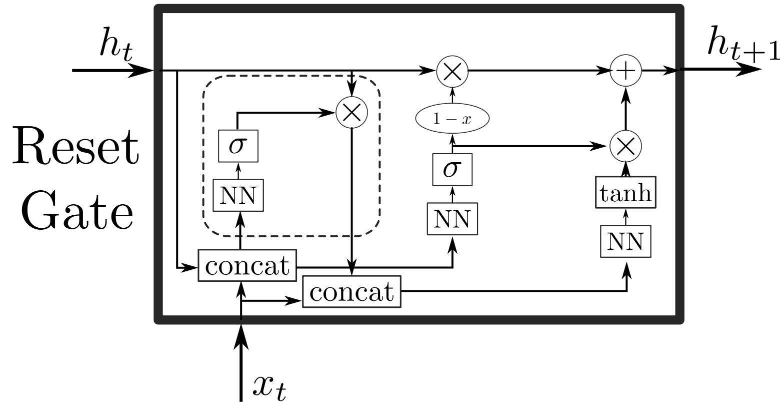
# Recurrent Neural Networks
## GRU

GRU blocks, or Gated Recurrent Units, are a type of recurrent neural network architecture that are similar to LSTMs in their function and ability to process sequential data. GRUs were introduced as a simplification of LSTMs, with the aim of reducing the number of parameters in the network and improving computational efficiency. GRUs achieve this by merging the forget and input gates in LSTMs into a single update gate, effectively combining the two operations in a single step.



# Recurrent Neural Networks
## GRU

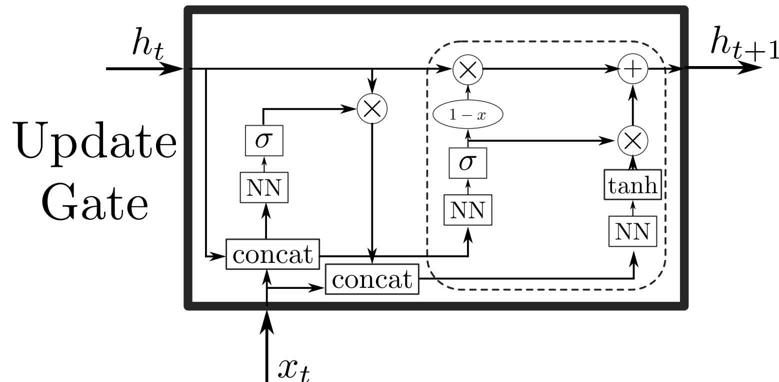
GRU blocks, or Gated Recurrent Units, are a type of recurrent neural network architecture that are similar to LSTMs in their function and ability to process sequential data. GRUs were introduced as a simplification of LSTMs, with the aim of reducing the number of parameters in the network and improving computational efficiency. GRUs achieve this by merging the forget and input gates in LSTMs into a single update gate, effectively combining the two operations in a single step.



# Recurrent Neural Networks

## GRU

GRU blocks, or Gated Recurrent Units, are a type of recurrent neural network architecture that are similar to LSTMs in their function and ability to process sequential data. GRUs were introduced as a simplification of LSTMs, with the aim of reducing the number of parameters in the network and improving computational efficiency. GRUs achieve this by merging the forget and input gates in LSTMs into a single update gate, effectively combining the two operations in a single step.



Limitations of GRU RNNs:

- Computational complexity: GRUs are more computationally efficient than LSTMs but still more complex than feedforward neural networks.
- Long-term dependencies: GRUs may struggle with capturing long-term dependencies in sequences, although they perform better in this regard than vanilla RNNs.
- Vanishing gradient problem: GRUs can still be affected by the vanishing gradient problem that plagues all RNN models. This problem makes it difficult for the model to learn from long sequences.
- Non-stationary data: GRUs may struggle with nonstationary data, where the statistical properties of the data change over time.

# Recurrent Neural Networks

## Application of RNNs

Applications of RNNs:

- Natural language processing (NLP): Using RNNs for text classification, language translation, and text generation.
- Time-series prediction: Using RNNs to make predictions based on sequential data, such as stock prices and weather patterns.
- Speech recognition: Using RNNs for speech-to-text conversion.

# Transformer and Attention Mechanism

Transformers and Attention Mechanisms are relatively recent developments in the field of deep learning, which have become popular for processing sequential data, such as natural language processing (NLP) tasks. Unlike Recurrent Neural Networks (RNNs) which process sequential data by repeatedly applying the same set of weights to the inputs over time, Transformers and Attention Mechanisms use self-attention mechanisms to dynamically weight the importance of different elements in the sequence. This enables Transformers to better capture the long-range dependencies between elements in the sequence, leading to improved performance on NLP tasks.

