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

November 26, 2024

Semester Schedule (Temporary)

- 17/09: Fundamentals of Deep Learning
- 24/09: TP1 Classification - *Introduction to PyTorch*
- 01/10: In a Deep Learning Model + Techniques to Improve Deep Learning Training
- 08/10: TP2 Autoencoders - *Hyperparameter Tuning*
- 15/10: Advanced Deep Learning Techniques
- 22/10: TP3 Image Segmentation - *From CPU to GPU and Parallelization*
- 29/10: **No Class**

Semester Schedule (Temporary)

- 05/11: **Graded Individual Practical Work**
- 12/11: TP4 Deep Reinforcement Learning - *From Notebook to Script*
- 19/11: TP5 Adversarial Attacks - *Importance of Git*
- 26/11: Project Presentation - Group Formation
- 03/12: Group Session - Help with Projects
- 10/12: **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

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, ...

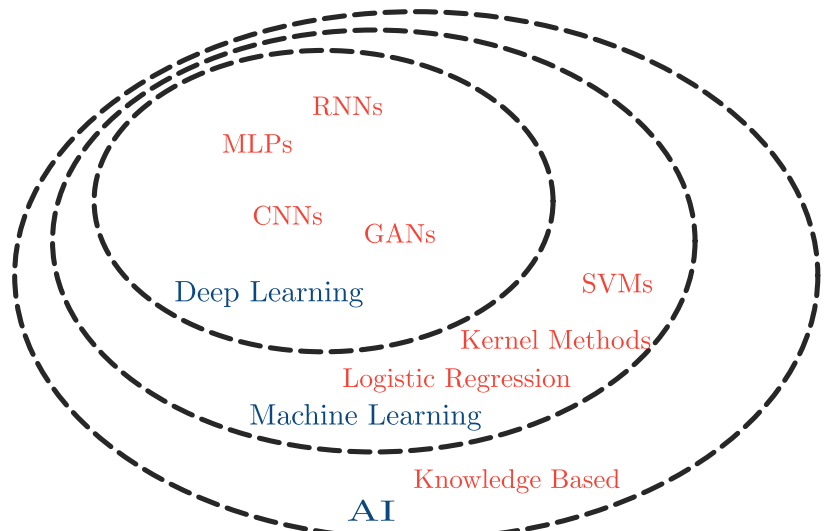
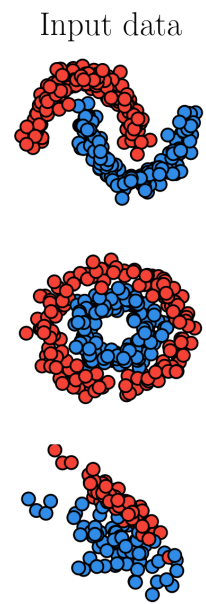


Figure. Subsets of Artificial Intelligence

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.



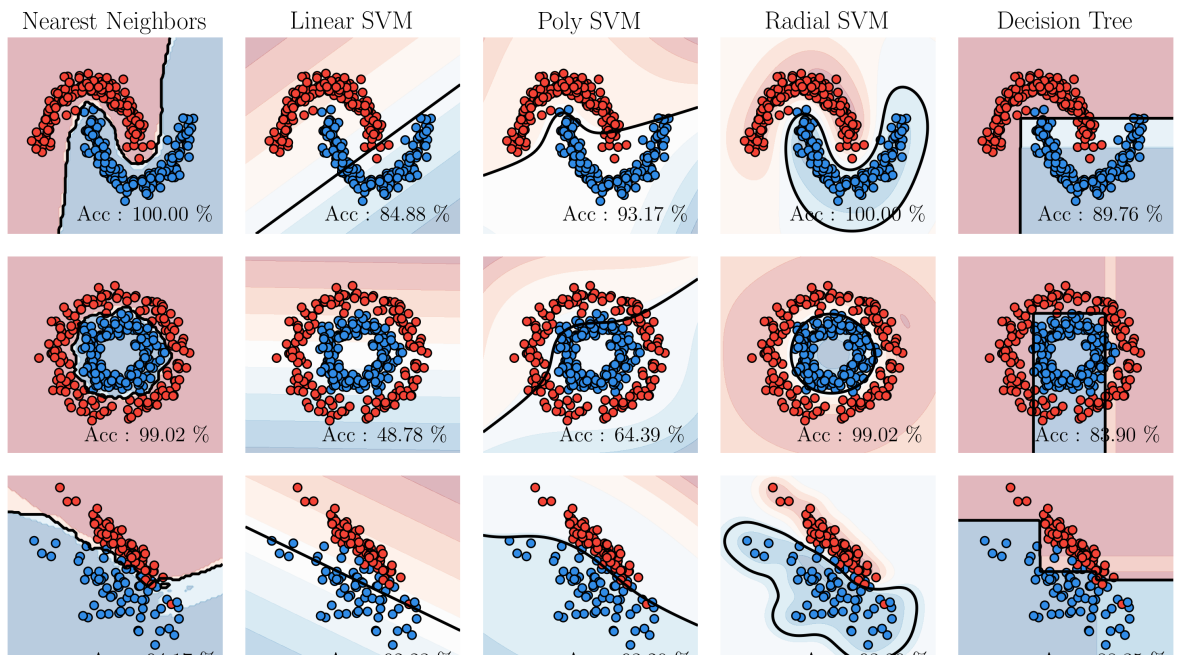


Figure. 2D classification for different AI models.

