DEEP LEARNING 2

FROM THEORY TO PRACTICE

Alexandre Vérine,
Research Fellow, École Normale Supérieure Paris

Double Licence Intelligence Artificielle et Sciences des Organisations
3e année de Licence
Université Paris-Dauphine, PSL

September 8, 2025

Semester Schedule (Temporary)

- 09/09: Fundamentals of Deep Learning + In a Deep Learning Model
- 16/09: **No Class**
- 23/09: TP1 Classification - *Introduction to PyTorch*
- 30/09: In a Deep Learning Model + Techniques to Improve Deep Learning Training + Advanced Deep Learning Techniques
- 07/10: TP2 Autoencoders - *Hyperparameter Tuning*
- 13/10: TP3 Image Segmentation - *From CPU to GPU and Parallelization* (**It is a Monday**)
- 21/10: **Graded Individual Practical Work**

Semester Schedule (Temporary)

- 28/10: **No Class**
- 04/11: TP4 Deep Reinforcement Learning - *From Notebook to Script* - Part 1
- 11/11: **No Class - Armistice Day**
- 18/11: TP4 Deep Reinforcement Learning - *From Notebook to Script* - Part 2
- 17/11: TP5 Adversarial Attacks - *Importance of Git* - Part 1
- 25/11: TP5 Adversarial Attacks - *Importance of Git* - Part 2
- 02/12: Project Presentation - Group Formation
- 16/12: **Project Presentation**

AI 101: From Fundamentals to Deep Learning

In a Deep Learning Model : From Neural Networks to Transformer Models

Techniques to Improve Deep Learning Training

Deep Learning and Applications

Managing Deep Learning Projects

Project Presentation

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, ...

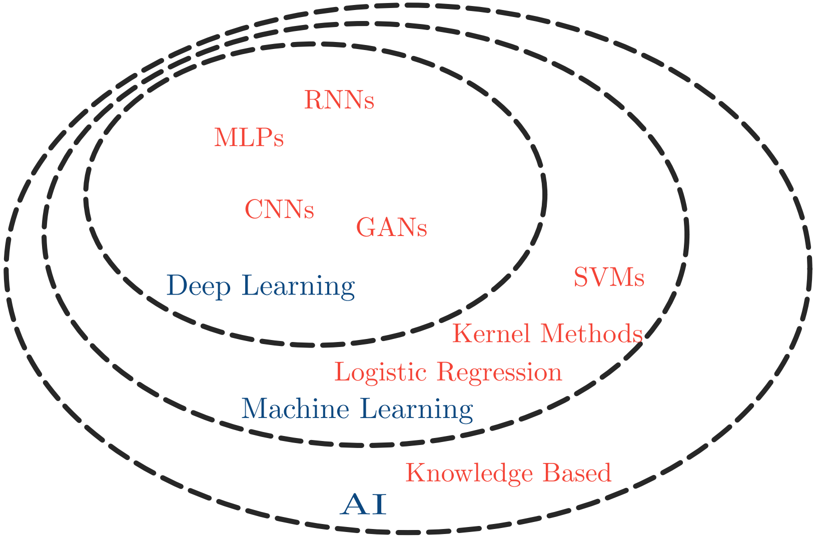


Figure. Subsets of Artificial Intelligence

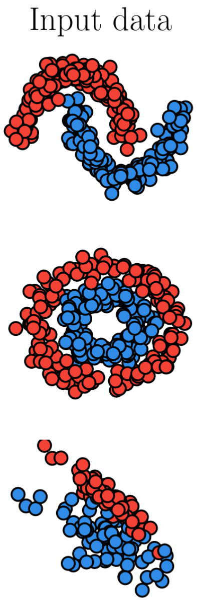
Introduction to Artificial Intelligence

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

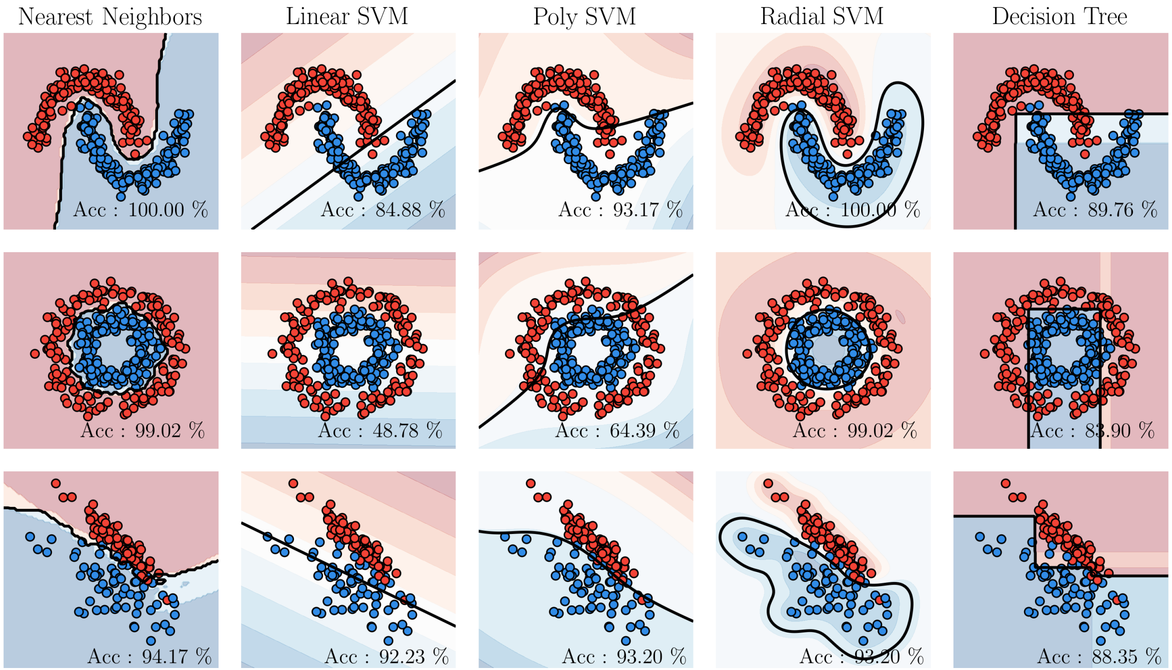


Figure. 2D classification for different AI models.

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

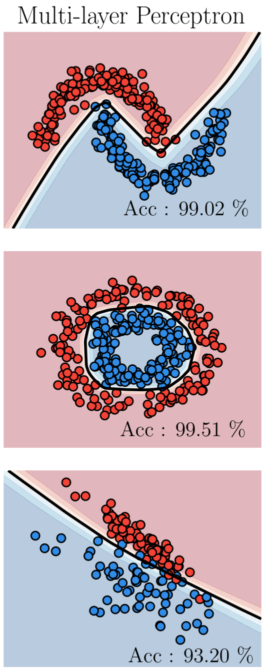


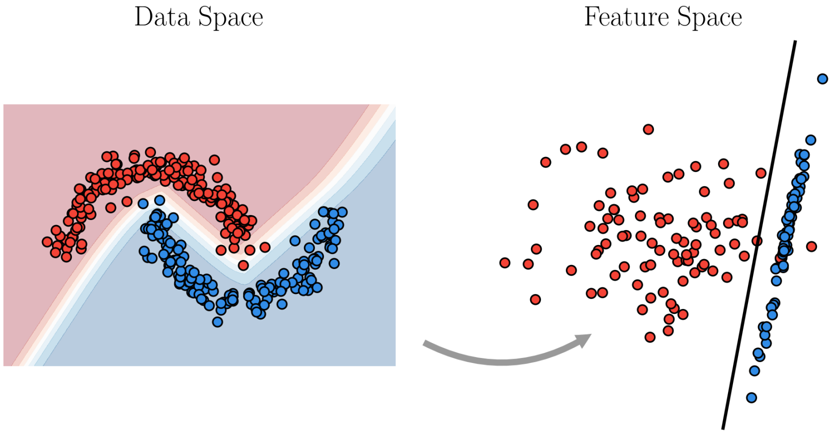
Figure. 2D classification for small Neural Network.

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.

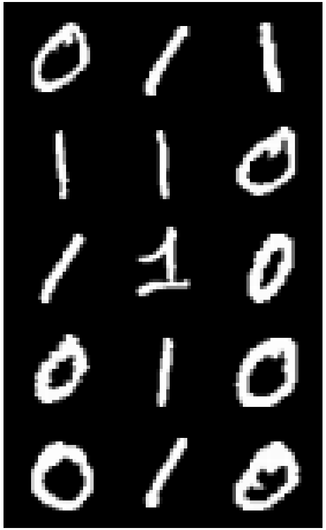


Figure. MNIST

Introduction to Artificial Intelligence

Example of Representation Learning

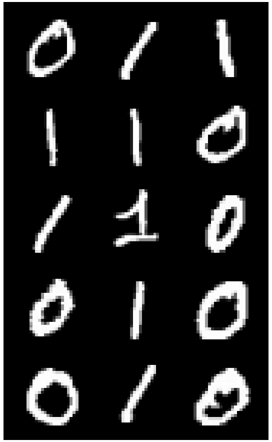


Figure. MNIST : Layer 0

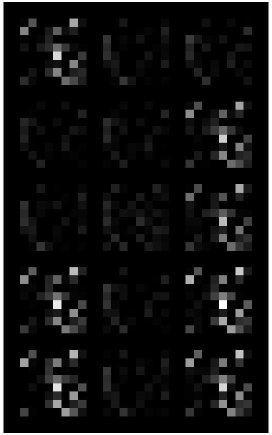


Figure. MNIST : Layer 1

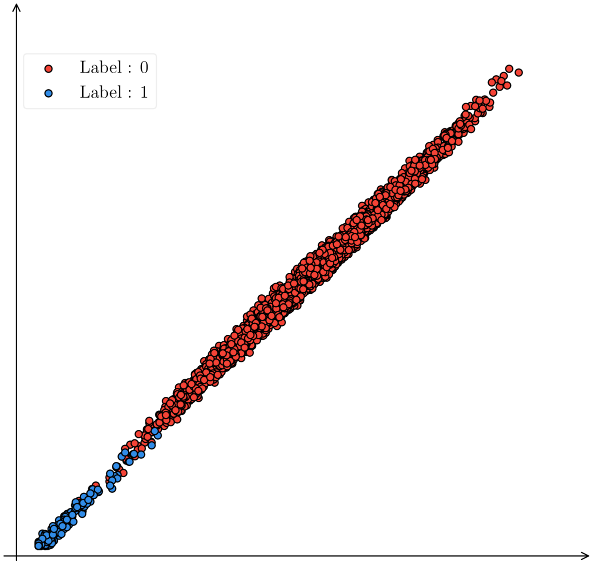


Figure. MNIST : Layer 2

Introduction to Artificial Intelligence

Example of Representation Learning

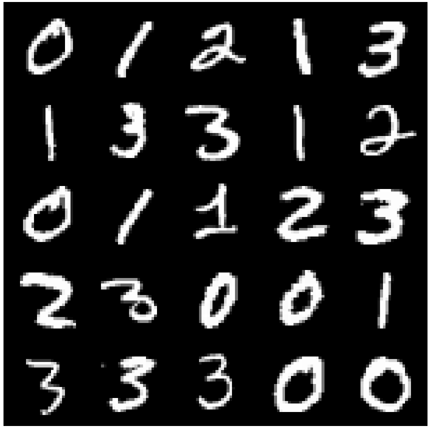


Figure. MNIST : Layer 0

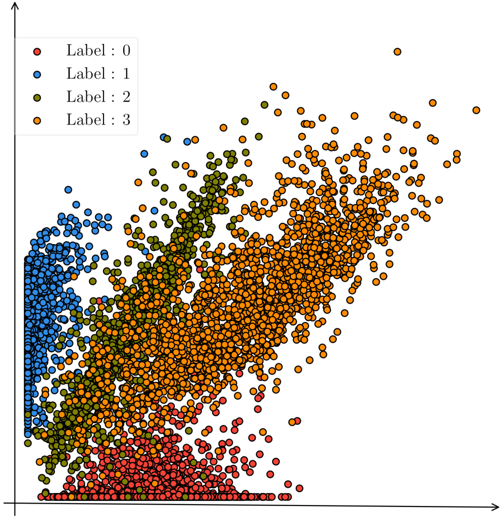


Figure. MNIST : Layer 2

Introduction to Artificial Intelligence

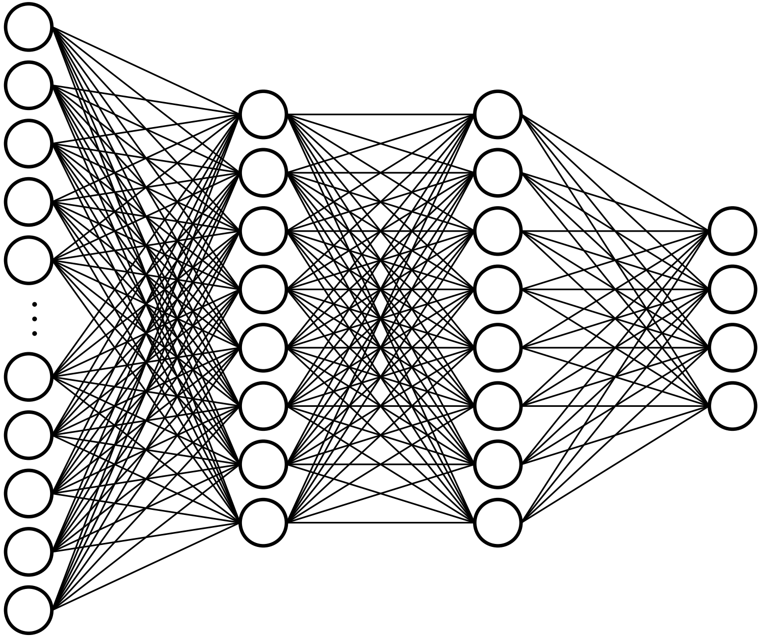
Deep Learning and Neural Networks

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.

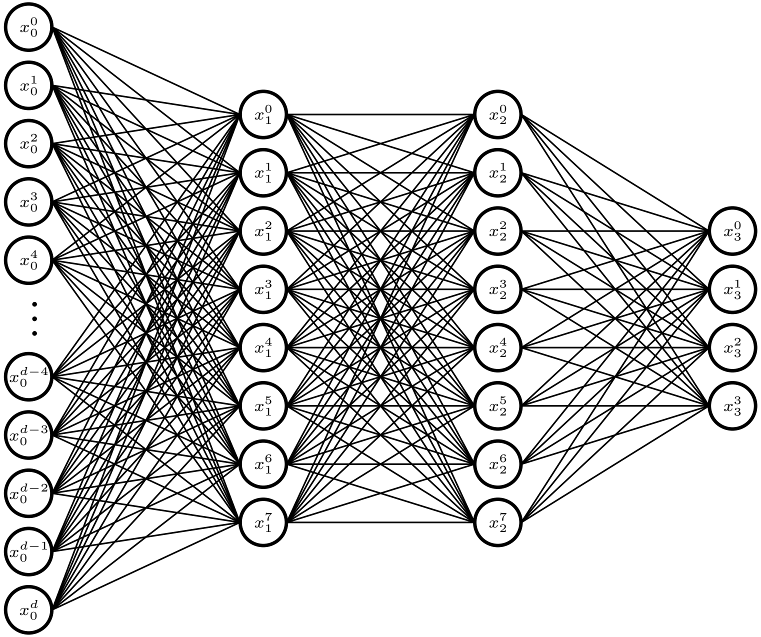


Let's see what the interconnected nodes, the layers and the transformations are.

Neural Networks Fundamentals

Neurons

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

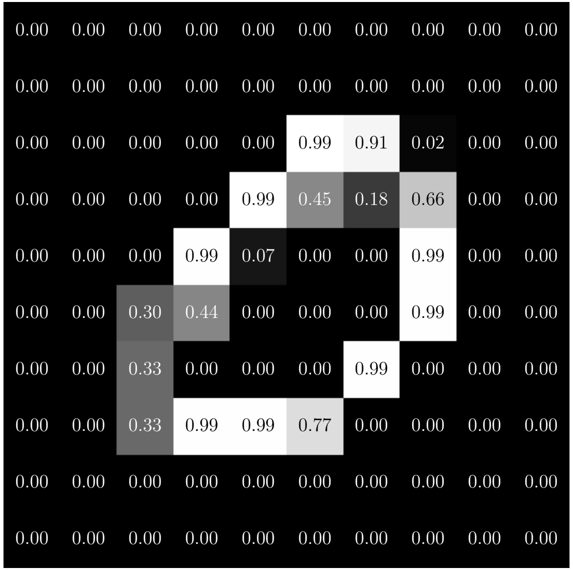


A Neuron is a processing unit that receives input, performs a computation, and produces an output. Here, the inputs are x_{i-1} and the output is x_i^k.

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

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

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

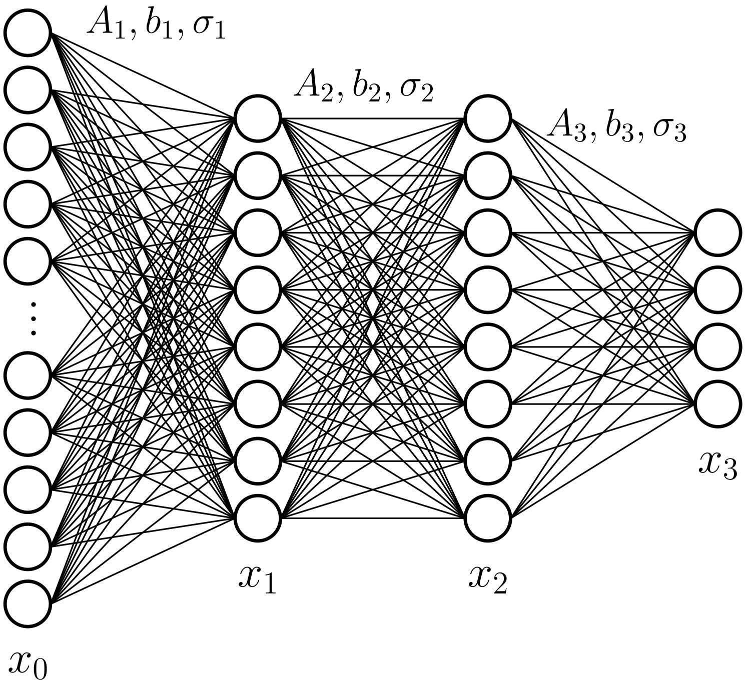
Neural Networks Fundamentals

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.