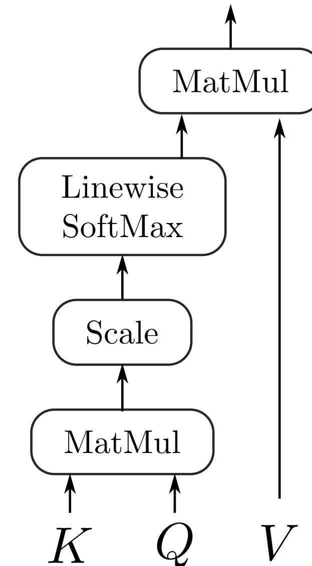
# Transformer and Attention Mechanism

## Self-Attention Mechanism

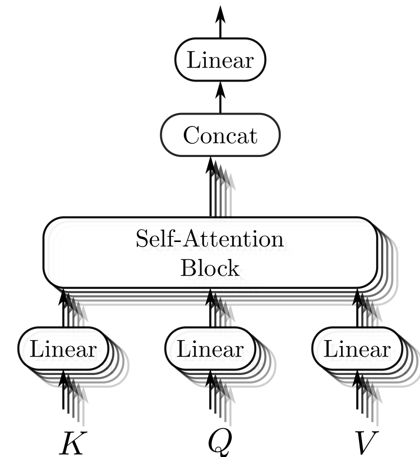
Self-attention mechanism in transformers is a method of calculating the weight of each input token in a sequence with respect to every other token in the same sequence, resulting in a representation of the input sequence in which the most relevant tokens have the highest weight. Mathematically, the self-attention mechanism can be represented as a dot product between the query ($Q$), key ($K$) and value ($V$) matrices, obtained from the input sequence, followed by a softmax activation to obtain the attention scores. These scores are then used to compute a weighted sum of the value matrix to produce the final representation.

$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad \text{where } Q \in \mathbb{R}^{m \times d_k},\ K \in \mathbb{R}^{n \times d_k},\ V \in \mathbb{R}^{n \times d_v}$$

The intuition behind these matrices is to provide a way to measure the similarity between each query and key, which is then used to determine the contribution of the values to the final result. The resulting weighted sum of the values represents the output of the self-attention mechanism, capturing the relationships between different parts of the input sequence.



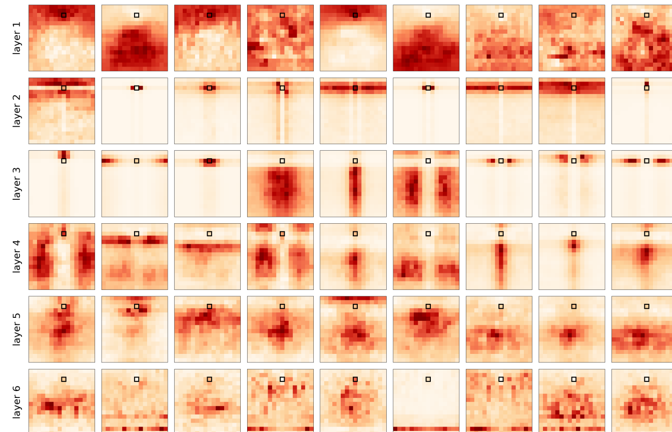
In Multi-head Attention, the self-attention mechanism is performed multiple times in parallel with different weight matrices, before being concatenated and once again projected, leading to a more robust representation of the input sequence. The intuition behind the three matrices ($Q$, $K$, $V$) remains the same as in self-attention, with $Q$ representing the query, $K$ the key and $V$ the value. Each head performs an attention mechanism on the input sequence, capturing different aspects and dependencies of the data, before being combined to form a more comprehensive representation of the input.



# Transformer and Attention Mechanism

## Visualizing Multi-Head Attention

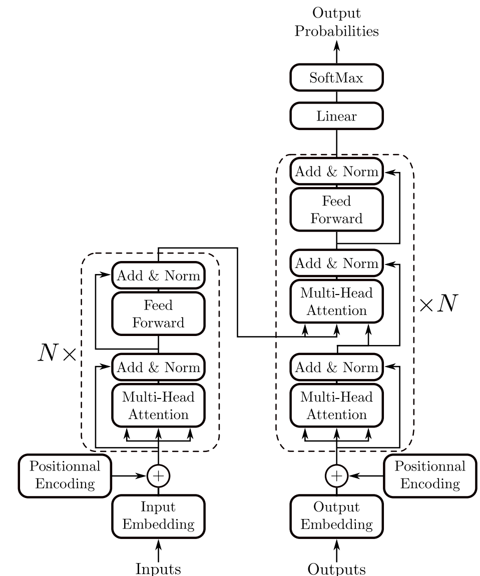
Visualizing Self-Attention for Image:
Link



Transformers are neural network models that use an encoder-decoder architecture. The encoder takes the input sequence and converts it into a continuous hidden representation, which is then passed to the decoder to generate the output sequence. The architecture of the transformer model is designed to allow the model to process the entire sequence in parallel, rather than processing one element at a time like in traditional RNNs.