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

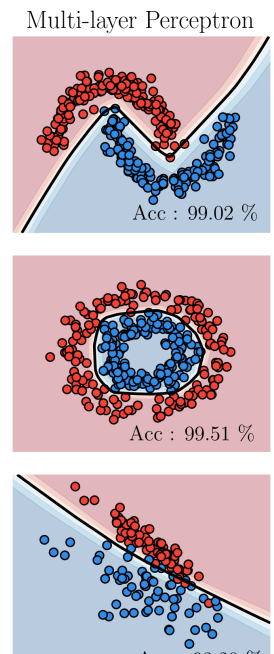
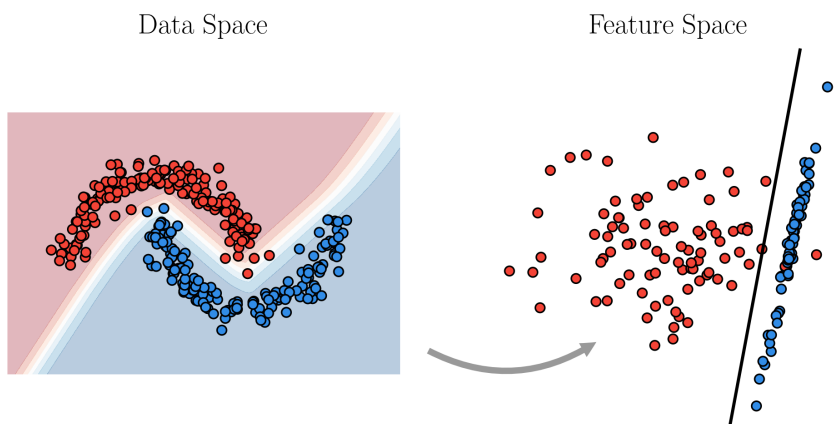


Figure. 2D classification for small Neural Network.

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.