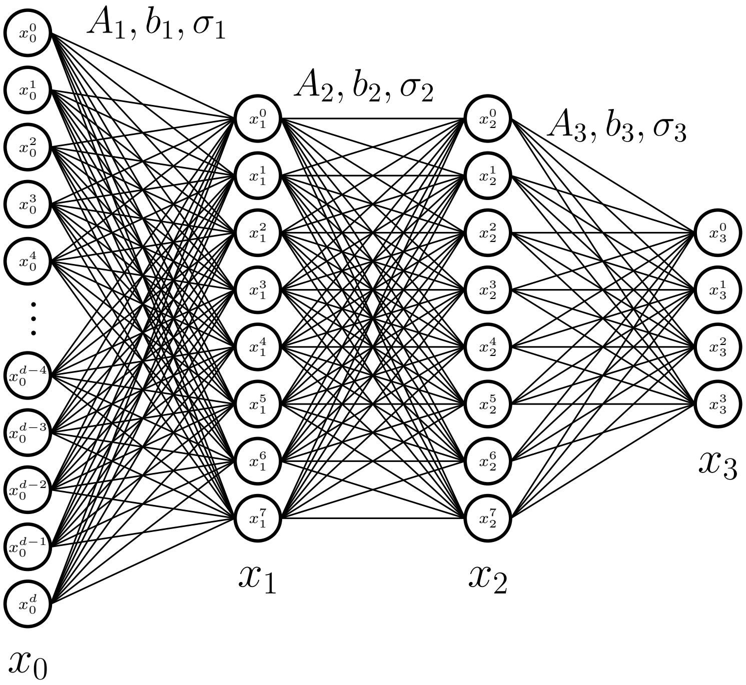
Neural Networks Fundamentals

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

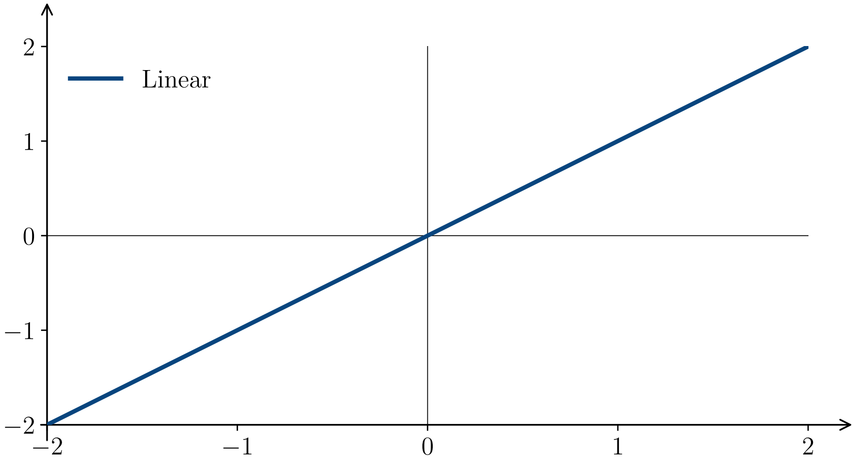
Neural Networks Fundamentals

Linear

- Linear activation Function:

$$\sigma(x) = x$$

- Final activation
- Use case : Regression



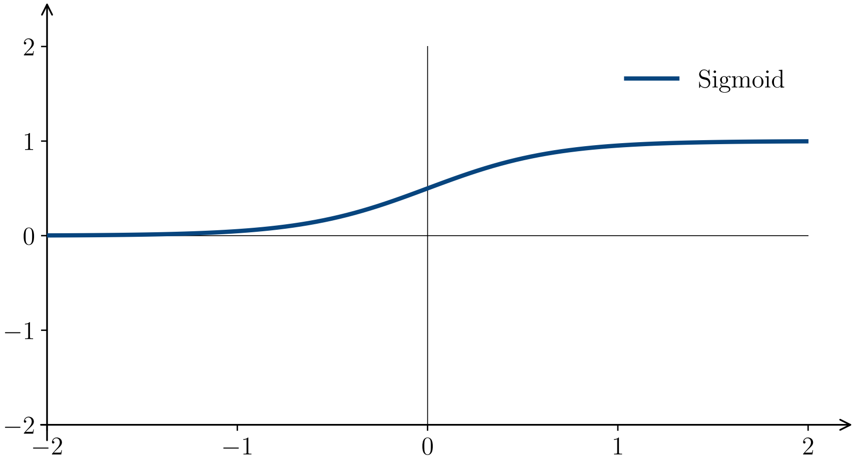
Neural Networks Fundamentals

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

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

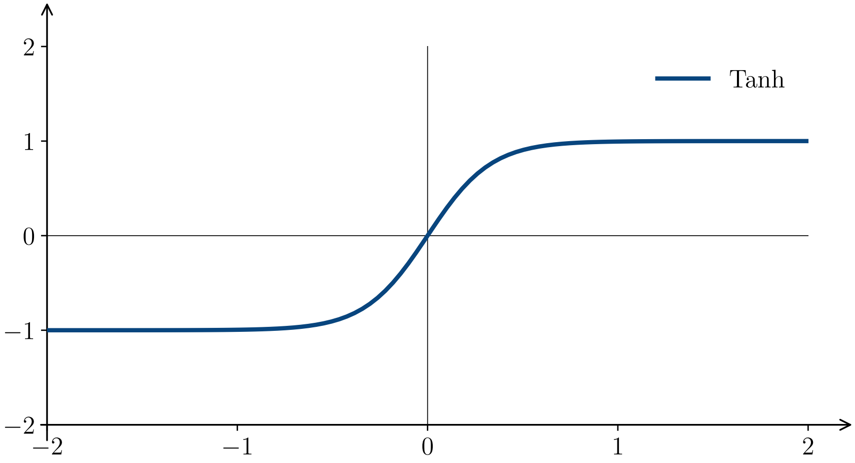
Neural Networks Fundamentals

Hyperbolic Tangent

- Hyperbolic Tangent

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

- Final activation
- Use case : Generative task



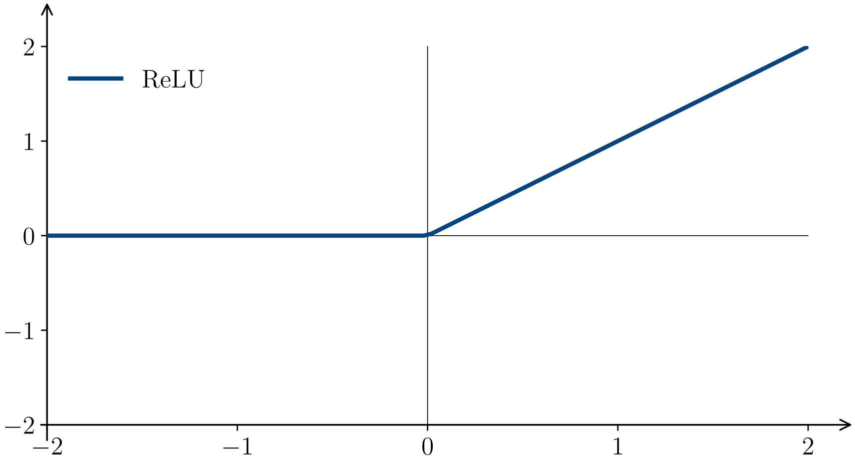
Neural Networks Fundamentals

ReLU

- Rectified Linear Unit (ReLU):

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

- Intermediate activation



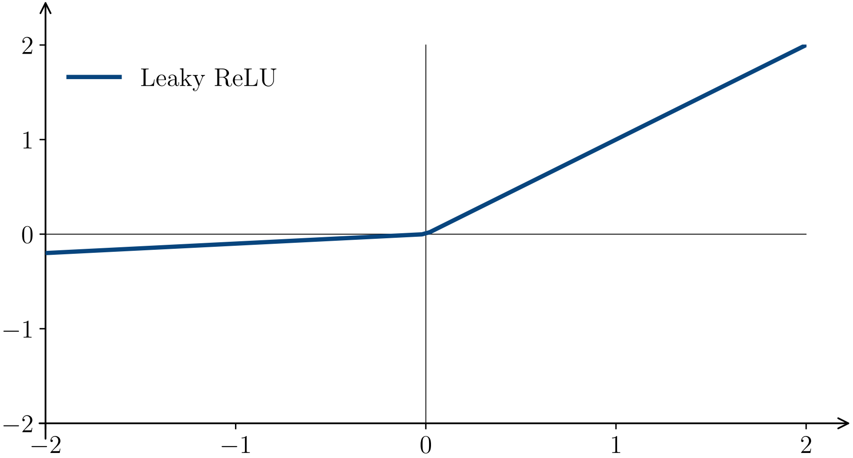
Neural Networks Fundamentals

Leaky ReLU

- Leaky Rectified Linear Unit (Leaky ReLU):

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

- Intermediate activation



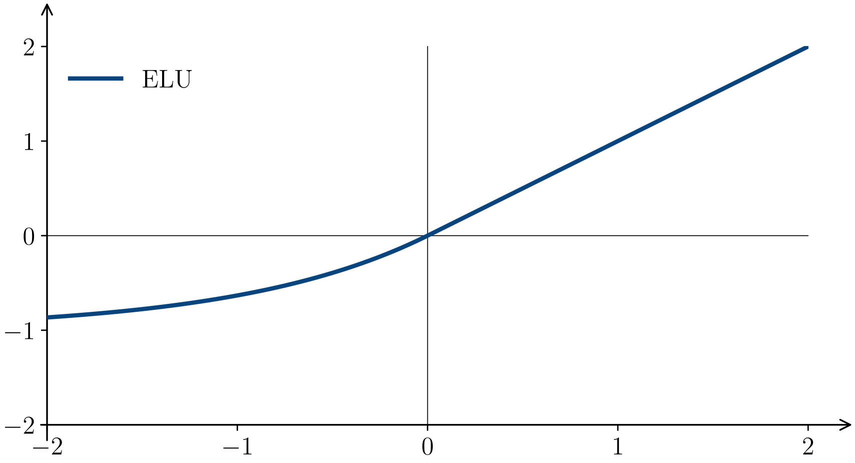
Neural Networks Fundamentals

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

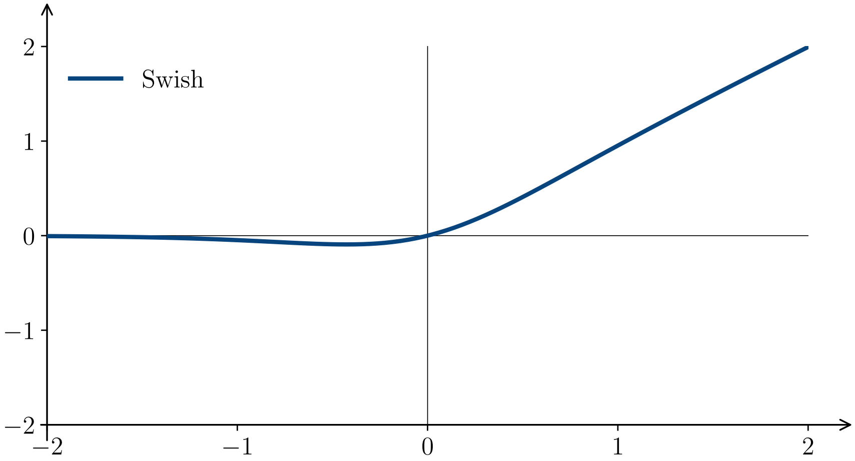


SWISH

- Sigmoid-Weighted Linear Unit (Swish):

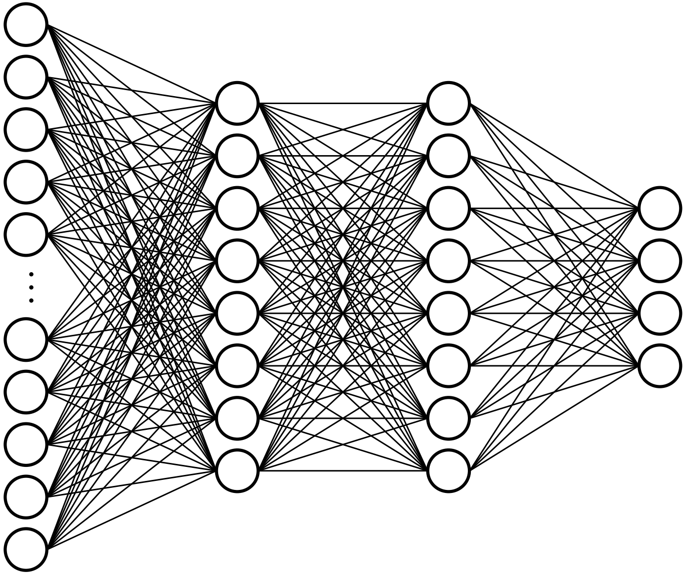
$$\sigma(x) = \frac{x}{1 + e^{-x}}$$

- 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.

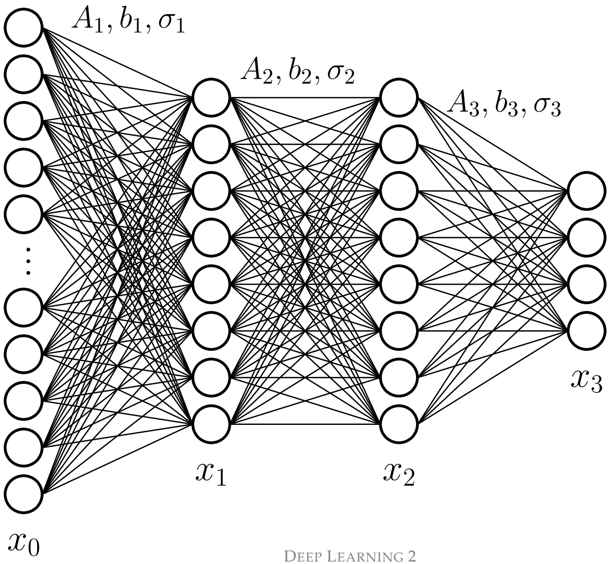


The Multi-layer Perceptron (MLP)