Training of transformers involves optimizing a loss function that measures the difference between the model predictions and the true outputs. This loss function is usually based on the cross entropy between the predicted and true sequences.

The encoder-decoder mechanism is commonly referred to as the seq2seq mechanism.



# Part III

# Advanced Deep Learning Techniques: Reinforcement Learning, GANs and Beyond

Deep Q-Learning is a reinforcement learning algorithm that utilizes a neural network to approximate the optimal Q function, which is defined as the expected cumulative reward obtained by following a specific policy. The expected reward can be represented mathematically as follows:

$$R(\pi) = \sum_{t \leq T} E_{p^\pi}\left[\gamma^t r(s_t, a_t)\right] ,$$

where $r(s_t, a_t)$ is the reward at time step $t$, $\gamma$ is the discount factor, $T$ is the final time step and

$$p^\pi(a_0, a_1, s_1, ..., a_T, s_T) = p(a_0) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t) .$$

The Q function, $Q(s, a)$, represents the expected cumulative reward obtained by taking action $a$ in state $s$:

$$Q^\pi(s, a) = E_{p^\pi}\left[\sum_{t \leq T} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a\right] .$$

The policy, represented by $\pi(a|s)$, is a probability distribution over actions given a state. The optimal Q function, $Q^*(s, a)$, can be found by solving the Bellman equation:

$$Q^*(s, a) = \mathbb{E}[R|s, a] = \mathbb{E}[r + \gamma \max_{a'} Q^\pi(s', a')|s, a].$$

The loss in Deep Q-Learning method is the difference between the predicted Q-value and the target Q-value, which is the maximum expected reward obtained from the next state:

$$\mathcal{L}(\theta) = E_{s' \sim \pi^*(.|s,a)} \| r + \gamma \max_{a'} Q(s', a', \theta) - Q(s, a, \theta) \|^2.$$

This loss is used to update the parameters of the deep learning model in order to make the predictions more accurate. The target Q-value is typically computed as the reward obtained from taking an action in the current state, plus the maximum expected reward obtainable from the next state:

1. At the state $s_t$, select the action $a_t$ with best reward using $Q_t$ and store the results;
2. Obtain the new state $s_{t+1}$ from the environment $p$;
3. Store $(s_t, a_t, s_{t+1})$;
4. Obtain $Q_{t+1}$ by minimizing $\mathcal{L}$ from a batch recovered from previously stored results.

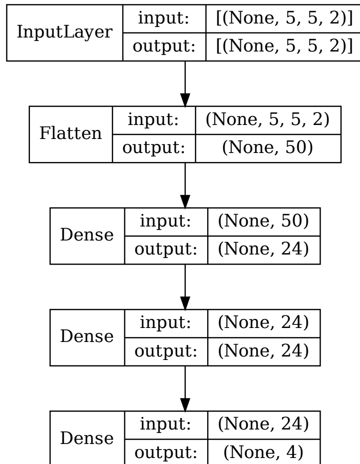
- A Mouse has to feed on food (red) and avoid poison (blue).
- It has a vision range of 2 squares. So it can see the 25 cells around.
- The reward for a cheese cell is 0.5, while the reward for eating poison is $-1$.

On this example, the mouse behaves randomly.

# Deep Reinforcement Learning

## Model 1

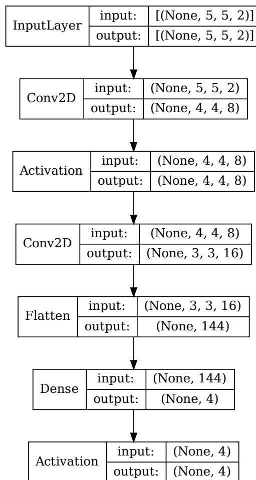
Fully connected network:



# Deep Reinforcement Learning

## Model 2

Convolutional network:



# Deep Reinforcement Learning

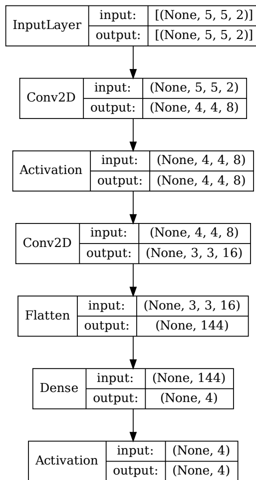
## $\epsilon$-greedy Algorithm

The $\epsilon$-greedy algorithm is a common exploration strategy used in Deep Q learning. Balances exploration, where the agent tries out new actions and collects new data, and exploitation, where the agent uses the information it already has to select the action with the highest expected reward. The algorithm selects a random action with probability $\epsilon$ and the action with the highest Q value with probability $1 - \epsilon$. The value of $\epsilon$ decreases over time to gradually shift the focus from exploration to exploitation.