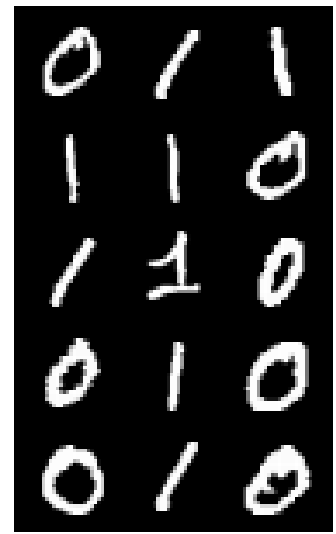


Figure. MNIST

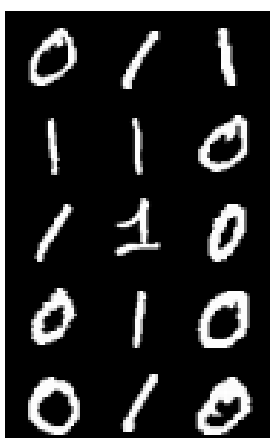


Figure. MNIST : Layer 0

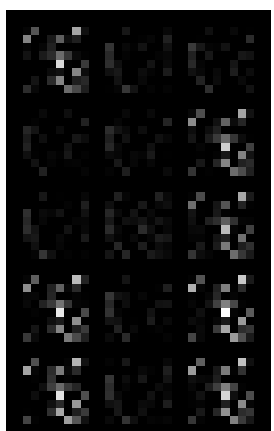


Figure. MNIST : Layer 1

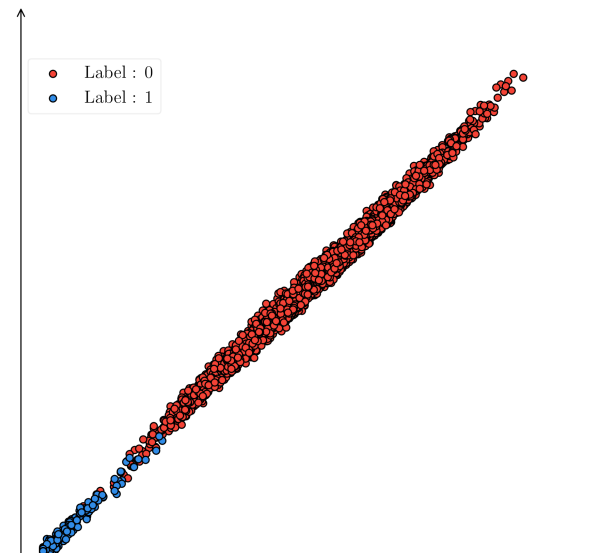


Figure. MNIST : Layer 2

Figure. MNIST : Layer 0

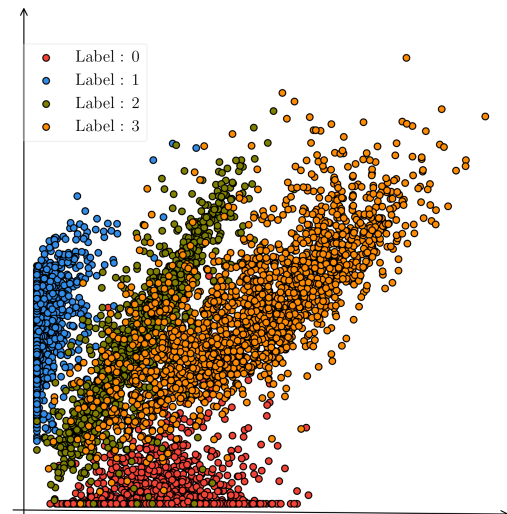


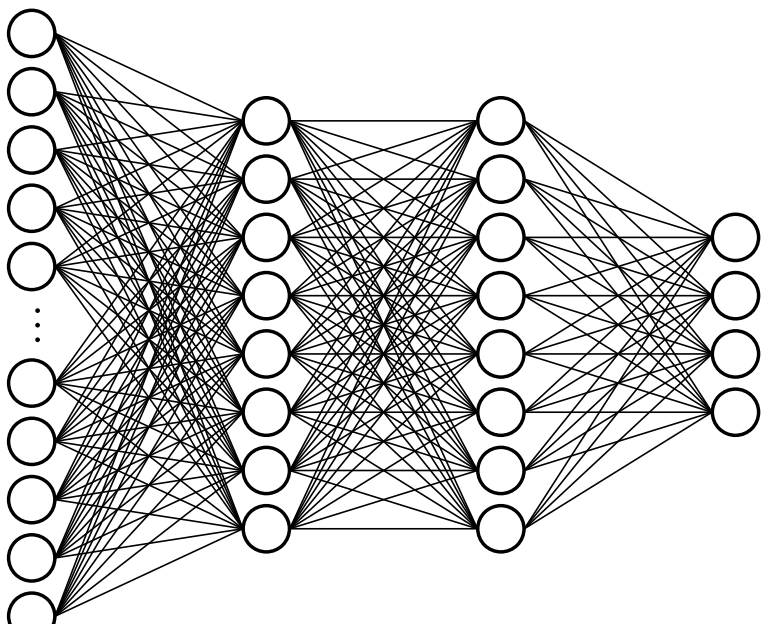
Figure. MNIST : Layer 2

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 ?

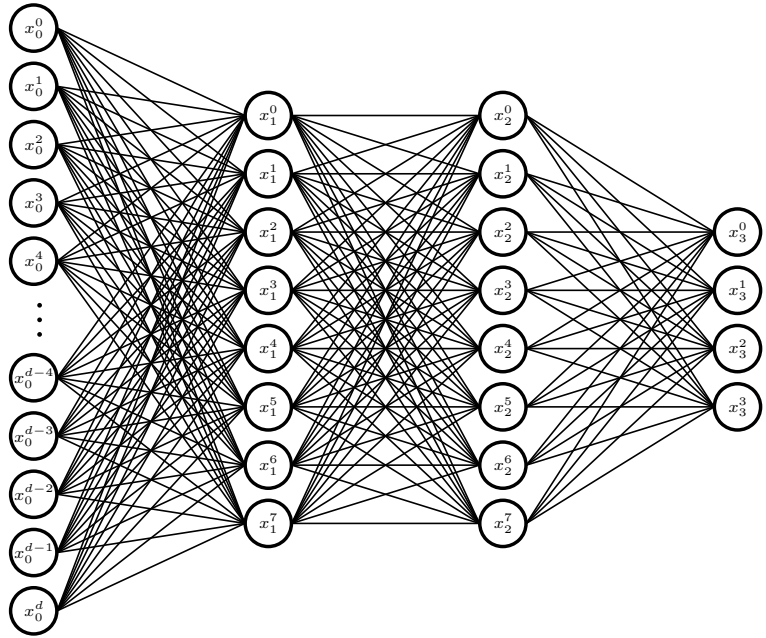
Neural Networks Fundamentals

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



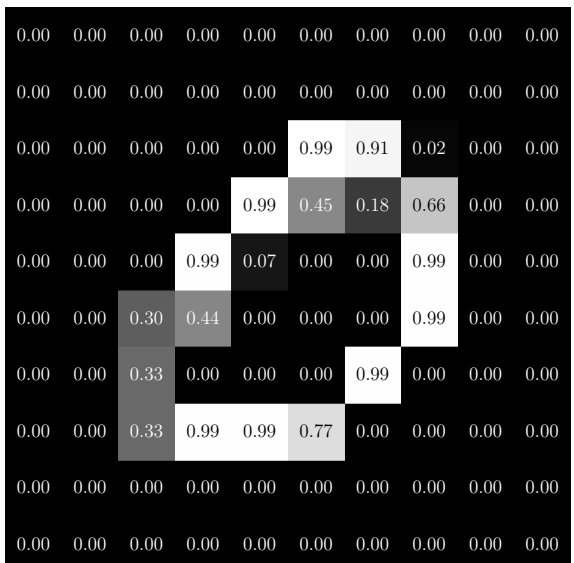
Let's see what the interconnected nodes, the layers and the transformations are.