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 $\lambda$
Initialize parameters $\theta$
while $\theta$ has not converged do
    for $i = 1$ to $N$ do
        Randomly select $x_i$ from the dataset
        Compute gradient of the loss with respect to $\theta$: $\nabla_\theta l(x_i, f_\theta(x_i))$
        Update parameters $\theta = \theta - \lambda \nabla_\theta l(x_i, f(x_i))$
    end for
end while
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 $b$
Initialize parameters $\theta$
Initialize the number of batches $B = \left\lfloor \frac{N}{b} \right\rfloor$
while $\theta$ has not converged do
    for $i = 1$ to $B$ do
        Randomly select a mini-batch of $b$ samples from the dataset
        Compute gradient of the loss with respect to $\theta$: $\frac{1}{B}\sum_{i=1}^{B} \nabla_\theta l(x_i, f_\theta(x_i))$
        Update parameters $\theta = \theta - \lambda \frac{1}{B}\sum_{i=1}^{B} \nabla_\theta l(x_i, f(x_i))$
    end for
end while
return $\theta$
```

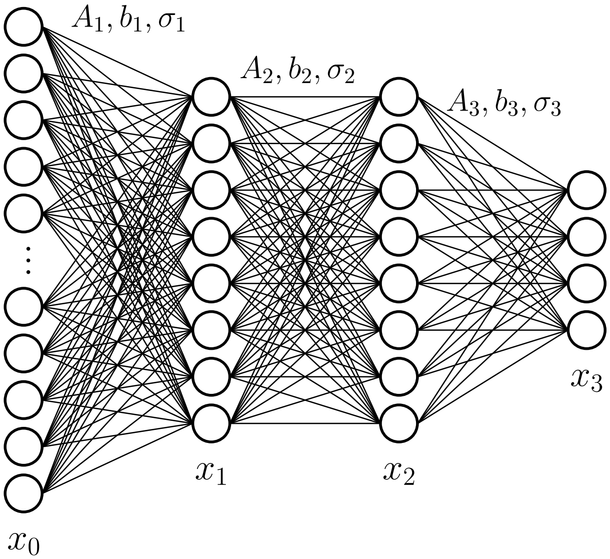
The Multi-layer Perceptron (MLP)

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:

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

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:

$$\begin{aligned}\frac{\partial l}{\partial x_2} &= \frac{\partial l}{\partial x_3}\frac{\partial x_3}{\partial x_2} = l'(x_3)\frac{\partial \sigma_3\left(A_3x_2+b_3\right)}{\partial x_2} = l'(x_3)\frac{\partial\left[A_3x_2+b_3\right]}{\partial x_2}\sigma_3'\left(A_3x_2+b_3\right)\\ &= l'(x_3)\,A_3^T\sigma_3'\left(A_3x_2+b_3\right)\end{aligned}$$

Therefore:

$$A_2 \leftarrow A_2 - \lambda\left[l'(x_3)A_3^T\sigma_3'\left(A_3x_2+b_3\right)\times\sigma_2'\left(A_2x_1+b_2\right)x_1^T\right]$$

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:

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

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:

$$\begin{aligned}
\frac{\partial l}{\partial b_3} &= \frac{\partial l}{\partial x_3}\frac{\partial x_3}{\partial b_3} \\
&= l'(x_3)\frac{\partial \sigma_3\left(A_3 x_2 + b_3\right)}{\partial b_3} \\
&= l'(x_3)\sigma_3'\left(A_3 x_2 + b_3\right)\frac{\partial\left[A_3 x_2 + b_3\right]}{\partial b_3} \\
&= \underbrace{l'(x_3)}_{\in\mathbb{R}}\underbrace{\sigma_3'\left(A_3 x_2 + b_3\right)}_{\in\mathbb{R}^{k_i\times 1}}
\end{aligned}$$

And thus :

$$b_3 \leftarrow b_3 - \lambda l'(x_3)\sigma'\left(A_3 x_2 + b_3\right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Let's move on the second layer:

$$\begin{aligned}\frac{\partial l}{\partial b_2} &= \frac{\partial l}{\partial x_2}\frac{\partial x_2}{\partial b_2}\\ &= \frac{\partial l}{\partial x_2}\frac{\partial \sigma_2\left(A_2x_1+b_2\right)}{\partial b_2}\\ &= \frac{\partial l}{\partial x_2}\sigma_2'\left(A_2x_1+b_2\right)\end{aligned}$$

And thus :

$$b_2 \leftarrow b_2 - \lambda\frac{\partial l}{\partial x_2}\sigma'\left(A_2x_1+b_2\right)$$

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

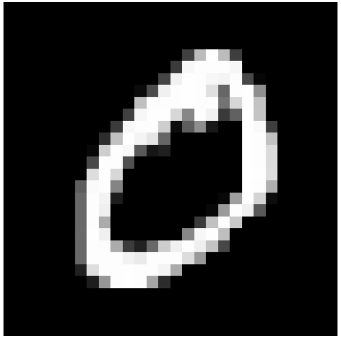
To update the weights, we need to compute the gradient of the loss with respect to the output of the network, and then **back-propagate** the gradient of the loss with respect to each activation, the $\frac{\partial l}{\partial x_i}$, through the network to compute the gradients with respect to the weights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

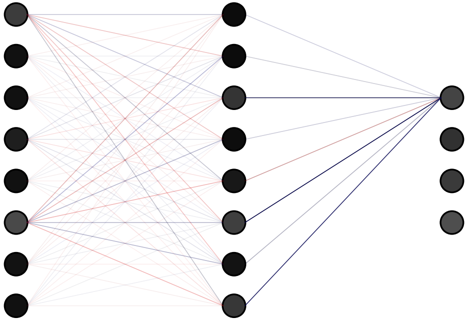
The Multi-layer Perceptron (MLP)

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.



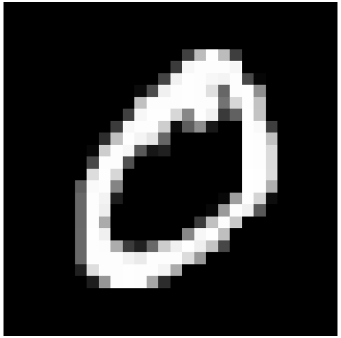


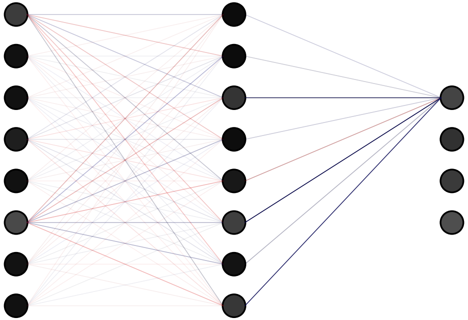
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



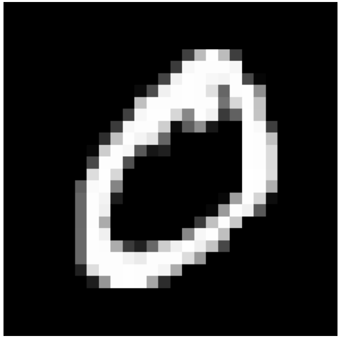


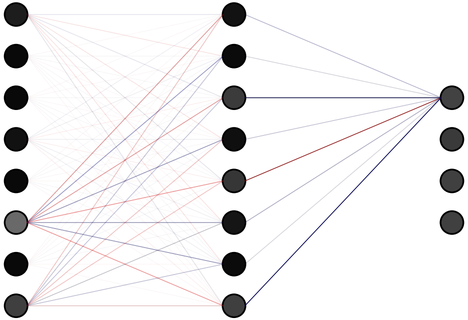
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



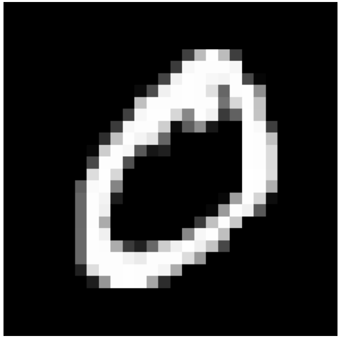


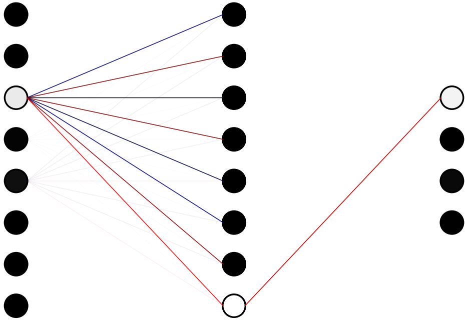
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



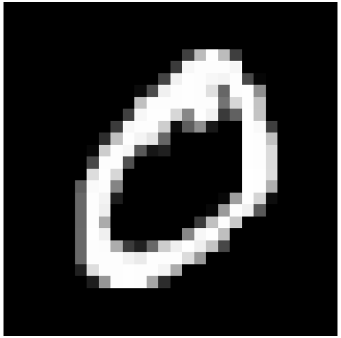


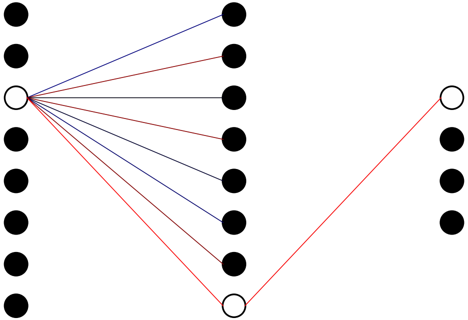
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.





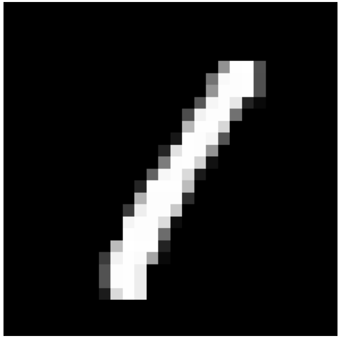
$$x_3^1 = \sigma_3 \left(A_3^{1,1} x_2^1 + A_3^{1,2} x_2^2 + \cdots + A_3^{1,8} x_2^8 \right)$$

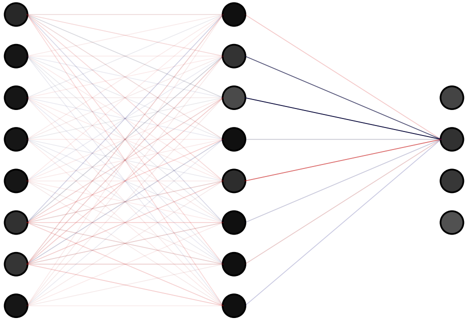
The Multi-layer Perceptron (MLP)

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.



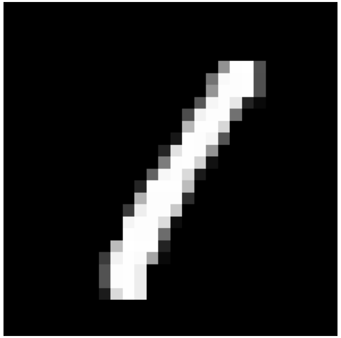


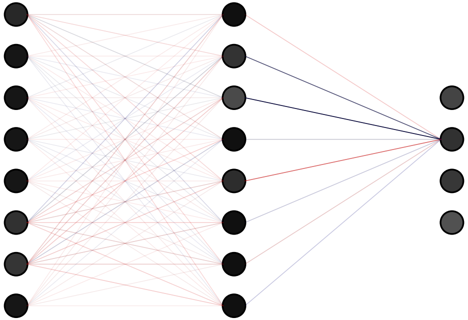
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



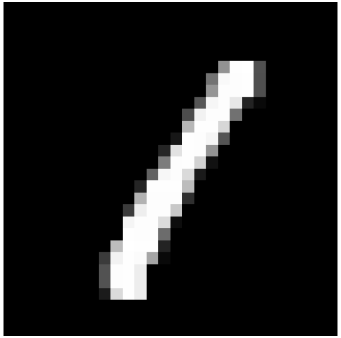


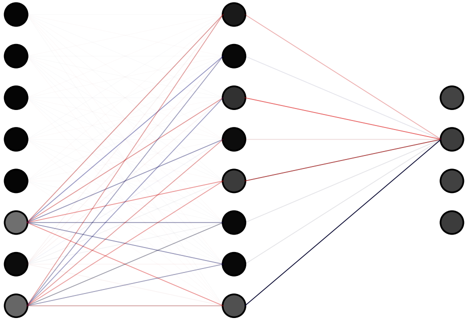
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



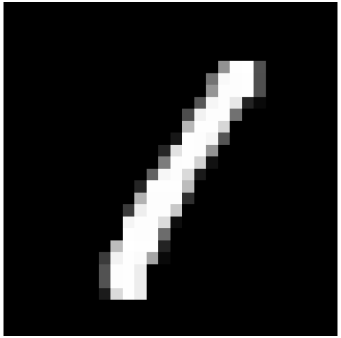