# Deep Reinforcement Learning

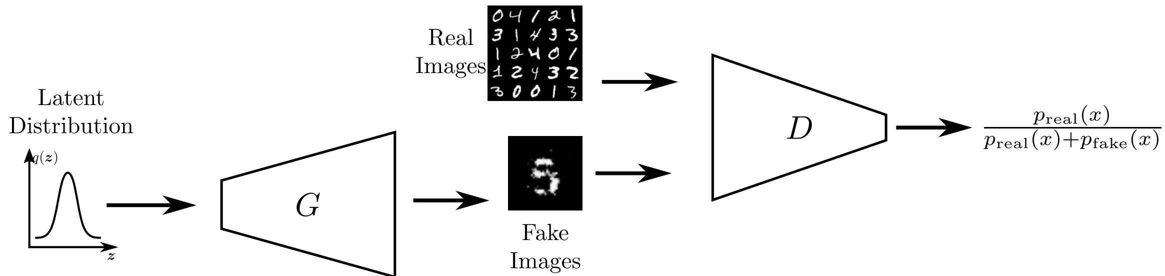
## Model 2+ Incorporated $\epsilon$-exploration

Convolutional network + $\epsilon$-greedy
Algorithm during training:



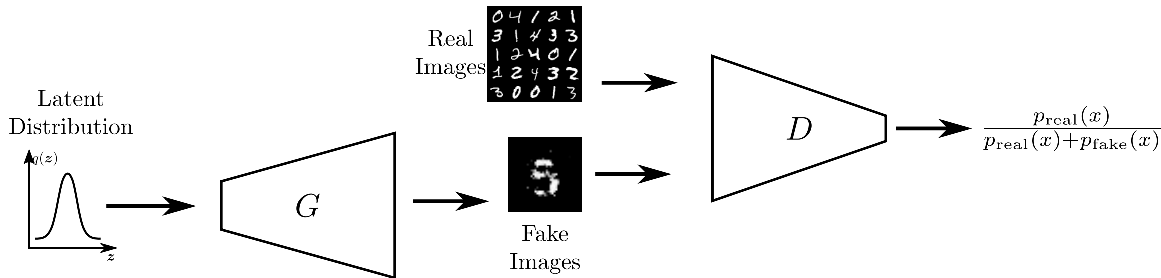
# Generative Adversarial Networks
## GANs Models

Generative Adversarial Networks (GANs) are a type of deep learning architecture composed of two neural networks, the generator and the discriminator, that are trained in a adversarial manner. The generator network is trained to generate fake data that appears similar to real data, while the discriminator network is trained to distinguish between real and fake data. The loss for GANs is defined as a min-max game, where the generator minimizes the loss function, and the discriminator maximizes it. $D$ and $G$ represent the discriminator and generator networks, respectively, and the goal is to find the optimal configuration for $D$ and $G$ such that the generated samples appear indistinguishable from real data.



The loss of a (GAN) is defined as a minimax game between the generator and discriminator models. The generator aims to generate samples that are indistinguishable from real samples, while the discriminator aims to distinguish the generated samples from real samples. The loss function for the generator is defined as $-\log(D(G(z)))$, where $D$ is the discriminator model and $G(z)$ is the generator's output for a random noise vector $z$. The loss function for the discriminator is defined as $\log(D(x)) + \log(1 - D(G(z)))$, where $x$ is a real sample.

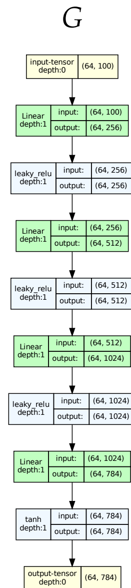
$$\min_D \max_G \mathbb{E}_{x\sim p_{\text{real}}} [\log(D(x))] + \mathbb{E}_{z\sim q} [\log(1 - D(G(z)))]$$