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:



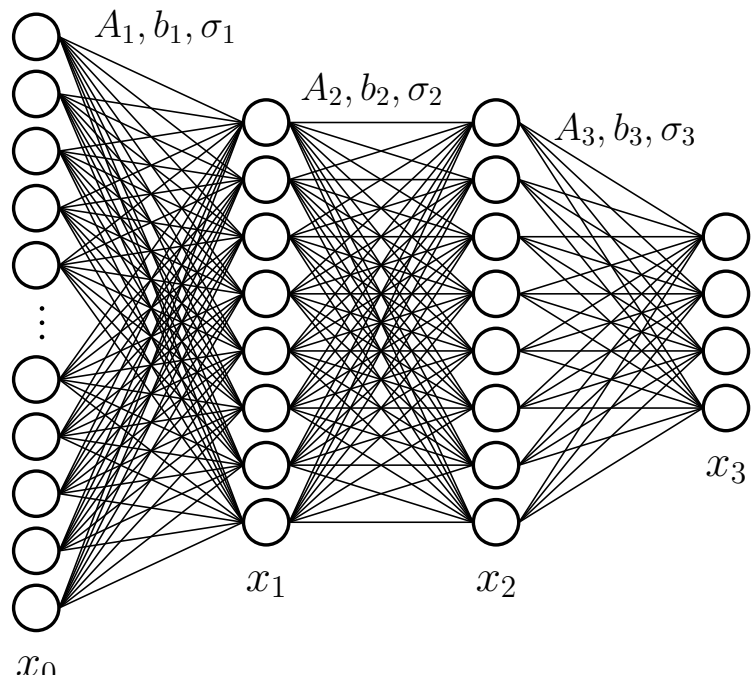
$\in [0,1]^{d/2 \times d/2}$

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

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:

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

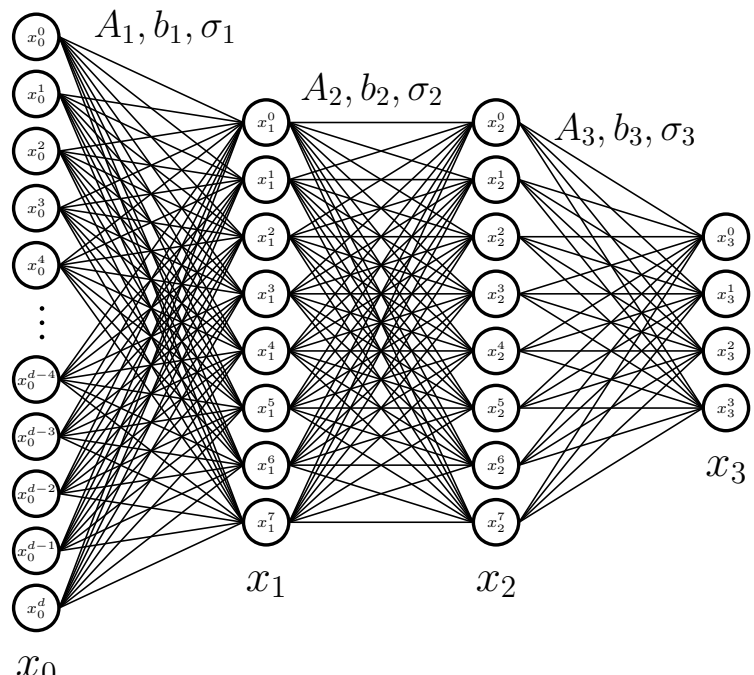
The non-linear function σ_i the activation function.



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.



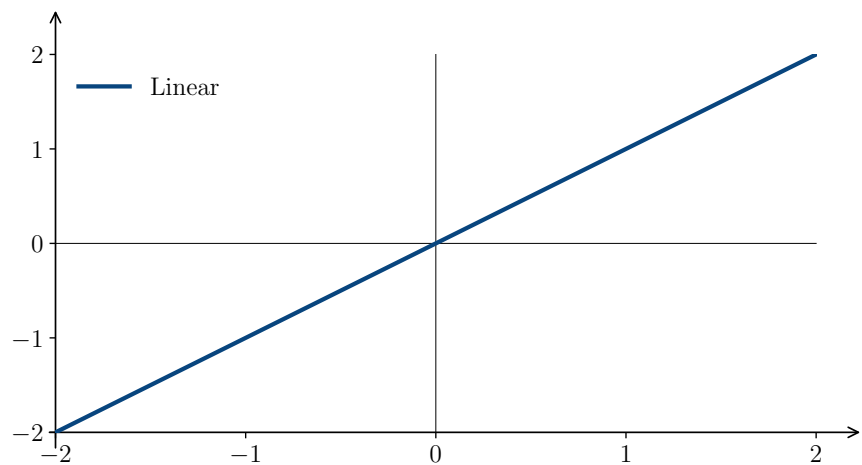
The activation functions play a crucial role in the implementation of deep neural networks, as they allow them to approximate any continuous function, as stated by the Universal Approximation Theorem. We can list some activation function that are commonly used :