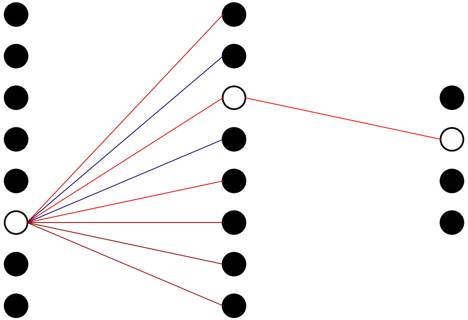
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



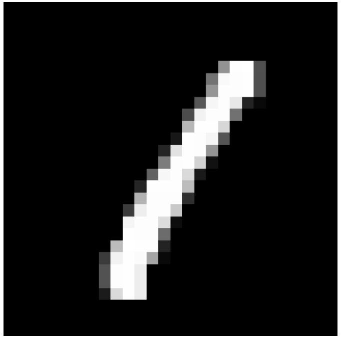


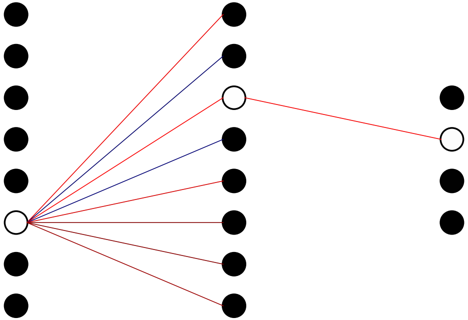
$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

The Multi-layer Perceptron (MLP)

Back-propagation

Iteratively, the neural networks improves its performance.



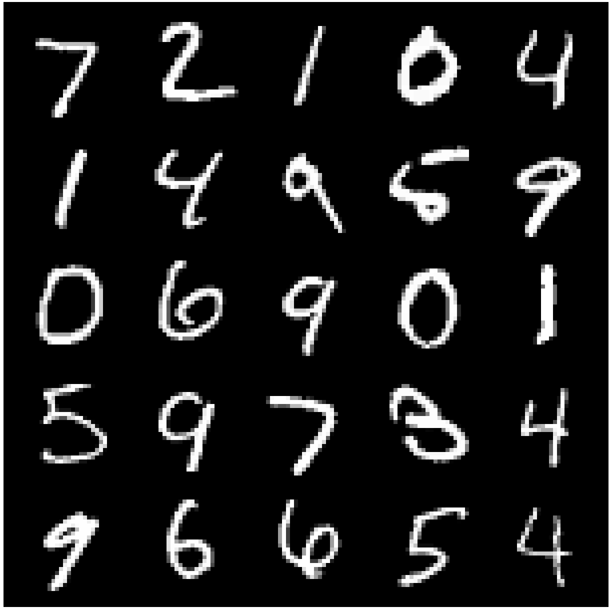


$$x_3^2 = \sigma_3 \left(A_3^{2,1} x_2^1 + A_3^{2,2} x_2^2 + \cdots + A_3^{2,8} x_2^8 \right)$$

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

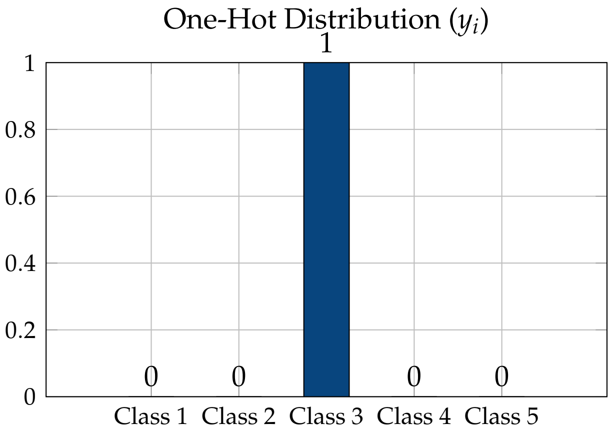
- Input shape : $1 \times 28 \times 28$.
- Number of Classes : 10.
- Number of training samples (x, y): 60000.
- Number of evaluating samples: 10000.
- Loss : cross-entropy

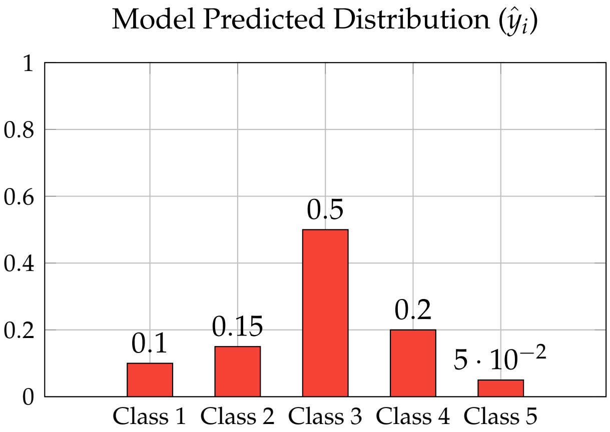
$$L(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij})$$

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:

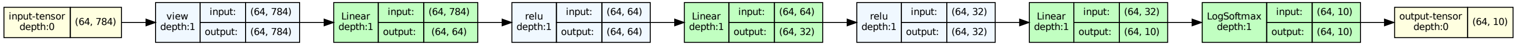
$$l(\hat{y}_i, y_i) = -\sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij}).$$

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)

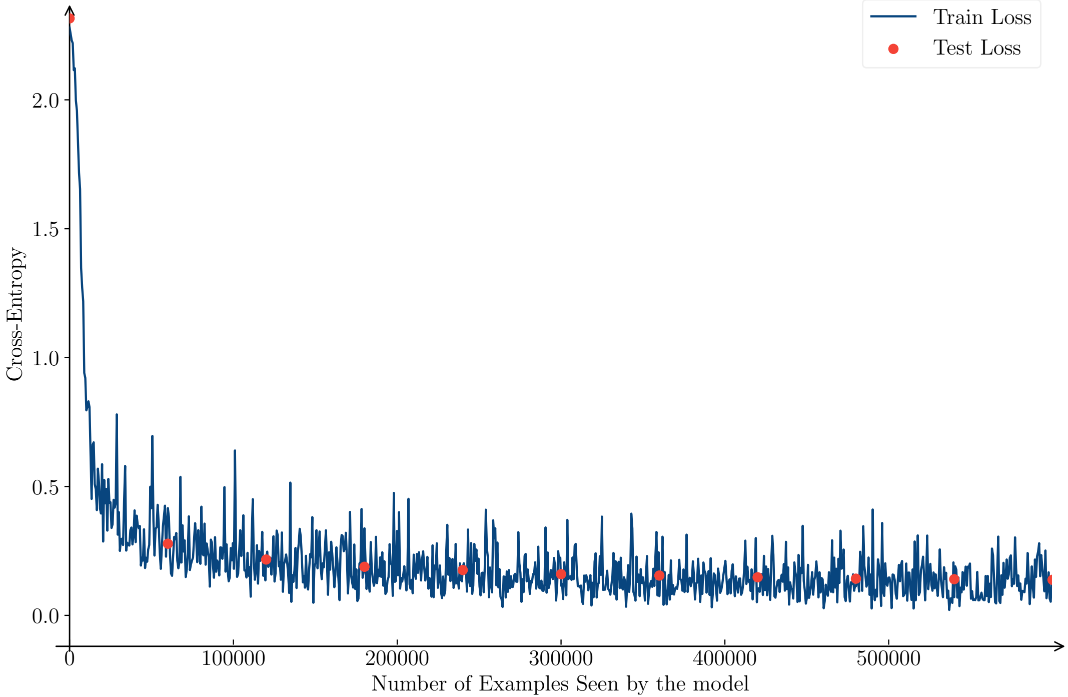
Linear depth:1 input: (64, 32) output: (64, 10)

LogSoftmax depth:1 input: (64, 10) output: (64, 10)

output-tensor depth:0 (64, 10)

The Multi-layer Perceptron (MLP)

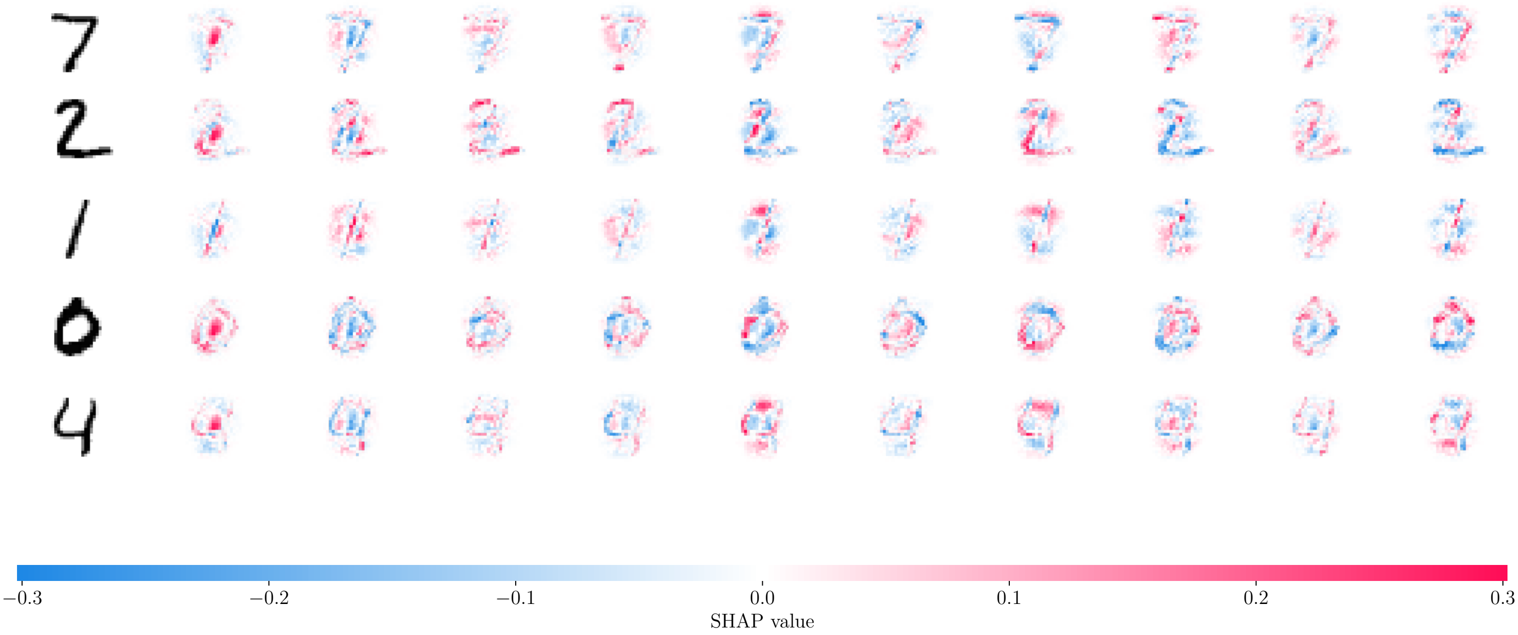
Example : Image classification of handwritten digits from A to Z



The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

With 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

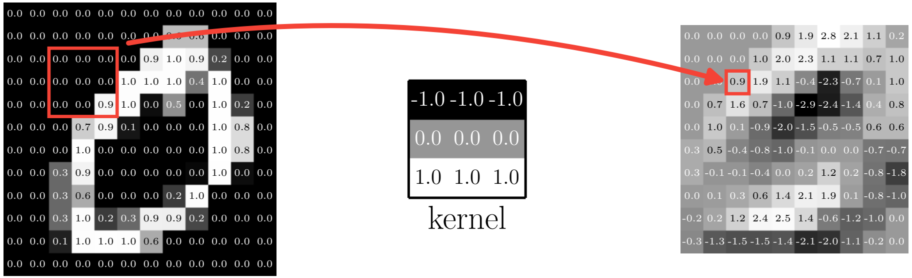
In the field of image processing, the Convolution Operators are widely considered as the most favoured approach. While it has been demonstrated that Dense blocks, or Linear blocks, are capable of accurately classifying images in the case of the MNIST dataset, the need for convolutional transformations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

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 $k \times k$ 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.

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	[illegible]	0.6	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.9	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	1.0	1.0	0.4	1.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.5	0.0	1.0	0.2	0.0
0.0	0.0	0.0	0.7	0.9	0.1	0.0	0.0	0.0	1.0	0.8	0.0
0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.8	0.0
0.0	0.0	0.3	0.9	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
0.0	0.0	0.3	0.6	0.0	0.0	0.0	0.2	1.0	0.0	0.0	0.0
0.0	0.0	0.3	1.0	0.2	0.3	0.9	0.9	0.2	0.0	0.0	0.0
0.0	0.0	0.1	1.0	1.0	1.0	0.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

-1.0	-1.0	-1.0
0.0	0.0	0.0
1.0	1.0	1.0

kernel

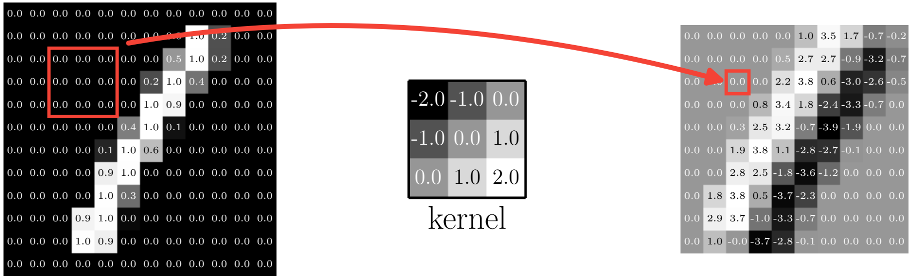
0.0	0.0	0.0	0.0	0.9	1.9	2.8	2.1	1.1	0.2
0.0	0.0	0.0	1.0	2.0	2.3	1.1	1.1	0.7	1.0
0.0	[illegible]	0.9	1.9	1.1	-0.4	-2.3	-0.7	0.1	1.0
0.0	0.7	1.6	0.7	-1.0	-2.9	-2.4	-1.4	0.4	0.8
0.0	1.0	0.1	-0.9	-2.0	-1.5	-0.5	-0.5	0.6	0.6
0.3	0.5	-0.4	-0.8	-1.0	-0.1	0.0	0.0	-0.7	-0.7
0.3	-0.1	-0.1	-0.4	0.0	0.2	1.2	0.2	-0.8	-1.8
0.0	0.1	0.3	0.6	1.4	2.1	1.9	0.1	-0.8	-1.0
-0.2	0.2	1.2	2.4	2.5	1.4	-0.6	-1.2	-1.0	0.0
-0.3	-1.3	-1.5	-1.5	-1.4	-2.1	-2.0	-1.1	-0.2	0.0

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.

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

-2.0	-1.0	0.0
-1.0	0.0	1.0
0.0	1.0	2.0

kernel

0.0	0.0	0.0	0.0	0.0	1.0	3.5	1.7	-0.7	-0.2
0.0	0.0	0.0	0.0	0.5	2.7	2.7	-0.9	-3.2	-0.7
0.0	0.0	0.0	0.0	2.2	3.8	0.6	-3.0	-2.6	-0.5
0.0	0.0	0.0	0.8	3.4	1.8	-2.4	-3.3	-0.7	0.0
0.0	0.0	0.3	2.5	3.2	-0.7	-3.9	-1.9	0.0	0.0
0.0	0.0	1.9	3.8	1.1	-2.8	-2.7	-0.1	0.0	0.0
0.0	0.0	2.8	2.5	-1.8	-3.6	-1.2	0.0	0.0	0.0
0.0	1.8	3.8	0.5	-3.7	-2.3	0.0	0.0	0.0	0.0
0.0	2.9	3.7	-1.0	-3.3	-0.7	0.0	0.0	0.0	0.0
0.0	1.0	-0.0	-3.7	-2.8	-0.1	0.0	0.0	0.0	0.0

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.

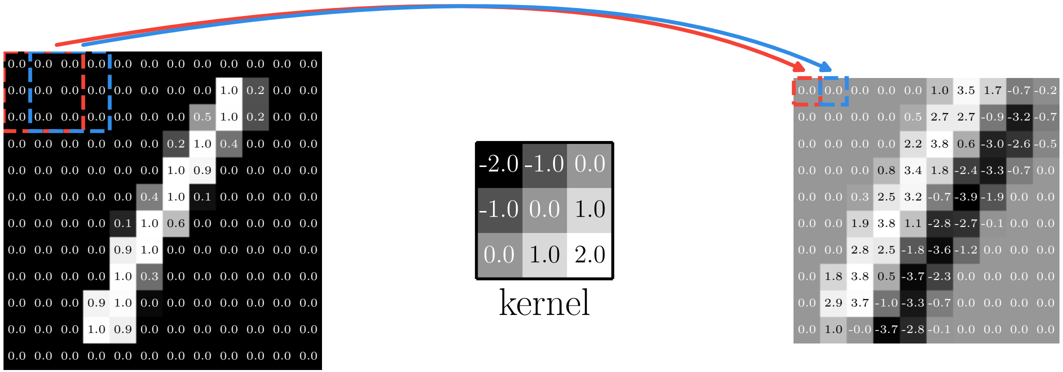
The size out the output is :

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

Convolutional Neural Networks

Kernel Size, Padding and Stride

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

-2.0	-1.0	0.0
-1.0	0.0	1.0
0.0	1.0	2.0

kernel

0.0	0.0	0.0	0.0	0.0	1.0	3.5	1.7	-0.7	-0.2
0.0	0.0	0.0	0.0	0.5	2.7	2.7	-0.9	-3.2	-0.7
0.0	0.0	0.0	0.0	2.2	3.8	0.6	-3.0	-2.6	-0.5
0.0	0.0	0.0	0.8	3.4	1.8	-2.4	-3.3	-0.7	0.0
0.0	0.0	0.3	2.5	3.2	-0.7	-3.9	-1.9	0.0	0.0
0.0	0.0	1.9	3.8	1.1	-2.8	-2.7	-0.1	0.0	0.0
0.0	0.0	2.8	2.5	-1.8	-3.6	-1.2	0.0	0.0	0.0
0.0	1.8	3.8	0.5	-3.7	-2.3	0.0	0.0	0.0	0.0
0.0	2.9	3.7	-1.0	-3.3	-0.7	0.0	0.0	0.0	0.0
0.0	1.0	-0.0	-3.7	-2.8	-0.1	0.0	0.0	0.0	0.0

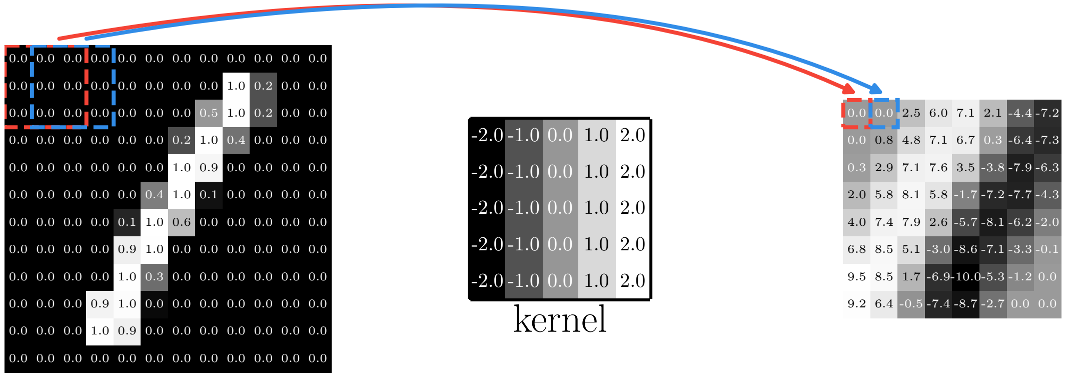