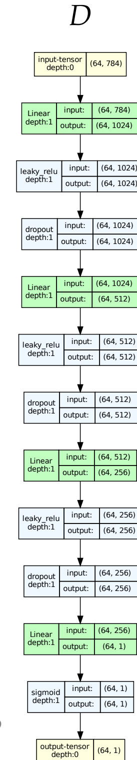
# Generative Adversarial Networks

## MNIST Generation

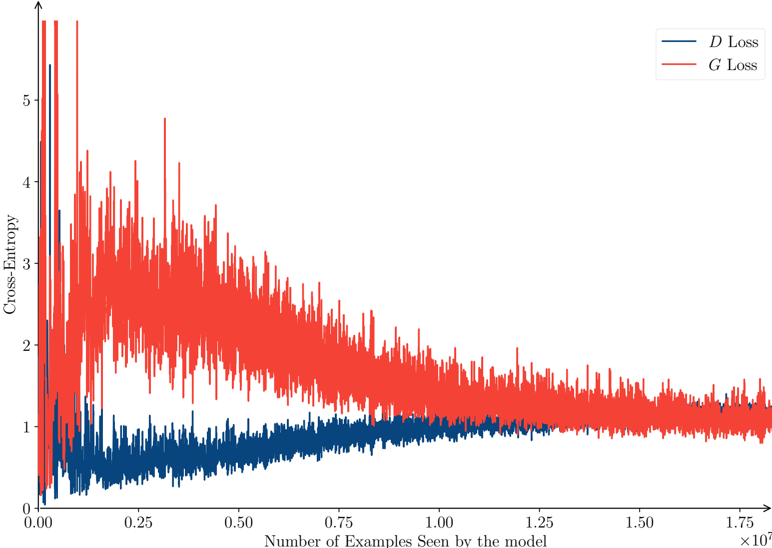
$G$




$D$



# Generative Adversarial Networks
## MNIST Generation



# Generative Adversarial Networks

## MNIST Generation

# Sentiment Analysis

## BERT

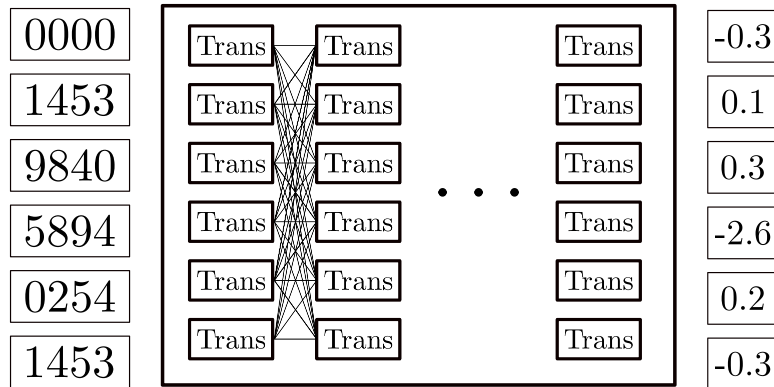
BERT (Bidirectional Encoder Representations from Transformers) is a pre-trained language model developed by Google in 2018. It is a transformer-based architecture that uses a masked language modeling task to generate a deep understanding of the contextual relationships between words in a sentence. BERT can be fine-tuned for various NLP tasks such as sentiment classification by adding a classification layer on top of its pre-trained representations. The model has achieved state-of-the-art performance in a wide range of NLP tasks, making it a popular choice for sentiment analysis.

A bidirectional transformer is a type of transformer architecture in natural language processing (NLP) where information from both past and future contexts is taken into consideration when making predictions. This is achieved by processing the input sequence in two directions, starting from the beginning and the end of the sequence, and concatenating the outputs to obtain the final representation. This allows the model to capture the context both in the forward and backward directions, providing a more comprehensive representation of the input sequence.

# Sentiment Analysis

## Bert

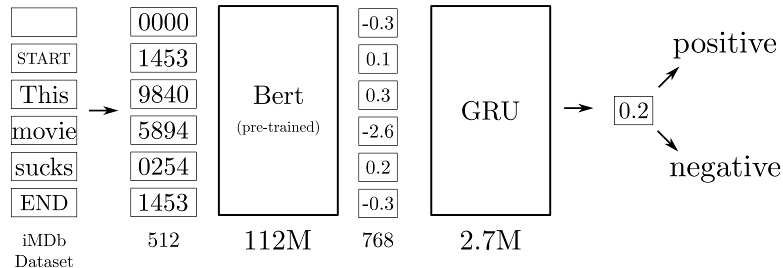
BERT model consists of multiple transformer encoder blocks, with a self-attention mechanism, a feedforward neural network and layer normalization, stacked on top of each other. It also includes a positional encoding component to capture the relative position of tokens in a sequence, and a segment encoding component to differentiate between different sequences within the same input.



Bert is trained on two tasks, the masked language model and the next sentence prediction. In the masked language model task, a portion of the input sequence is masked and the model must predict the original token based on its context. In the next sentence prediction task, the model receives a pair of sentences and must predict whether the second sentence follows the first one in the context of the input text. Both of these tasks are used to train Bert to understand the context of words in a sentence and how they relate to each other, allowing it to perform well on a wide range of natural language processing tasks.