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

- Linear activation Function:

$$\sigma(x) = x$$

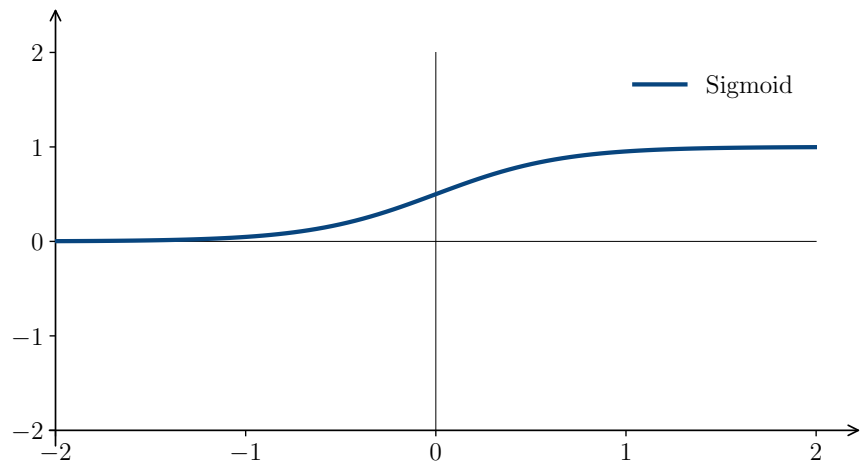
- Final activation
- Use case : Regression



- Sigmoid Function:

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

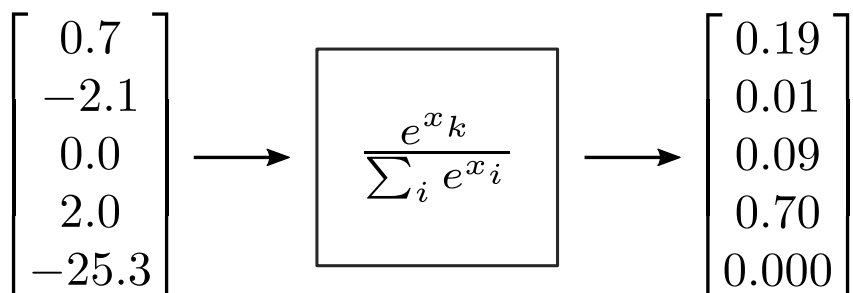
- Final activation
- Use case : Classification



- Softmax Function:

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

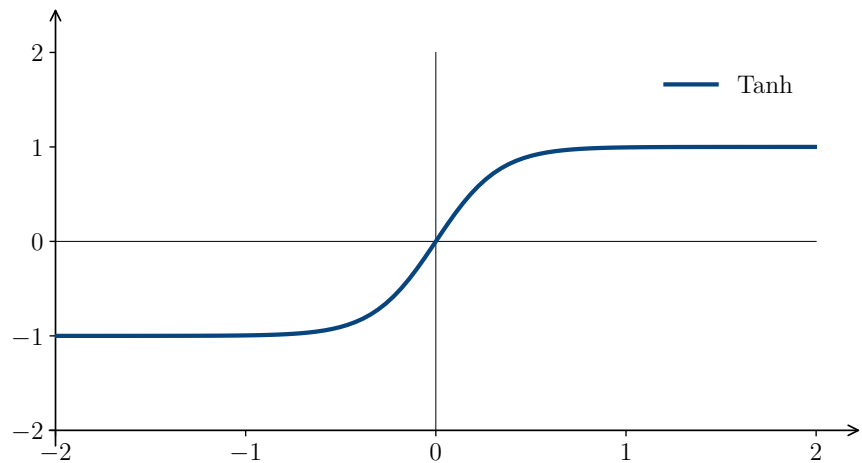
- Final activation
- Use case : Multi-class Classification



- Hyperbolic Tangent

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

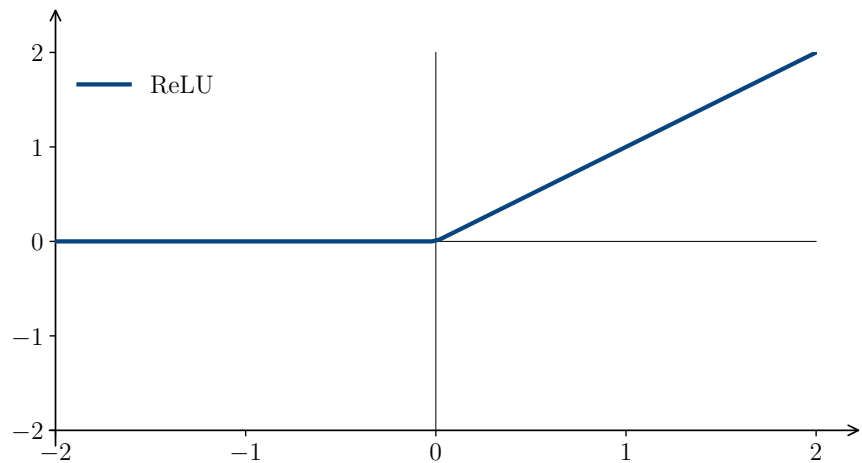
- Final activation
- Use case : Generative task