Size 12×12 Size 3×3 Size 10×10

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

Convolutional Neural Networks

Kernel Size, Padding and Stride

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Size 12 × 12

-2.0	-1.0	0.0	1.0	2.0
-2.0	-1.0	0.0	1.0	2.0
-2.0	-1.0	0.0	1.0	2.0
-2.0	-1.0	0.0	1.0	2.0
-2.0	-1.0	0.0	1.0	2.0

Size 5 × 5

0.0	0.0	2.5	6.0	7.1	2.1	-4.4	-7.2
0.0	0.8	4.8	7.1	6.7	0.3	-6.4	-7.3
0.3	2.9	7.1	7.6	3.5	-3.8	-7.9	-6.3
2.0	5.8	8.1	5.8	-1.7	-7.2	-7.7	-4.3
4.0	7.4	7.9	2.6	-5.7	-8.1	-6.2	-2.0
6.8	8.5	5.1	-3.0	-8.6	-7.1	-3.3	-0.1
9.5	8.5	1.7	-6.9	-10.0	-5.3	-1.2	0.0
9.2	6.4	-0.5	-7.4	-8.7	-2.7	0.0	0.0

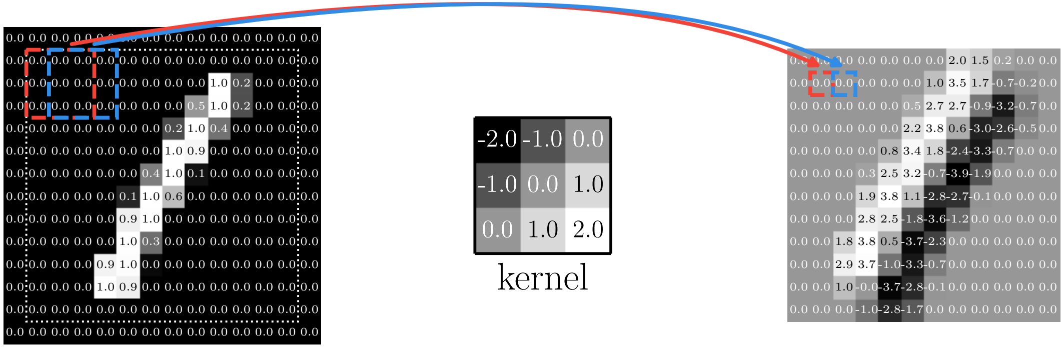
Size 8 × 8

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

Convolutional Neural Networks

Kernel Size, Padding and Stride

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Size 12 × 12

-2.0	-1.0	0.0
-1.0	0.0	1.0
0.0	1.0	2.0

kernel

Size 3 × 3

0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	1.5	0.2	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	1.0	3.5	1.7	-0.7	-0.2	0.0
0.0	0.0	0.0	0.0	0.0	0.5	2.7	2.7	-0.9	-3.2	-0.7	0.0
0.0	0.0	0.0	0.0	0.0	2.2	3.8	0.6	-3.0	-2.6	-0.5	0.0
0.0	0.0	0.0	0.0	0.8	3.4	1.8	-2.4	-3.3	-0.7	0.0	0.0
0.0	0.0	0.0	0.3	2.5	3.2	-0.7	-3.9	-1.9	0.0	0.0	0.0
0.0	0.0	0.0	1.9	3.8	1.1	-2.8	-2.7	-0.1	0.0	0.0	0.0
0.0	0.0	0.0	2.8	2.5	-1.8	-3.6	-1.2	0.0	0.0	0.0	0.0
0.0	0.0	1.8	3.8	0.5	-3.7	-2.3	0.0	0.0	0.0	0.0	0.0
0.0	0.0	2.9	3.7	-1.0	-3.3	-0.7	0.0	0.0	0.0	0.0	0.0
0.0	0.0	1.0	-0.0	-3.7	-2.8	-0.1	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	-1.0	-2.8	-1.7	0.0	0.0	0.0	0.0	0.0	0.0

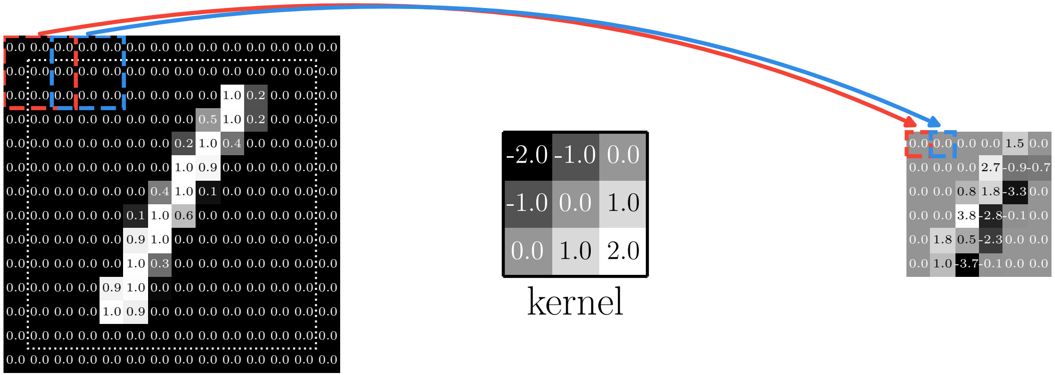
Size 12 × 12

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

Convolutional Neural Networks

Kernel Size, Padding and Stride

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Size 12 × 12

-2.0	-1.0	0.0
-1.0	0.0	1.0
0.0	1.0	2.0

kernel

Size 3 × 3

0.0	0.0	0.0	0.0	1.5	0.0
0.0	0.0	0.0	2.7	-0.9	-0.7
0.0	0.0	0.8	1.8	-3.3	0.0
0.0	0.0	3.8	-2.8	-0.1	0.0
0.0	1.8	0.5	-2.3	0.0	0.0
0.0	1.0	-3.7	-0.1	0.0	0.0

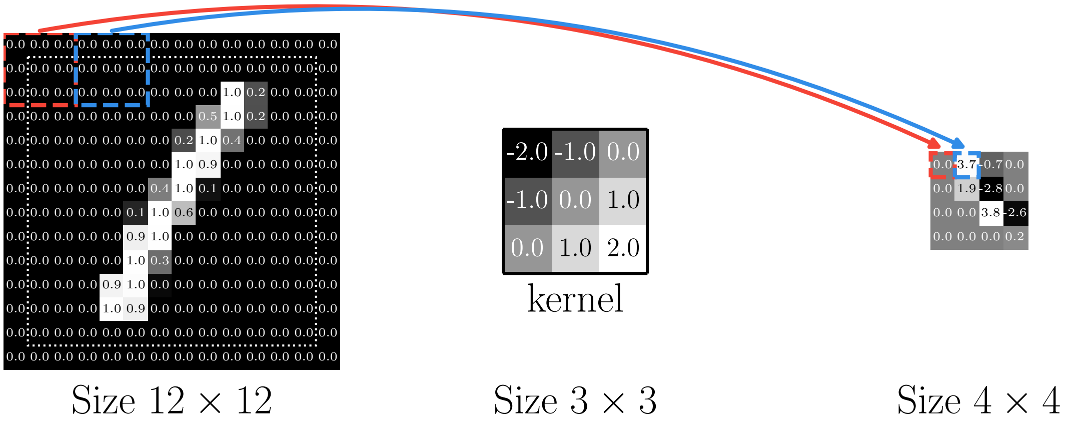
Size 6 × 6

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

Convolutional Neural Networks

Kernel Size, Padding and Stride

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



0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5	1.0	0.2	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	1.0	0.4	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.4	1.0	0.1	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.1	1.0	0.6	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	1.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.9	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	1.0	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Size 12 × 12

-2.0	-1.0	0.0
-1.0	0.0	1.0
0.0	1.0	2.0

kernel

Size 3 × 3

0.0	3.7	-0.7	0.0
0.0	1.9	-2.8	0.0
0.0	0.0	3.8	-2.6
0.0	0.0	0.0	0.2

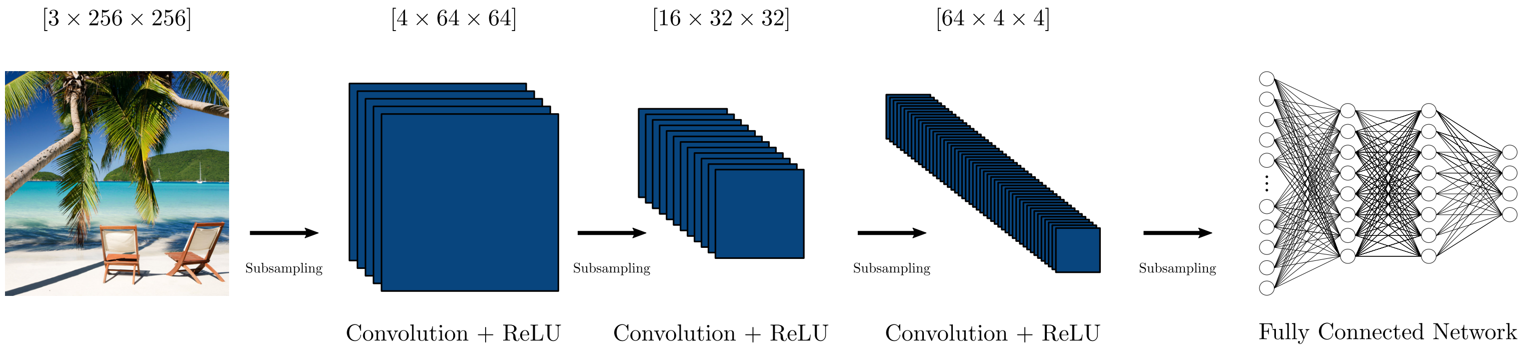
Size 4 × 4

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

Convolutional Neural Networks

CNN : Convolutional in a network Networks

We can represent a CNN as under this form:



$[3 \times 256 \times 256]$ $[4 \times 64 \times 64]$ $[16 \times 32 \times 32]$ $[64 \times 4 \times 4]$

Subsampling Subsampling Subsampling Subsampling

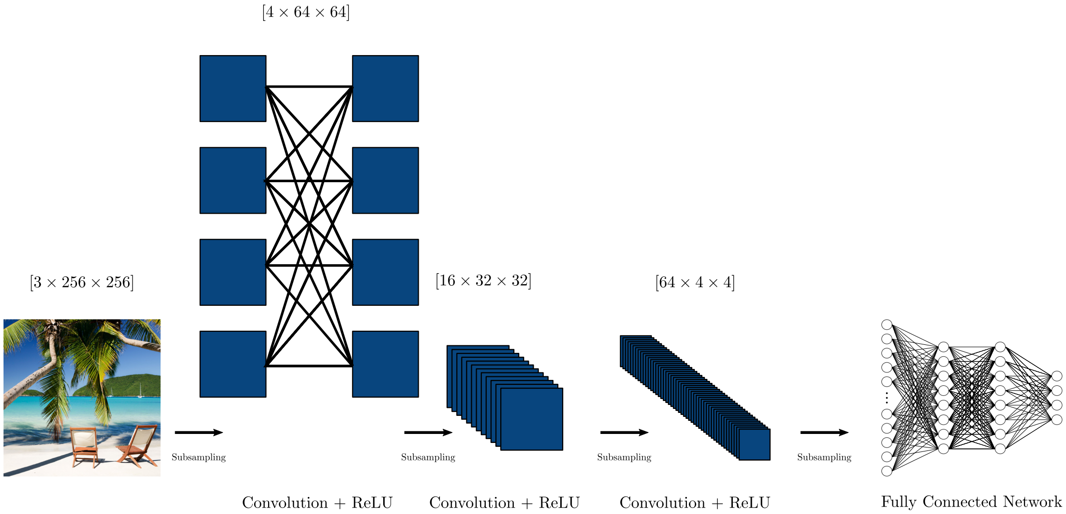
Convolution + ReLU Convolution + ReLU Convolution + ReLU Fully Connected Network

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:

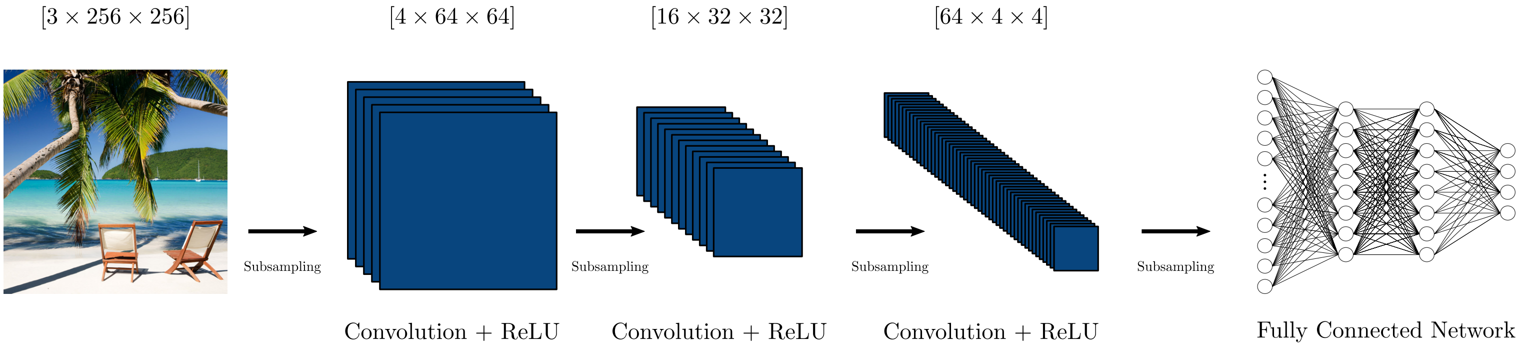
$$\text{out}_{i,j}(c_{\text{out}}) = \text{bias}(c_{\text{out}}) + \sum_{k=0}^{|c_{\text{in}}|-1} \text{Conv}(\text{input}(k), \text{kernel}_k)_{i,j}$$



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.



$[3 \times 256 \times 256]$ $[4 \times 64 \times 64]$ $[16 \times 32 \times 32]$ $[64 \times 4 \times 4]$

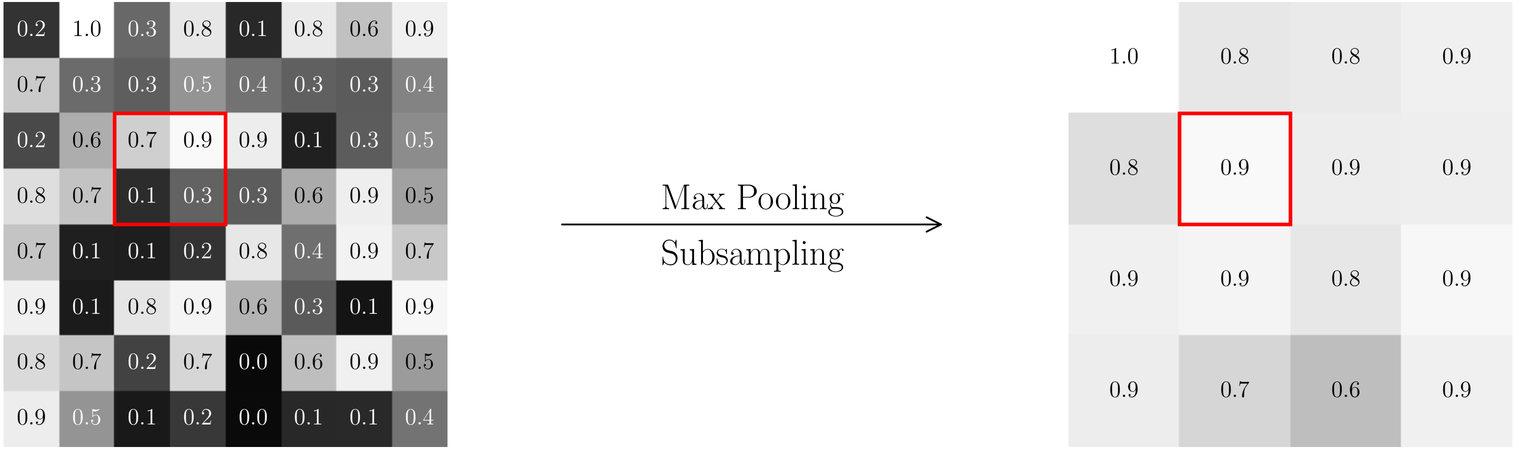
Subsampling Subsampling Subsampling Subsampling

Convolution + ReLU Convolution + ReLU Convolution + ReLU Fully Connected Network

Convolutional Neural Networks

Max Pooling

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



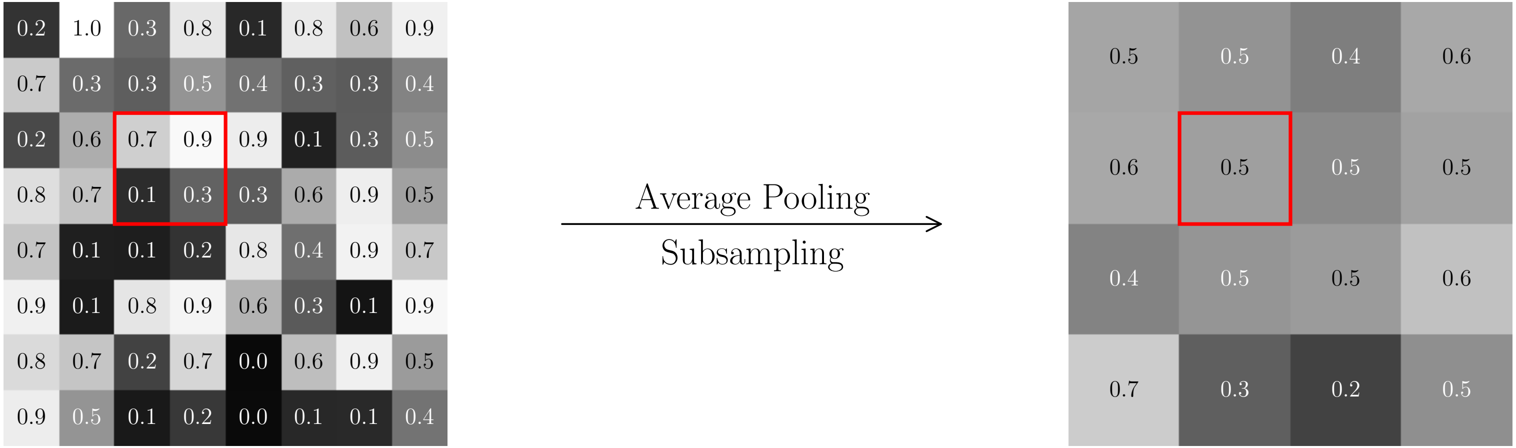
0.2	1.0	0.3	0.8	0.1	0.8	0.6	0.9
0.7	0.3	0.3	0.5	0.4	0.3	0.3	0.4
0.2	0.6	0.7	0.9	0.9	0.1	0.3	0.5
0.8	0.7	0.1	0.3	0.3	0.6	0.9	0.5
0.7	0.1	0.1	0.2	0.8	0.4	0.9	0.7
0.9	0.1	0.8	0.9	0.6	0.3	0.1	0.9
0.8	0.7	0.2	0.7	0.0	0.6	0.9	0.5
0.9	0.5	0.1	0.2	0.0	0.1	0.1	0.4

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

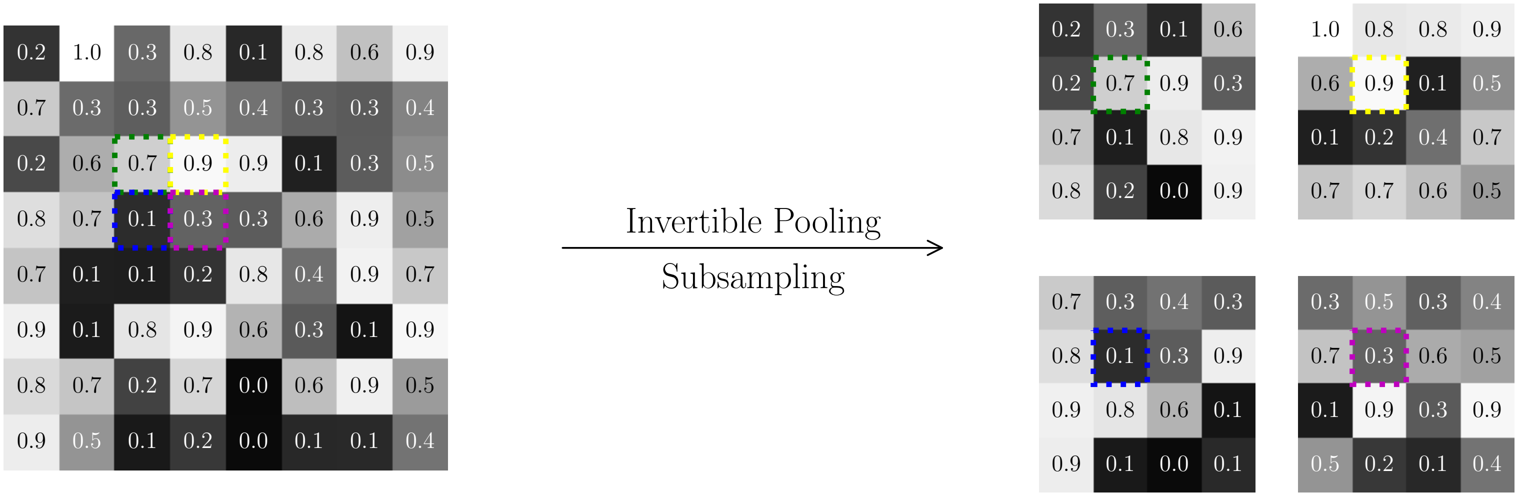
$$\xrightarrow[\text{Subsampling}]{\text{Average Pooling}}$$

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.



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