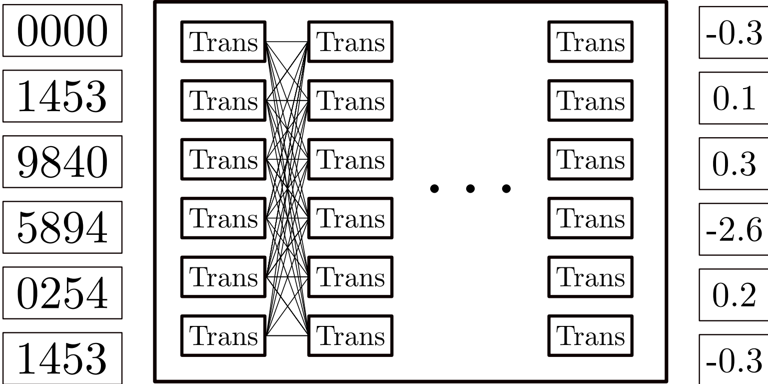
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Sentence | There | is | no | curse | in | Elvish, | Entish, | or | the | tongues | of | Men | for | this | treachery. |
| Basic-level masking | There | is | | curse | in | Elvish, | | or | the | tongues | of | Men | for | this | |
| Entity-level masking | There | is | | | in | Elvish, | Entish, | or | the | | | | for | this | treachery. |
| Phrase-level masking | | | | curse | in | Elvish, | Entish, | or | | | | | for | this | treachery. |

Bert can be fine-tuned for sentiment analysis by adding a classifier layer on top of the pretrained Bert model. The layer is trained on a labeled sentiment analysis dataset to predict the sentiment of a given input sequence, which can be a sentence, paragraph, or document. Fine-tuning the model allows it to learn the specific nuances of sentiment in the target task and produce improved results compared to using the pre-trained model alone.



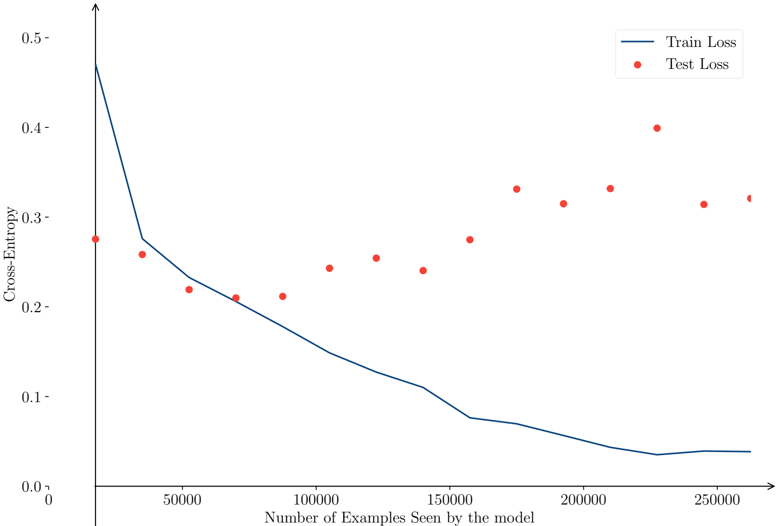
# Sentiment Analysis

## Sentiment Analysis

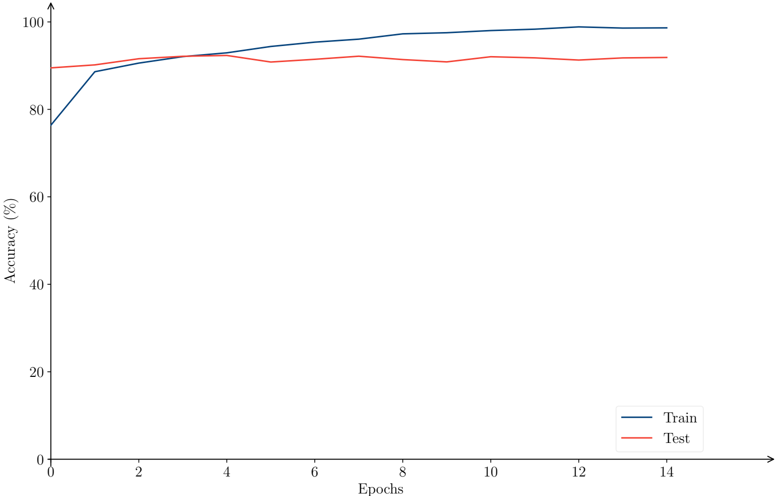


# Sentiment Analysis

## Sentiment Analysis



# Sentiment Analysis
## Sentiment Analysis



# Sentiment Analysis

## Sentiment Analysis

```
In [117]: predict_sentiment(model, tokenizer, "This film is terrible")
Out[117]: 0.028073227033019066

In [118]: predict_sentiment(model, tokenizer, "This film is great")
Out[118]: 0.9749133586883545

In [119]: predict_sentiment(model, tokenizer, "This lecture was quite boring")
Out[119]: 0.0537508986890316

In [120]: predict_sentiment(model, tokenizer, "This lecture was challenging")
Out[120]: 0.6184227466583252

In [121]: predict_sentiment(model, tokenizer, "This lecture was dense but interesting")
Out[121]: 0.7105910181999207
```