- Rectified Linear Unit (ReLU):

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

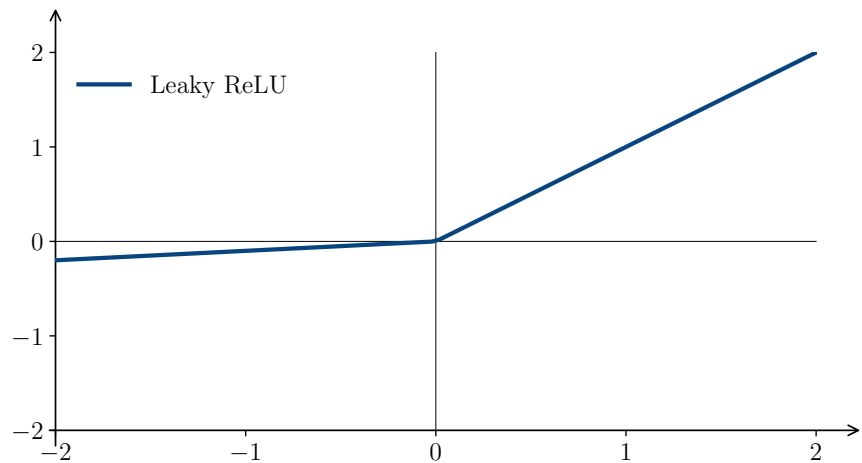
- Intermediate activation



- Leaky Rectified Linear Unit (Leaky ReLU):

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

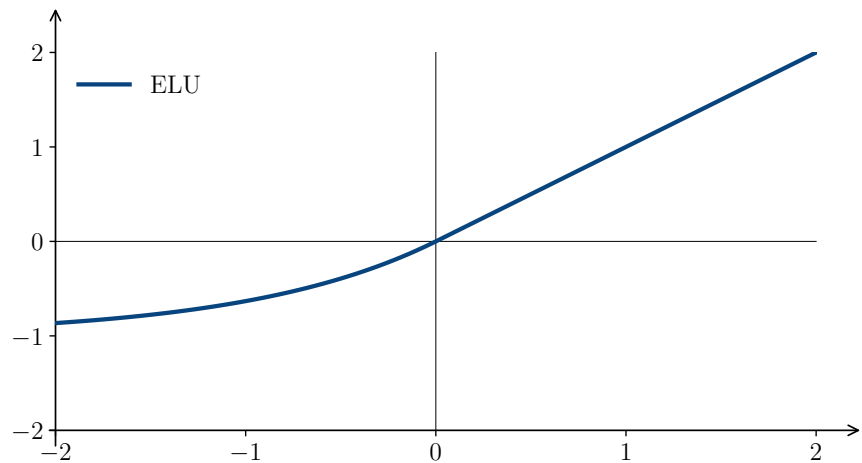
- Intermediate activation



- Exponential Linear Unit (ELU):

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

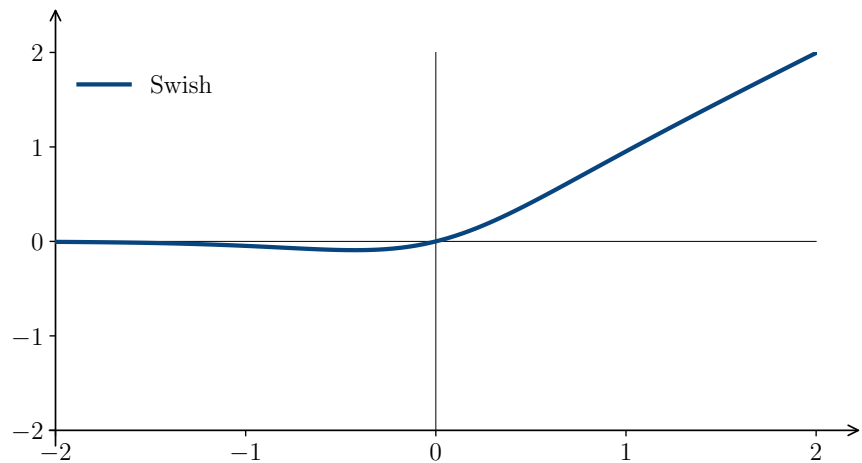
- Intermediate activation



- Sigmoid-Weighted Linear Unit (Swish):