0.2	0.3	0.1	0.6
0.2	0.7	0.9	0.3
0.7	0.1	0.8	0.9
0.8	0.2	0.0	0.9

1.0	0.8	0.8	0.9
0.6	0.9	0.1	0.5
0.1	0.2	0.4	0.7
0.7	0.7	0.6	0.5

0.7	0.3	0.4	0.3
0.8	0.1	0.3	0.9
0.9	0.8	0.6	0.1
0.9	0.1	0.0	0.1

0.3	0.5	0.3	0.4
0.7	0.3	0.6	0.5
0.1	0.9	0.3	0.9
0.5	0.2	0.1	0.4

Convolutional Neural Networks

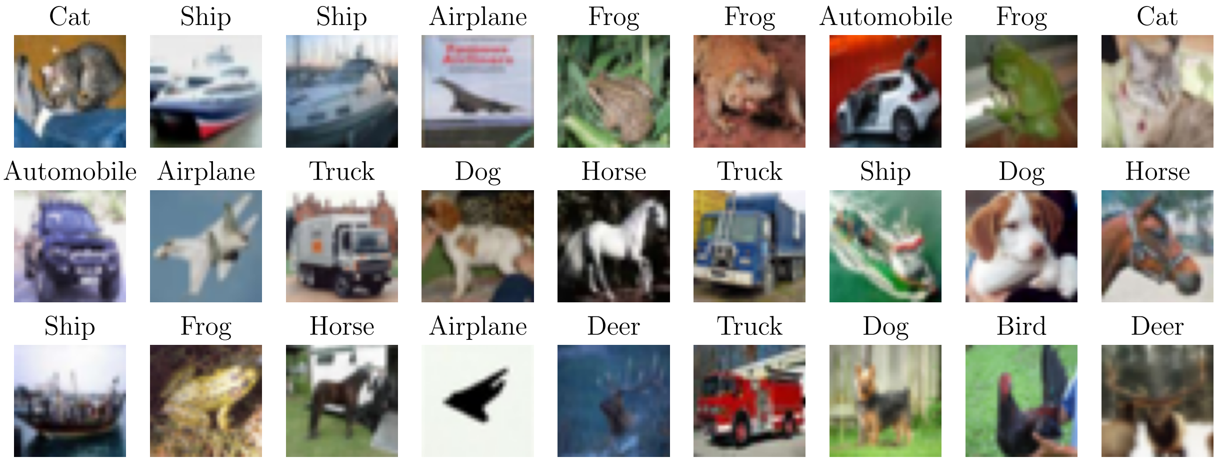
CNN : Convolutional in a network Networks

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:

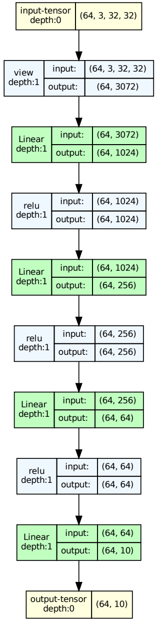
- Model 1 : Fully Connected Neural Network with 3.4 million parameters.
- Model 2 : CNN with 62 thousand parameters.
- Model 3 : Wider and longer CNN with 5.8 million parameters.

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



input-tensor depth:0 (64, 3, 32, 32)

view depth:1 input: (64, 3, 32, 32) output: (64, 3072)

Linear depth:1 input: (64, 3072) output: (64, 1024)

relu depth:1 input: (64, 1024) output: (64, 1024)

Linear depth:1 input: (64, 1024) output: (64, 256)

relu depth:1 input: (64, 256) output: (64, 256)

Linear depth:1 input: (64, 256) output: (64, 64)

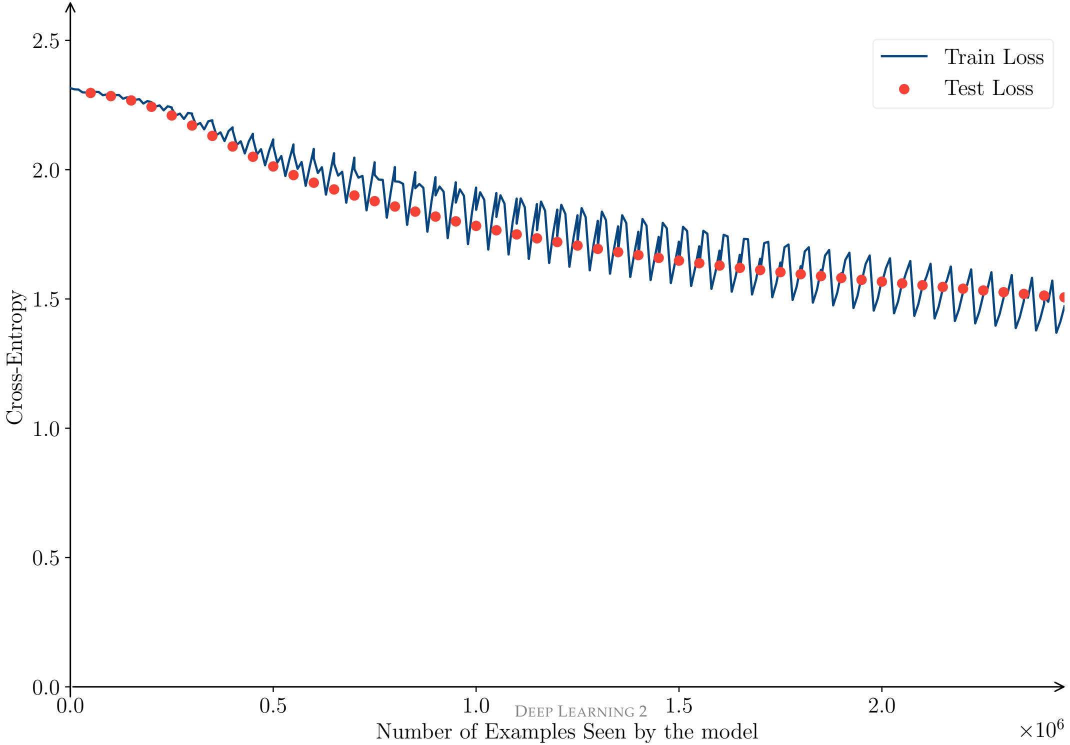
relu depth:1 input: (64, 64) output: (64, 64)

Linear depth:1 input: (64, 64) output: (64, 10)

output-tensor depth:0 (64, 10)

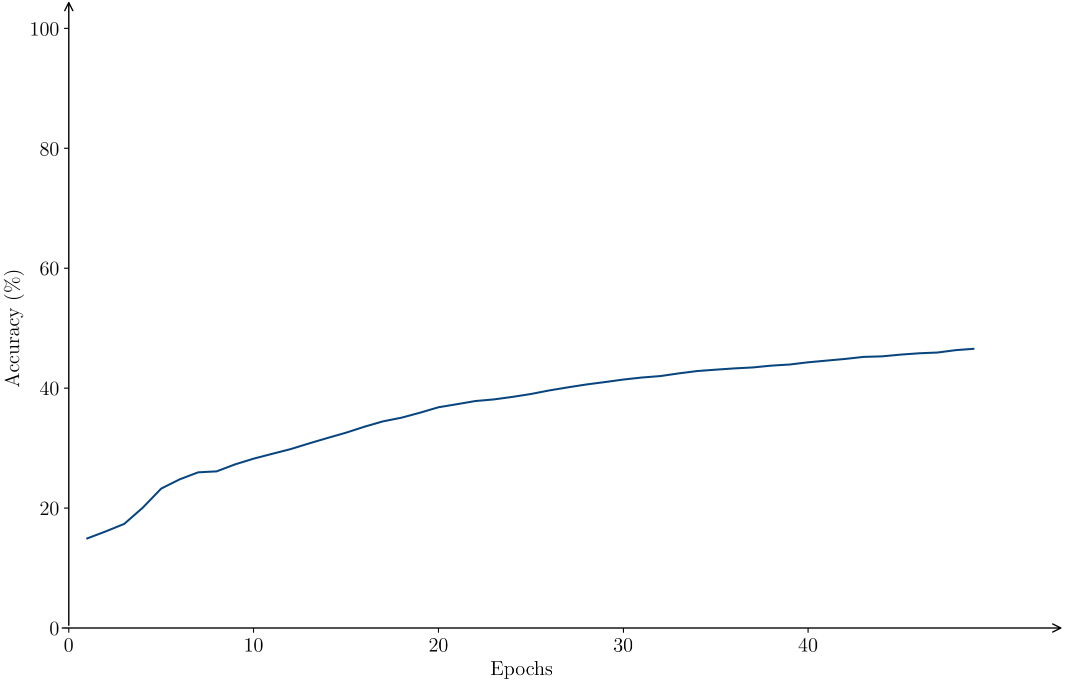
Convolutional Neural Networks

Model 1



Convolutional Neural Networks

Model 1

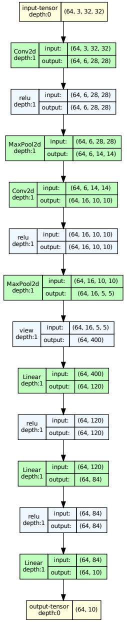


Convolutional Neural Networks

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



input-tensor depth:0 (64, 3, 32, 32)

Conv2d depth:1 input: (64, 3, 32, 32) output: (64, 6, 28, 28)

relu depth:1 input: (64, 6, 28, 28) output: (64, 6, 28, 28)

MaxPool2d depth:1 input: (64, 6, 28, 28) output: (64, 6, 14, 14)

Conv2d depth:1 input: (64, 6, 14, 14) output: (64, 16, 10, 10)

relu depth:1 input: (64, 16, 10, 10) output: (64, 16, 10, 10)

MaxPool2d depth:1 input: (64, 16, 10, 10) output: (64, 16, 5, 5)

view depth:1 input: (64, 16, 5, 5) output: (64, 400)

Linear depth:1 input: (64, 400) output: (64, 120)

relu depth:1 input: (64, 120) output: (64, 120)

Linear depth:1 input: (64, 120) output: (64, 84)

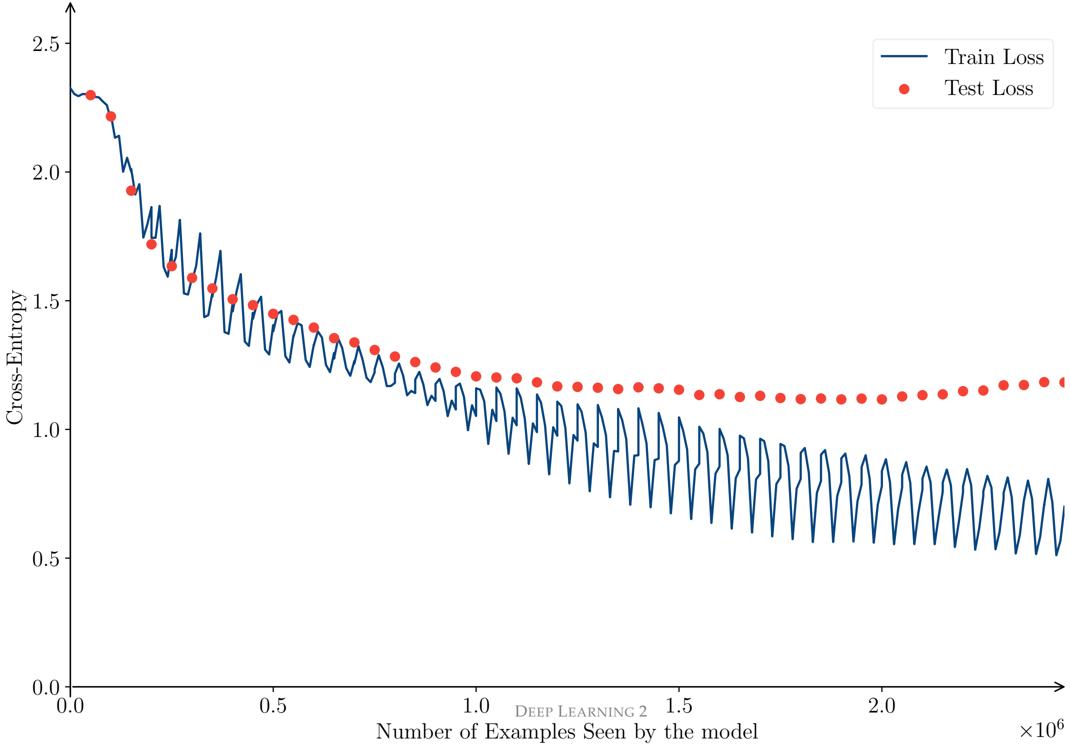
relu depth:1 input: (64, 84) output: (64, 84)

Linear depth:1 input: (64, 84) output: (64, 10)

output-tensor depth:0 (64, 10)

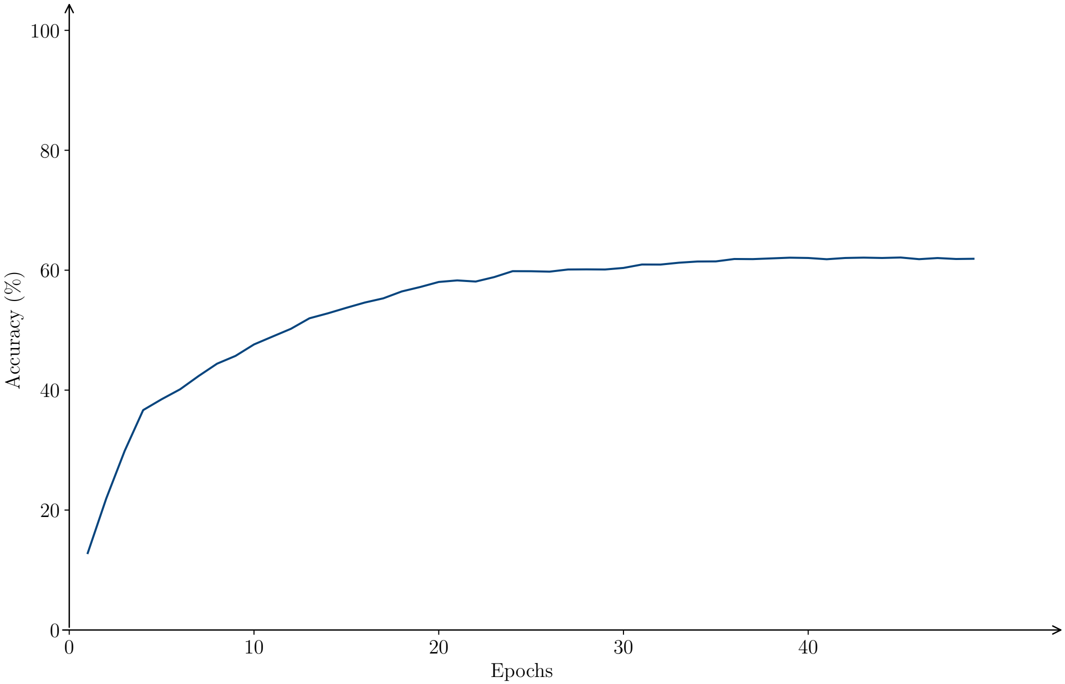
Convolutional Neural Networks

Model 2



Convolutional Neural Networks

Model 2



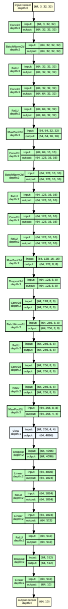
Convolutional Neural Networks

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

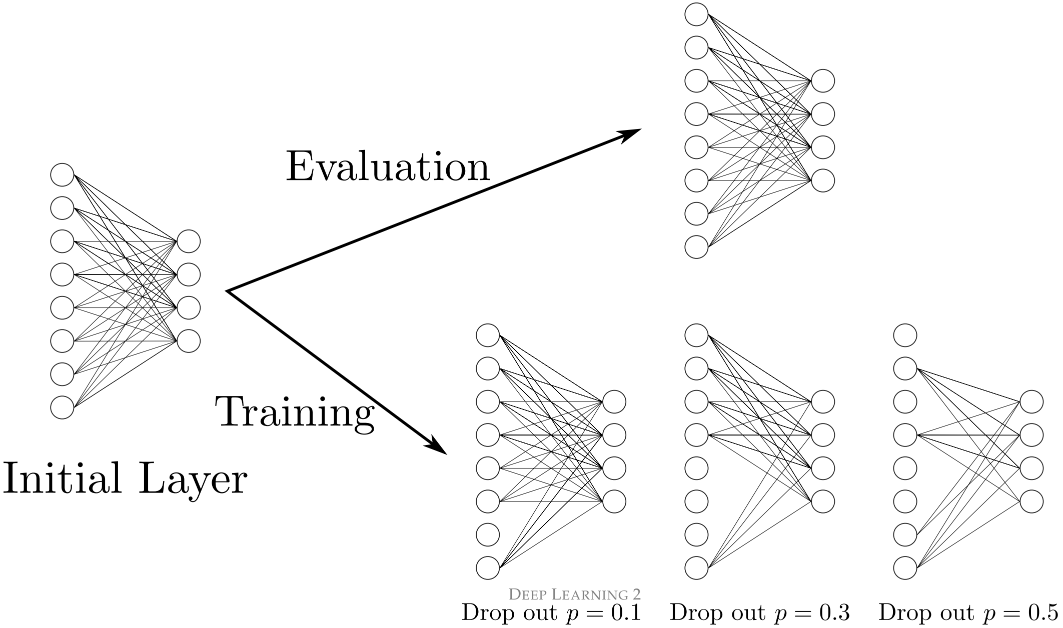
We have added Batch Normalization to improve the training stability and Drop Out to reduce overfitting.


input-tensor depth:0 (64, 3, 32, 32)
Conv2d depth:2 input: (64, 3, 32, 32) output: (64, 32, 32, 32)
BatchNorm2d depth:2 input: (64, 32, 32, 32) output: (64, 32, 32, 32)
ReLU depth:2 input: (64, 32, 32, 32) output: (64, 32, 32, 32)
Conv2d depth:2 input: (64, 32, 32, 32) output: (64, 64, 32, 32)
ReLU depth:2 input: (64, 64, 32, 32) output: (64, 64, 32, 32)
MaxPool2d depth:2 input: (64, 64, 32, 32) output: (64, 64, 16, 16)
Conv2d depth:2 input: (64, 64, 16, 16) output: (64, 128, 16, 16)
BatchNorm2d depth:2 input: (64, 128, 16, 16) output: (64, 128, 16, 16)
ReLU depth:2 input: (64, 128, 16, 16) output: (64, 128, 16, 16)
Conv2d depth:2 input: (64, 128, 16, 16) output: (64, 128, 16, 16)
ReLU depth:2 input: (64, 128, 16, 16) output: (64, 128, 16, 16)
MaxPool2d depth:2 input: (64, 128, 16, 16) output: (64, 128, 8, 8)
Dropout2d depth:2 input: (64, 128, 8, 8) output: (64, 128, 8, 8)
Conv2d depth:2 input: (64, 128, 8, 8) output: (64, 256, 8, 8)
BatchNorm2d depth:2 input: (64, 256, 8, 8) output: (64, 256, 8, 8)
ReLU depth:2 input: (64, 256, 8, 8) output: (64, 256, 8, 8)
Conv2d depth:2 input: (64, 256, 8, 8) output: (64, 256, 8, 8)
ReLU depth:2 input: (64, 256, 8, 8) output: (64, 256, 8, 8)
MaxPool2d depth:2 input: (64, 256, 8, 8) output: (64, 256, 4, 4)
view depth:1 input: (64, 256, 4, 4) output: (64, 4096)
Dropout depth:2 input: (64, 4096) output: (64, 4096)
Linear depth:2 input: (64, 4096) output: (64, 1024)
ReLU depth:2 input: (64, 1024) output: (64, 1024)
Linear depth:2 input: (64, 1024) output: (64, 512)
ReLU depth:2 input: (64, 512) output: (64, 512)
Dropout depth:2 input: (64, 512) output: (64, 512)
Linear depth:2 input: (64, 512) output: (64, 10)
output-tensor depth:0 (64, 10)

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.



Convolutional Neural Networks

Batch Normalization

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.

Convolutional Neural Networks

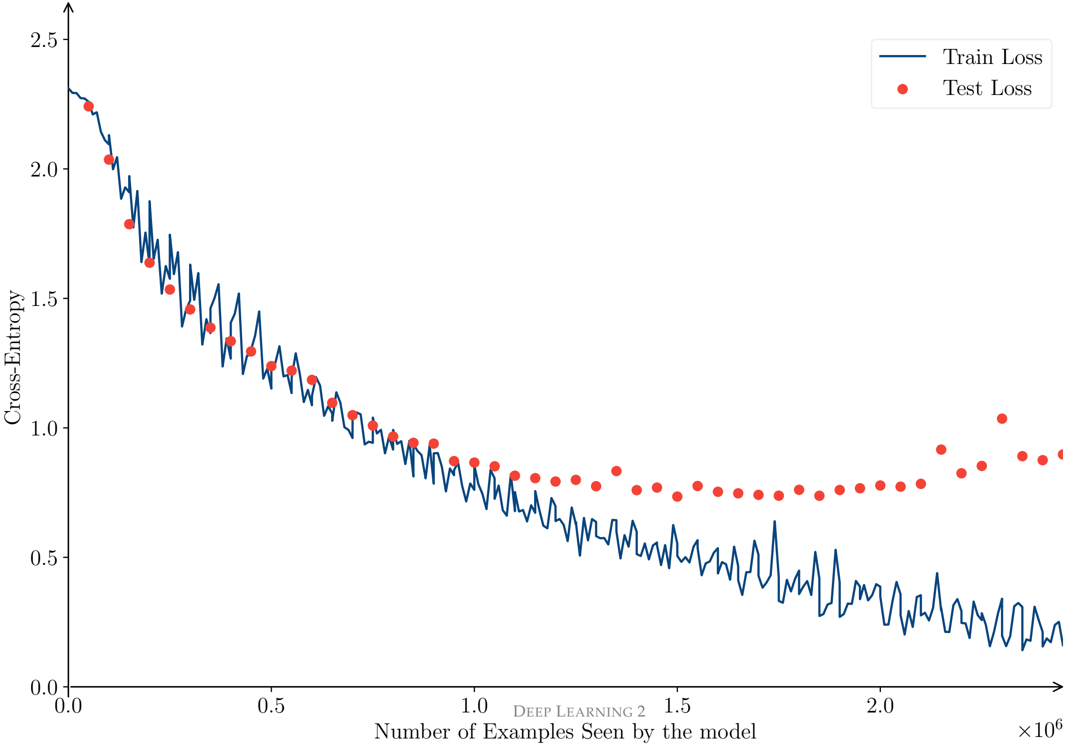
Batch Normalization