# Sentiment Analysis

## Note on BERT and ChatGPT

ChatGPT and BERT are both large language models developed by OpenAI and Google, respectively, but they differ in their architecture and application. ChatGPT is a generative language model based on the Transformer architecture and is trained to predict the next word given a large corpus of text. BERT, on the other hand, is a pre-trained language representation model designed for natural language processing tasks such as sentiment classification, named entity recognition, and question answering. BERT uses a transformer architecture with a masked language modeling objective, training the model to predict the masked words in a sentence given the surrounding context. In summary, while ChatGPT is focused on generating text, BERT is focused on understanding and manipulating text.

Part IV

# Practical Lessons

# Build and use an autoencoder

## Formal introduction of an autoencoder

### Definition

An autoencoder is a type of artificial neural network used to learn efficient codings of unlabeled data. It consists of two main components:

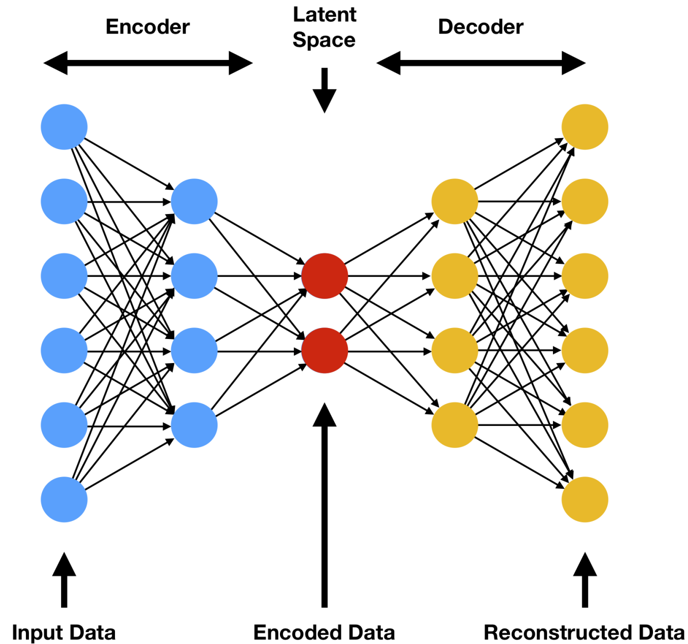
- An encoder function: $encoder(x) : \mathbb{R}^d \to \mathbb{R}^m$
  Maps an input $x$ from the input space $\mathbb{R}^d$ to a hidden representation space $\mathbb{R}^m$.
- A decoder function: $decoder(z) : \mathbb{R}^m \to \mathbb{R}^d$
  Maps the hidden representation $z$ back to the original input space $\mathbb{R}^d$.

### Goal

The primary goal of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction or feature learning. Through training, the autoencoder learns to compress the data from $\mathbb{R}^d$ to $\mathbb{R}^m$ (where $m < d$) and then reconstruct the data back to $\mathbb{R}^d$ as accurately as possible. This process forces the autoencoder to capture the most important features of the data in the hidden representation $z$.

# Build and use an autoencoder

## Formal introduction of an autoencoder



## Unsupervised Learning

Autoencoders are a classic example of unsupervised learning. In unsupervised learning, the goal is to learn patterns from unlabelled data. Autoencoders learn to compress and decompress the input data without any explicit labels, aiming to capture the underlying structure of the data.

## Objective Function

The learning process of an autoencoder is guided by the minimization of a loss function, typically involving a norm that measures the difference between the input and the reconstructed output. Formally, the objective is to minimize:

$$\min_{\theta} \ \mathbb{E}_{x \sim P} \left[ l(x, \text{decoder}_{\theta}(\text{encoder}_{\theta}(x))) \right]$$

where $x$ is the input data, $\theta$ represents the parameters of the encoder and decoder, and $l$ is a loss function.

## Unsupervised Learning

Autoencoders are a classic example of unsupervised learning. In unsupervised learning, the goal is to learn patterns from unlabelled data. Autoencoders learn to compress and decompress the input data without any explicit labels, aiming to capture the underlying structure of the data.