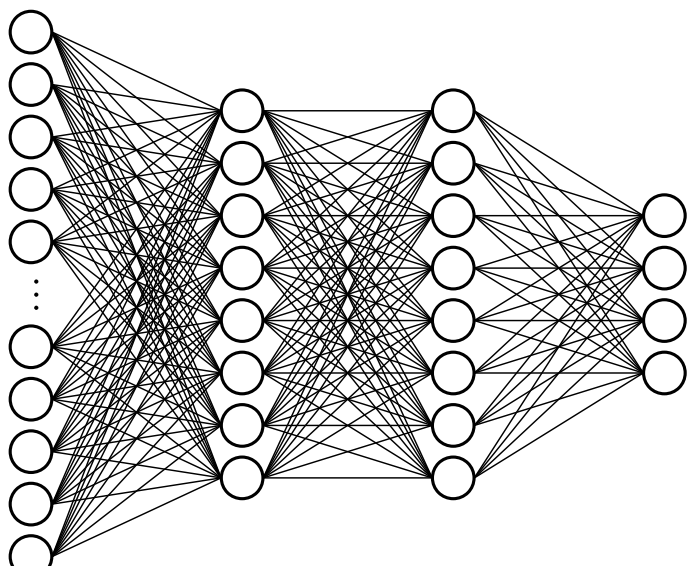
$$\sigma(x) = \frac{x}{1 + e^{-x}}$$

- 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.

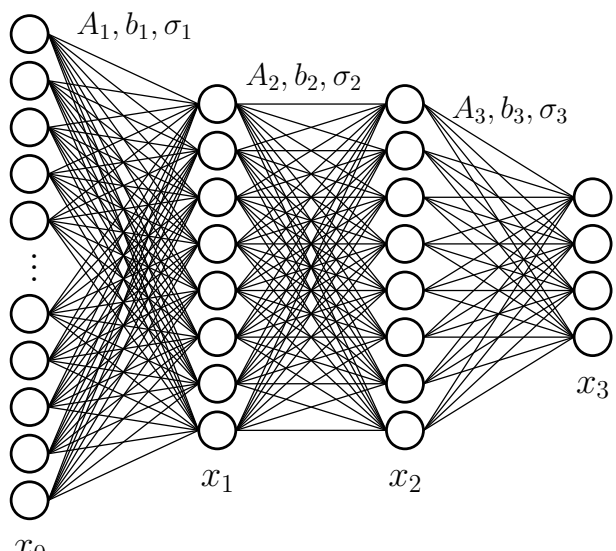


The Multi-layer Perceptron (MLP)

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 $\theta = \{A_1, A_2, A_3, b_1, b_2, b_3\}$:

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



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.

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

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 λ

Initialize parameters θ
while θ 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 θ: $\nabla_\theta l(x_i, f_\theta(x_i))$
    Update parameters $\theta = \theta - \lambda \nabla_\theta l(x_i, f(x_i))$
  end for
end while
return θ

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:

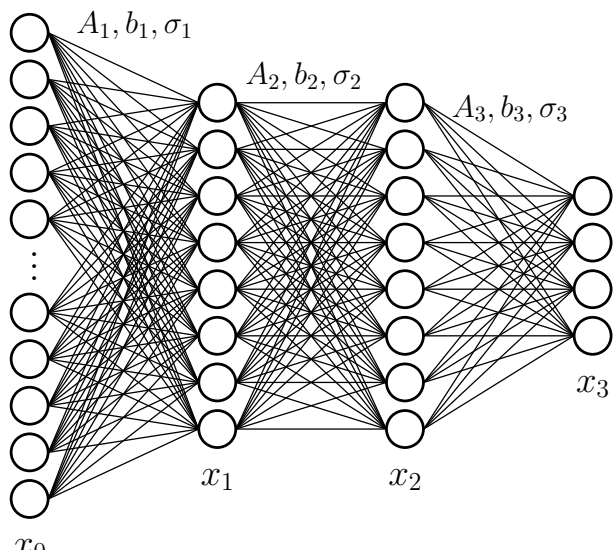
Require: Given a loss function l, a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_N\}$, a learning rate λ and a batch size b

Initialize parameters θ
Initialize the number of batches $B = \lfloor \frac{N}{b} \rfloor$
while θ 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 θ: $\frac{1}{B} \sum_{i=1}^{B} \nabla_\theta l(x_i, f_\theta(x_i))$
    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 θ

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.



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.

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:

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

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

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.

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 $l'(x_3)$,

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.

Let us compute the gradient with respect to x_1:

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

Therefore:

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

Back-Propagation

In other words, the update on the weights is:

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

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 :

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

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.

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.

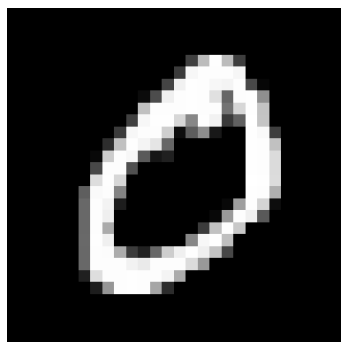
To update the weights, we need to compute the gradient of the loss with respect to the output of the network, and then **back-propagate** the gradient of the loss with respect to each activation, the $\frac{\partial l}{\partial x_i}$, through the network to compute the gradients with respect to the weights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

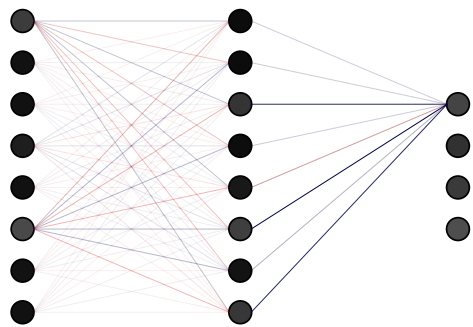
The Multi-layer Perceptron (MLP)