```
for each $x_i$ in a mini-batch $B$ of size $b$ do
    Compute the mean $\mu_B$ and variance $\sigma_B^2$ of the features in the mini-batch $B$.
        $$\mu_B = \frac{1}{b}\sum_i x_i \quad \text{and} \quad \sigma_B^2 = \frac{1}{m}\sum_i (x_i - \mu_B)^2$$
    Normalize each feature $x_i$ in the mini-batch $B$ using $\mu_B$ and $\sigma_B^2$.
        $$\bar{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$$
    Scale and shift each normalized feature $x_i$ using two learnable parameters $\gamma$ and $\beta$ respectively.
        $$y_i = \gamma \bar{x}_i + \beta$$
end for
```

Algorithm 1: Batch Normalization

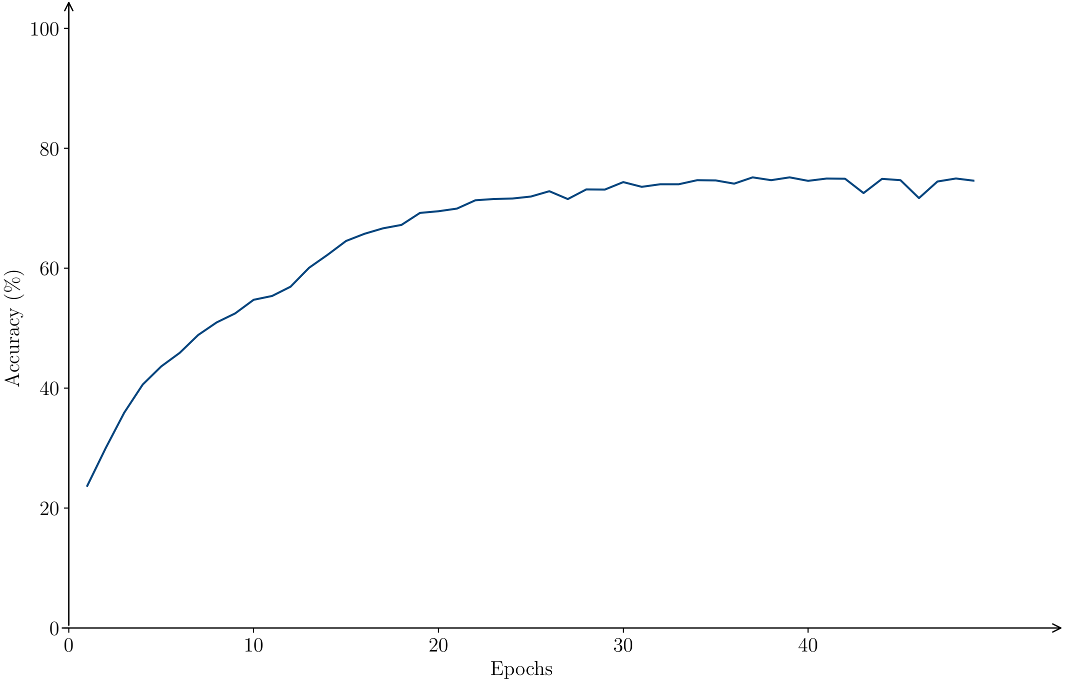
Convolutional Neural Networks

Model 3



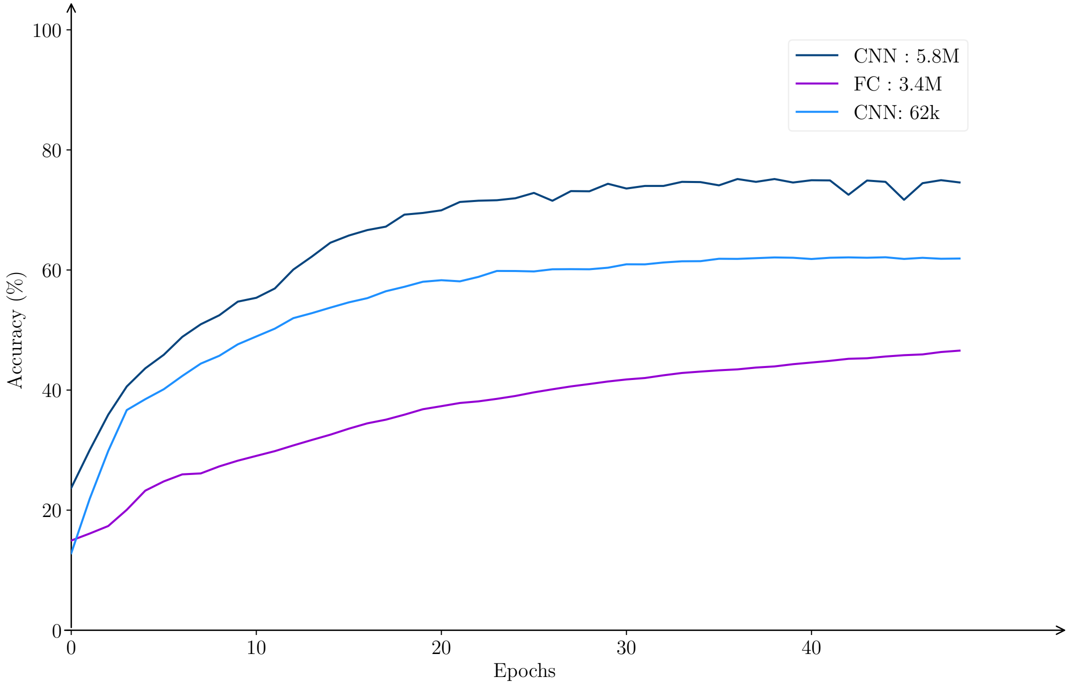
Convolutional Neural Networks

Model 3



Convolutional Neural Networks

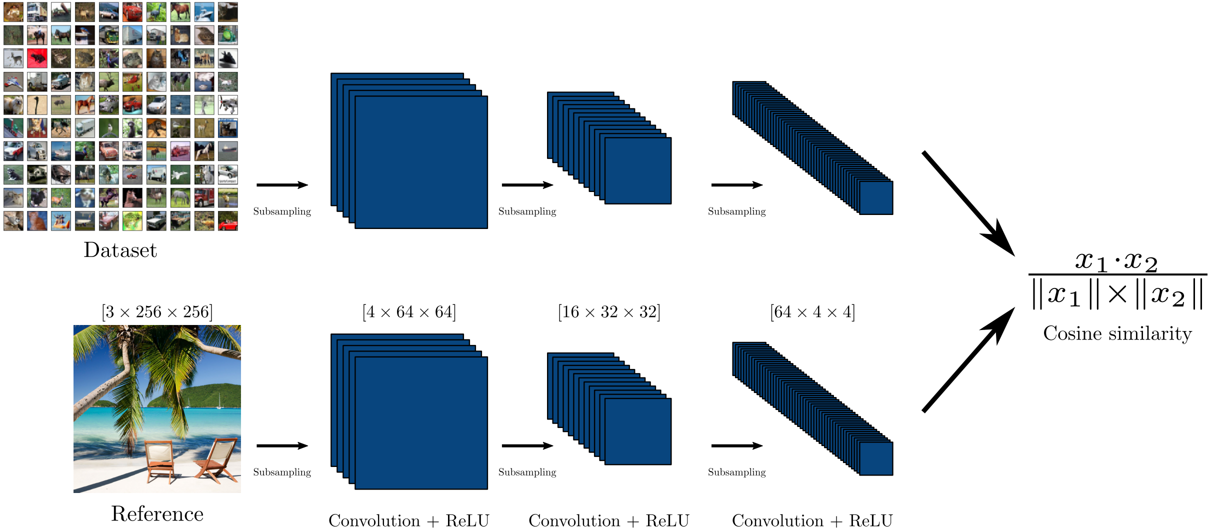
CNN in practice: CIFAR 10



Convolutional Neural Networks

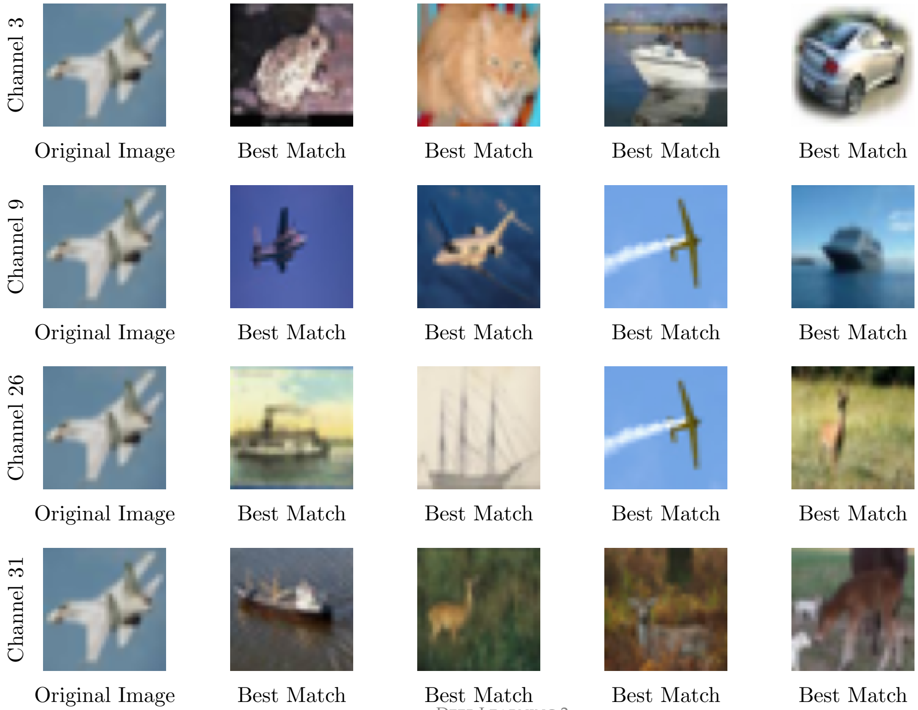
Intuition behind channels

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



Channel 3

Original Image | Best Match | Best Match | Best Match | Best Match

Channel 9

Original Image | Best Match | Best Match | Best Match | Best Match

Channel 26

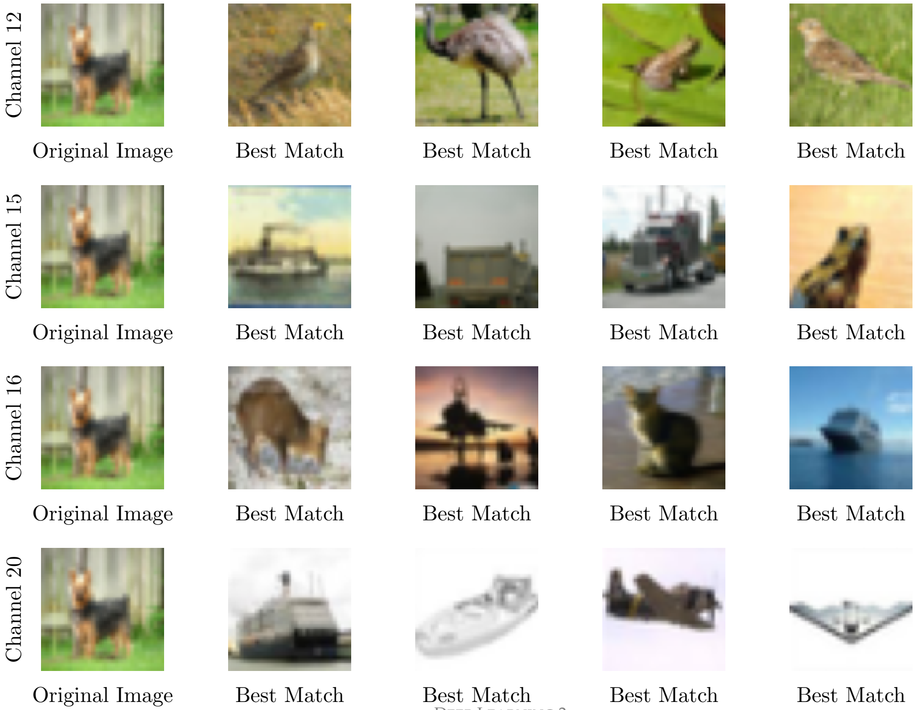
Original Image | Best Match | Best Match | Best Match | Best Match

Channel 31

Original Image | Best Match | Best Match | Best Match | Best Match

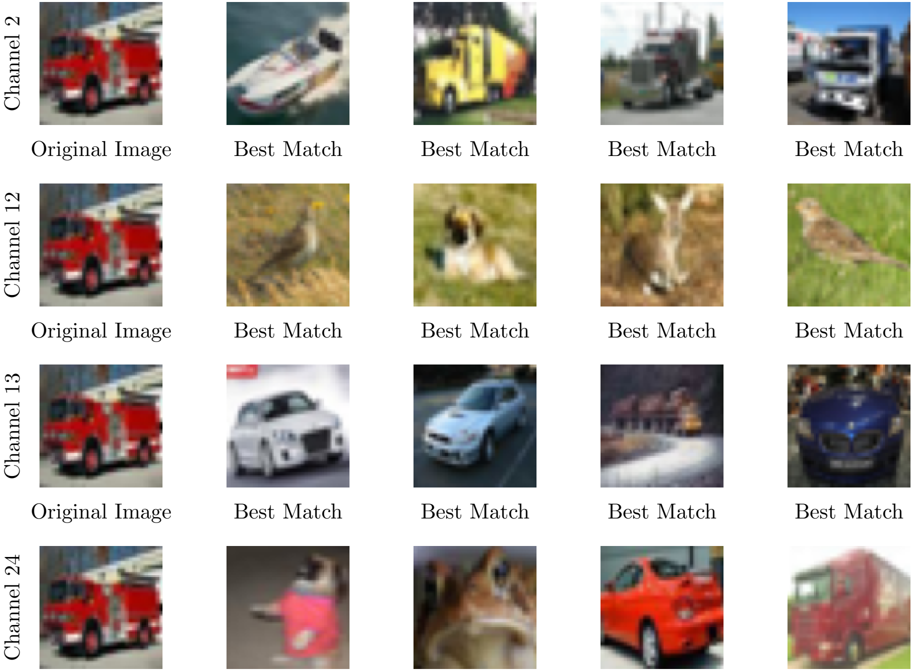
Convolutional Neural Networks

Intuition behind channels



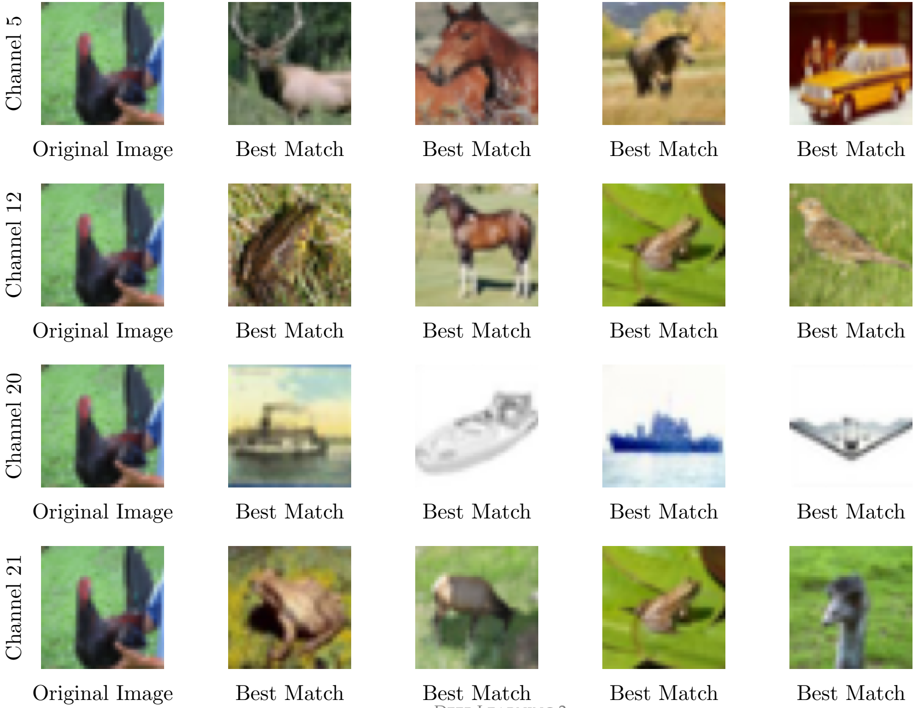
Convolutional Neural Networks

Intuition behind channels



Convolutional Neural Networks

Intuition behind channels



Original Image Best Match Best Match Best Match Best Match

Original Image Best Match Best Match Best Match Best Match

Original Image Best Match Best Match Best Match Best Match

Original Image Best Match Best Match Best Match Best Match

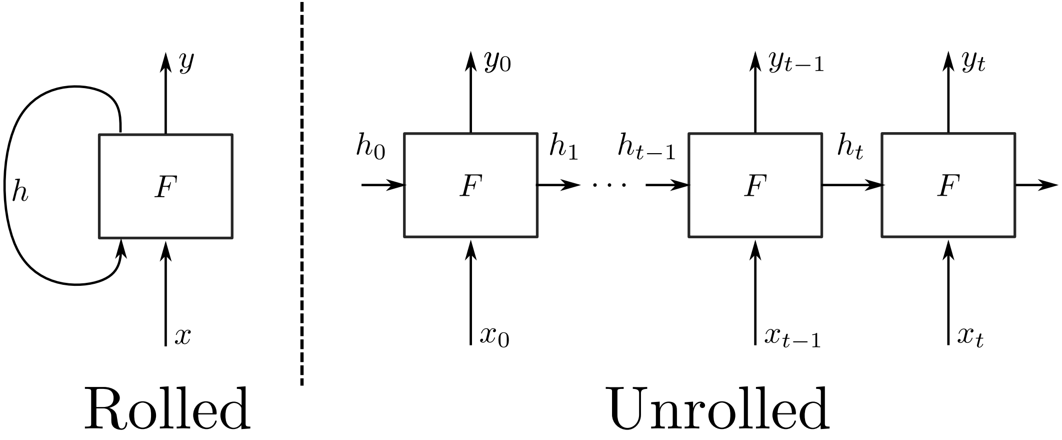
Recurrent Neural Networks

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.

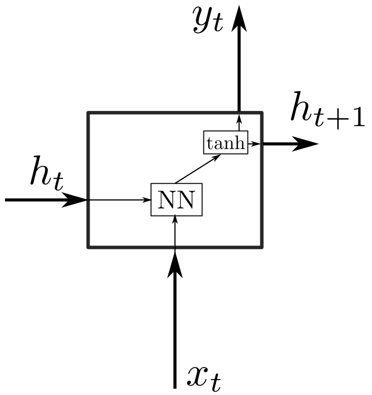


Recurrent Neural Networks

Recurrent Block

Some of the limitations of Vanilla RNNs:

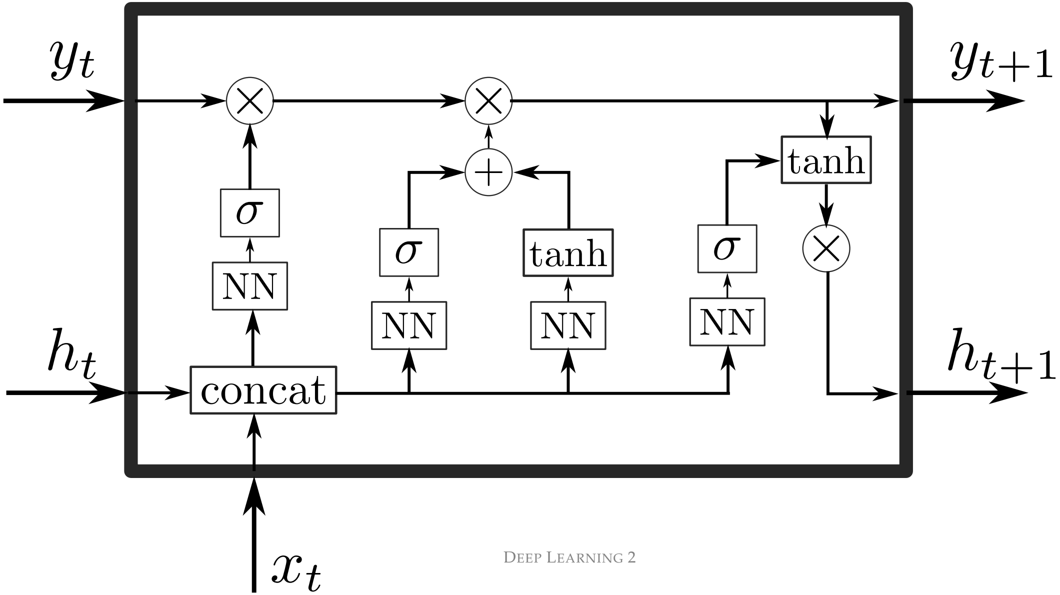
- Vanishing gradient problem: The gradient signals used to update the weights during training can become very small, making it difficult to train RNNs effectively.
- Exploding gradient problem: On the other hand, gradients can become too large and cause numeric instability, making it difficult to train 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

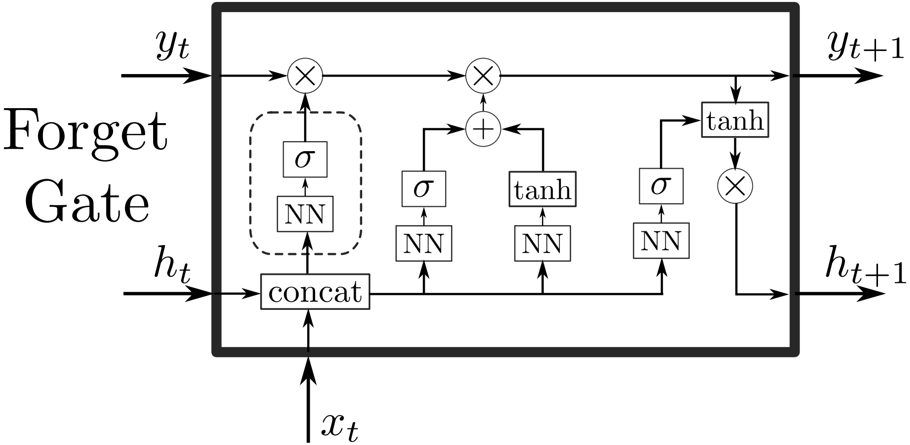
Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for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



Recurrent Neural Networks

LSTM

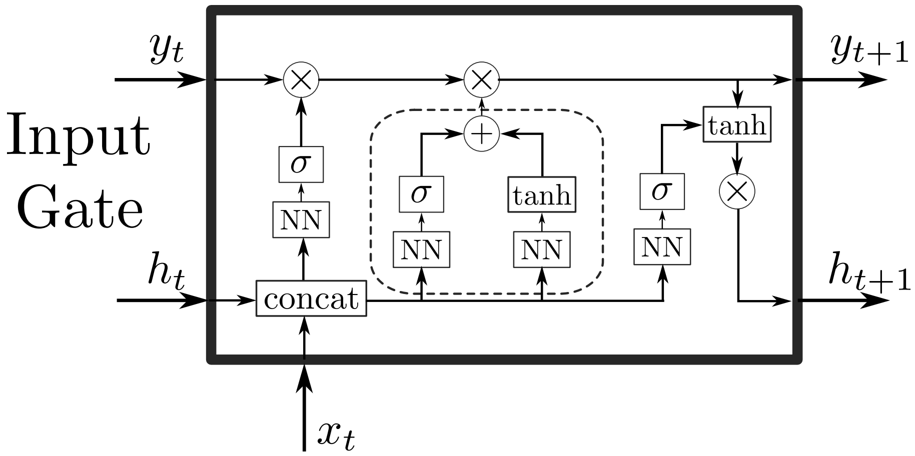
Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the problem of vanishing gradients and the difficulty of learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process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



Recurrent Neural Networks

LSTM

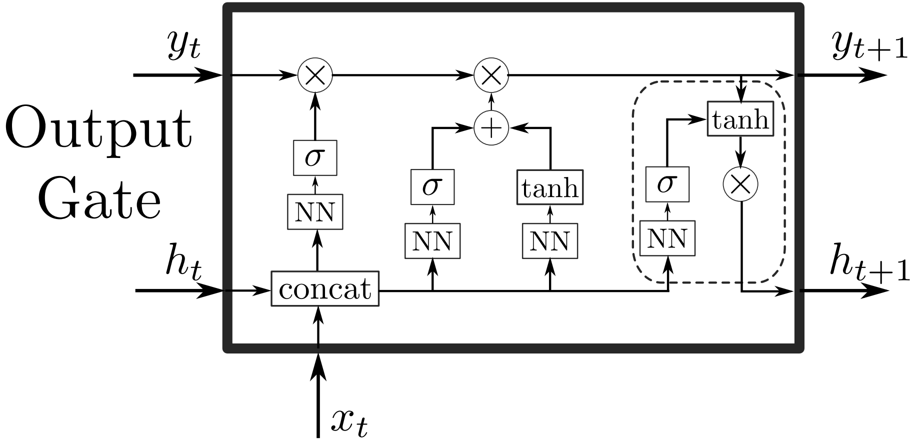
Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for 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.



Limits of LSTM

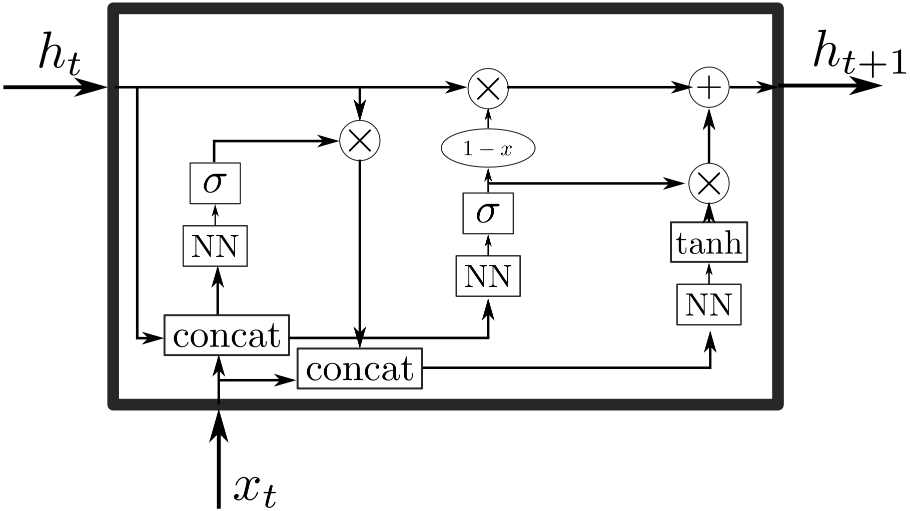
Limitations of LSTM RNNs:

- 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

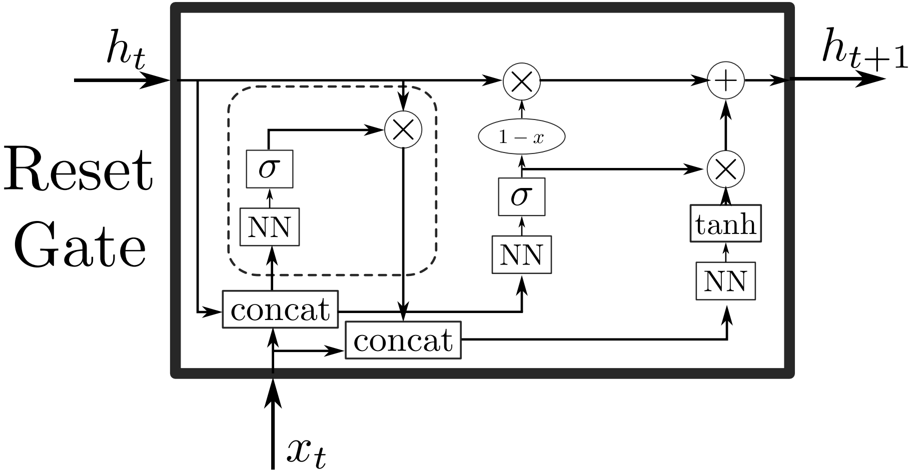
GRU blocks, or Gated Recurrent Units, are a type of recurrent neural network architecture that are similar to LSTMs in their function and ability to process sequential data. GRUs were introduced as a simplification of LSTMs, with the aim of reducing the number of parameters in the network and improving 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

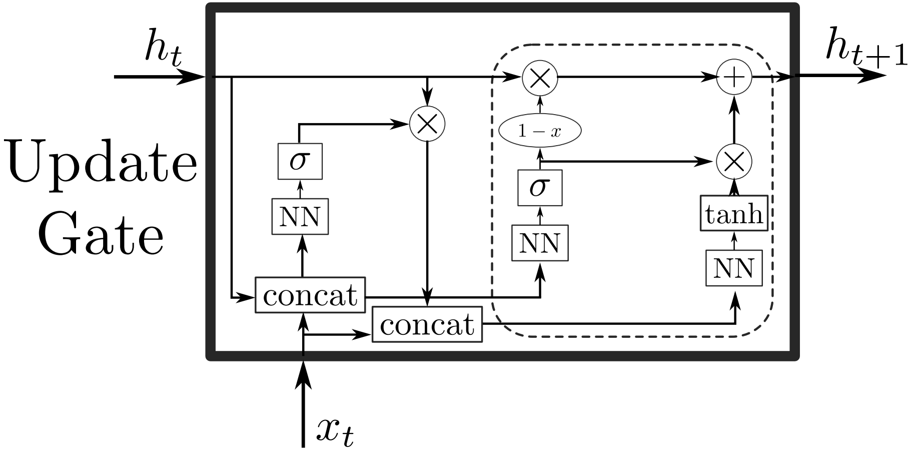
GRU blocks, or Gated Recurrent Units, are a type of recurrent neural network architecture that are similar to LSTMs in their function and ability to process sequential data. GRUs were introduced as a simplification of LSTMs, with the aim of reducing the number of parameters in the network and improving 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.

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.

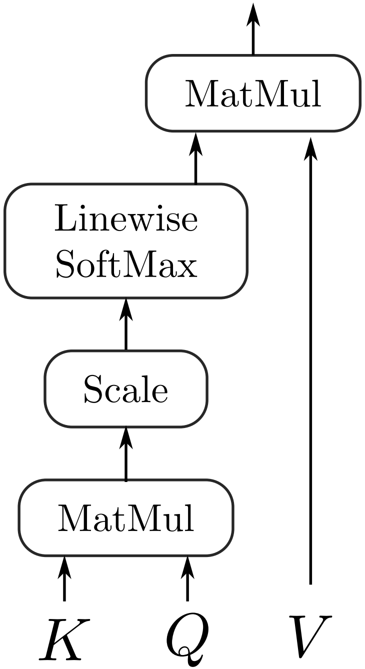
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad \text{where } Q \in \mathbb{R}^{m\times d_k},\ K \in \mathbb{R}^{n\times d_k},\ V \in \mathbb{R}^{n\times d_v}$$

Transformer and Attention Mechanism

Self-Attention Mechanism