## Objective Function

The learning process of an autoencoder is guided by the minimization of a loss function, typically involving a norm that measures the difference between the input and the reconstructed output. Formally, the objective is to minimize:

$$\min_{\theta} \mathbb{E}_{x \sim P} \left[ \|x - \hat{x}\|_2^2 \right]$$

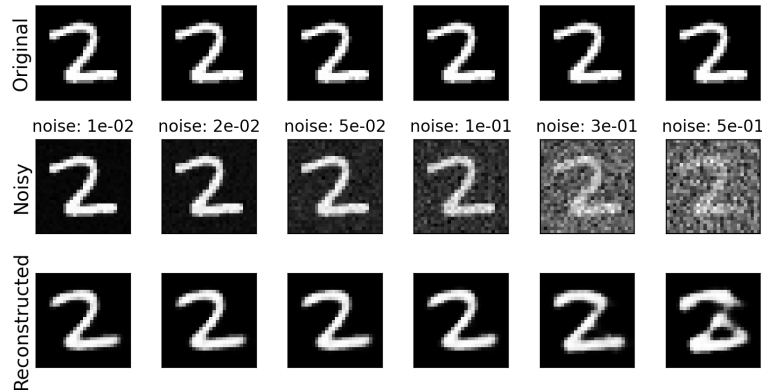
where $x$ is the input data, $\theta$ represents the parameters of the encoder and decoder and $\hat{x} = \text{decoder}_\theta(\text{encoder}_\theta(x))$ in the reconstruction.

## Applications of Autoencoders

- **Reduce the size of the data to transfer:** Autoencoders can compress data into a lower-dimensional space, facilitating faster data transfer by reducing the amount of data that needs to be transmitted.

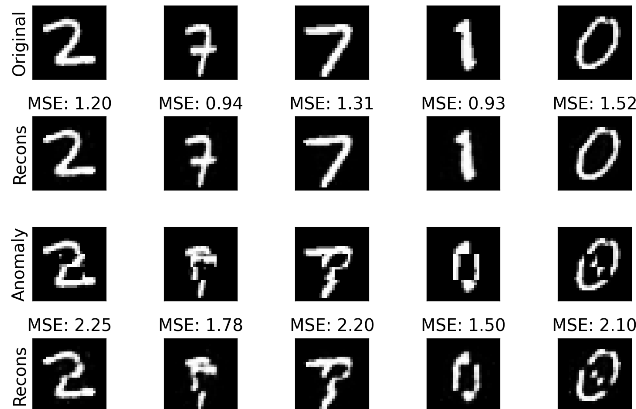
# Applications of Autoencoders

- **Reduce the size of the data to transfer:** Autoencoders can compress data into a lower-dimensional space, facilitating faster data transfer by reducing the amount of data that needs to be transmitted.
- **Denoise image:** By learning to ignore the "noise" in the input data, autoencoders can reconstruct cleaner versions of noisy images, effectively removing the noise.



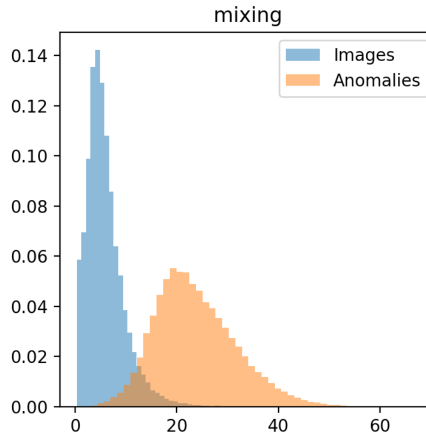
# Applications of Autoencoders

- **Reduce the size of the data to transfer:** Autoencoders can compress data into a lower-dimensional space, facilitating faster data transfer by reducing the amount of data that needs to be transmitted.
- **Denoise image:** By learning to ignore the "noise" in the input data, autoencoders can reconstruct cleaner versions of noisy images, effectively removing the noise.
- **Anomaly detection:** Autoencoders can learn the normal patterns within data. Deviations from these patterns, when the reconstruction error is high, can indicate anomalies or outliers in the data.



# Applications of Autoencoders

- **Reduce the size of the data to transfer:** Autoencoders can compress data into a lower-dimensional space, facilitating faster data transfer by reducing the amount of data that needs to be transmitted.
- **Denoise image:** By learning to ignore the "noise" in the input data, autoencoders can reconstruct cleaner versions of noisy images, effectively removing the noise.
- **Anomaly detection:** Autoencoders can learn the normal patterns within data. Deviations from these patterns, when the reconstruction error is high, can indicate anomalies or outliers in the data.



# Build and use an autoencoder

## Your turn !

Get the link of the notebook : Here. Or at:

- `https://www.alexverine.com`
- Teaching
- Introduction to Deep Learning
- Lien Notebooks Python