- Query (Q): Represents the query vector, which is used to calculate the attention scores. Intuitively, the query vector represents the token that we are interested in.
- Key (K): Represents the key vector, which is used to calculate the attention scores. The key vector helps to determine the importance of each token in the input sequence.
- Value (V): Represents the value vector, which is used to compute the weighted sum of the values. The value vector provides the information that is used to update the representation of the input sequence.

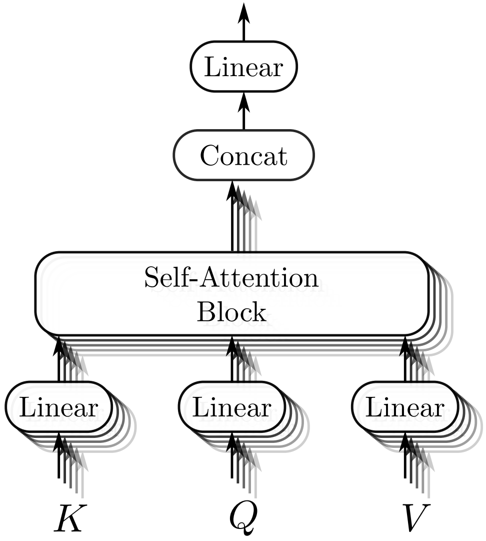
The resulting weighted sum of the values represents the output of the self-attention mechanism, capturing the relationships between different parts of the input sequence.



Transformer and Attention Mechanism

Multi-Head Attention

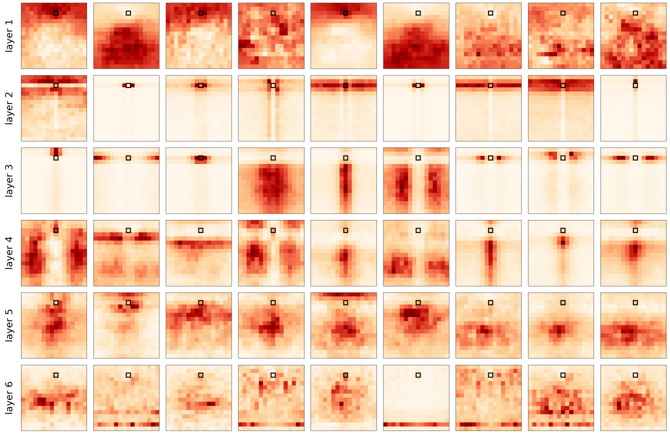
In Multi-head Attention, the self-attention mechanism is performed multiple times in parallel with different weight matrices, before being concatenated and once again projected, leading to a more robust representation of the input sequence. The intuition behind the three matrices (Q, K, V) remains the 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



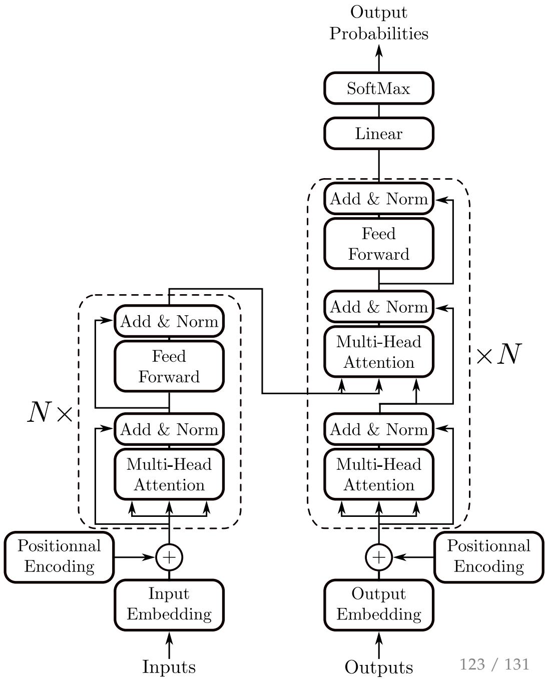
Transformer and Attention Mechanism

Transformers Model

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.



Transformer and Attention Mechanism

Transformers Model

More information about transformers and specific model architectures will be covered next semester in the course on Applied Deep Learning.

TP2: 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:

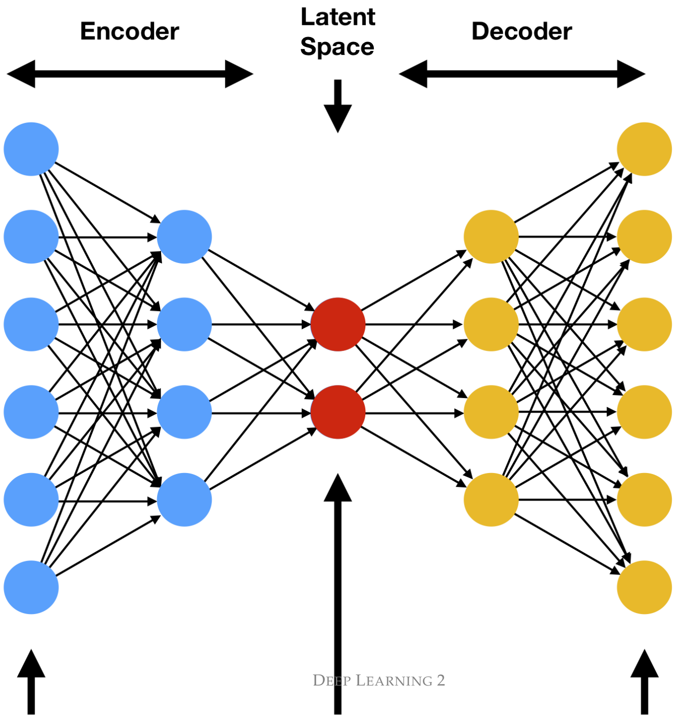
- An encoder function: $\textit{encoder}(x) : \mathbb{R}^d \to \mathbb{R}^m$
  Maps an input x from the input space $\mathbb{R}^d$ to a hidden representation space $\mathbb{R}^m$.
- A decoder function: $\textit{decoder}(z) : \mathbb{R}^m \to \mathbb{R}^d$
  Maps the hidden representation z back to the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

TP2: Build and use an autoencoder

Formal introduction of an autoencoder



TP2: Build and use an autoencoder

Autoencoders and Unsupervised Learning

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, θ represents the parameters of the encoder and decoder, and l is a loss function.

TP2: Build and use an autoencoder

Autoencoders and Unsupervised Learning

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:

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

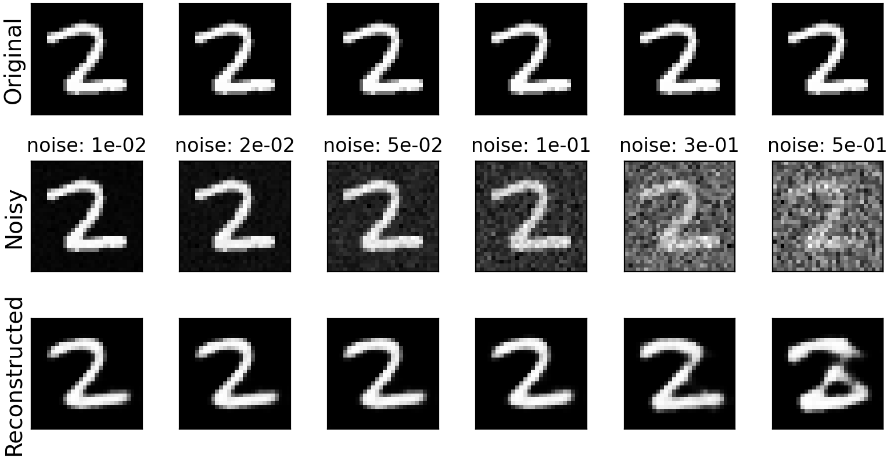
where x is the input data, θ represents the parameters of the encoder and decoder and $\hat{x} = \text{decoder}_\theta(\text{encoder}_\theta(x))$ in the reconstruction.

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.

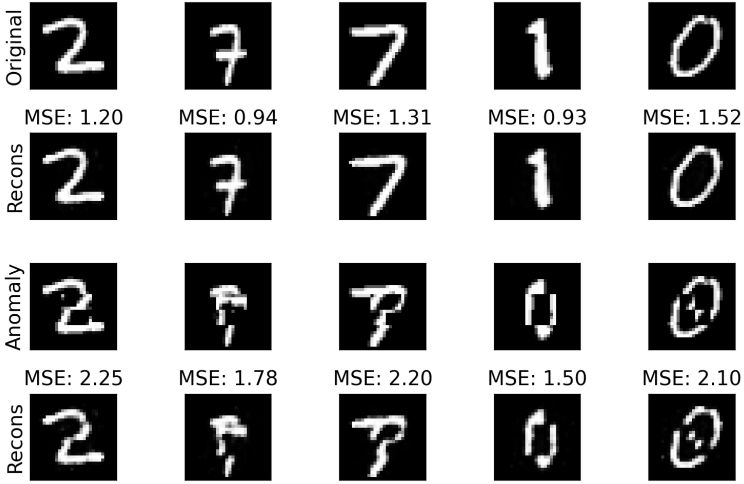
Applications of Autoencoders

- **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.



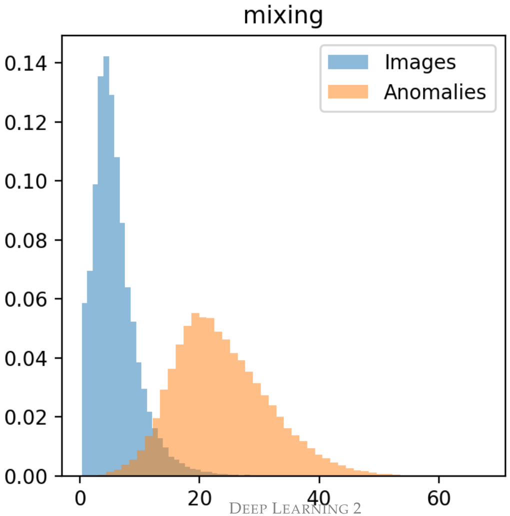
APPLICATIONS OF AUTOENCODERS

- **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.



TP2: Build and use an autoencoder

Your turn !

Get the TP2 on the course website and start working on it.

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

References I

[1] Brock, A., Donahue, J., and Simonyan, K. (2019). Large Scale GAN Training for High Fidelity Natural Image Synthesis. arXiv:1809.11096 [cs, stat].

[2] Bronnec, F. L., Verine, A., Negrevergne, B., Chevaleyre, Y., and Allauzen, A. (2024). Exploring Precision and Recall to assess the quality and diversity of LLMs. *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. arXiv:2402.10693 [cs].

[3] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative Adversarial Networks. In *27th Conference on Neural Information Processing Systems (NeurIPS 2014)*. arXiv: 1406.2661.

[4] Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. (2020). Analyzing and Improving the Image Quality of StyleGAN. arXiv:1912.04958 [cs, eess, stat].

[5] Kingma, D. P. and Dhariwal, P. (2018). Glow: Generative Flow with Invertible 1x1 Convolutions. In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.*, volume 31.

[6] Kynkäänniemi, T., Karras, T., Laine, S., Lehtinen, J., and Aila, T. (2019). Improved Precision and Recall Metric for Assessing Generative Models. In *33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.* arXiv: 1904.06991.

[7] Yann LeCun, Corinna Cortes, and Burges, C. (2010). MNIST handwritten digit database. *ATT Labs*, 2.