Last Layer

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

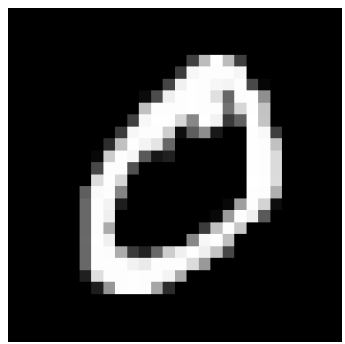
The red lines show positive values for A_i, the blue lines represent negative values for A_i. The level of transparency is proportional to the previous neurons.

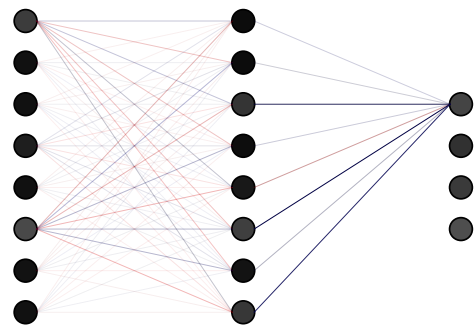




$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

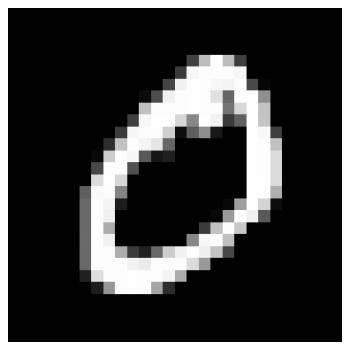
Iteratively, the neural networks improves its performance.

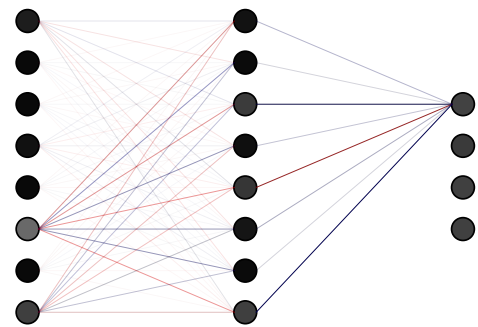




$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

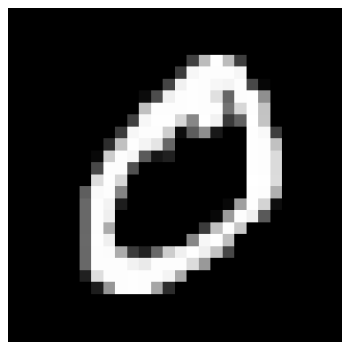
Iteratively, the neural networks improves its performance.

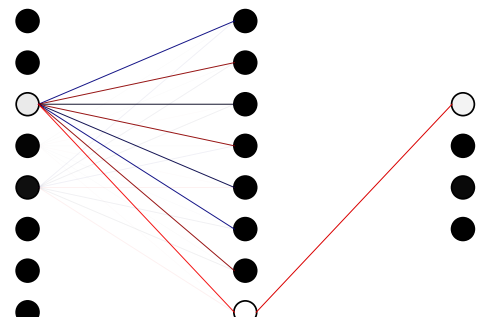




$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

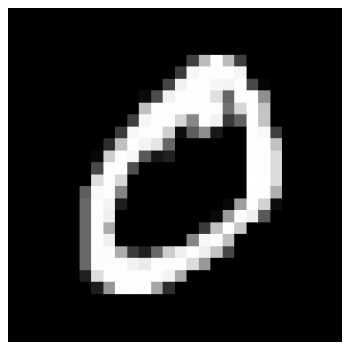
Iteratively, the neural networks improves its performance.

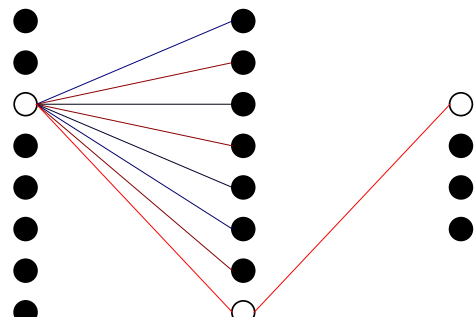




$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

Iteratively, the neural networks improves its performance.





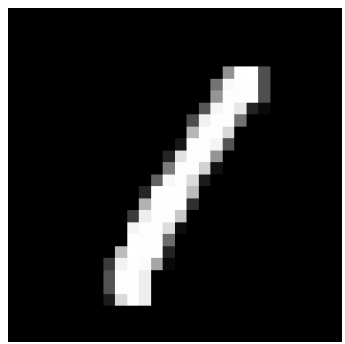
$$x_3^1 = \sigma_3\left(A_3^{1,1}x_2^1 + A_3^{1,2}x_2^2 + \cdots + A_3^{1,8}x_2^8\right)$$

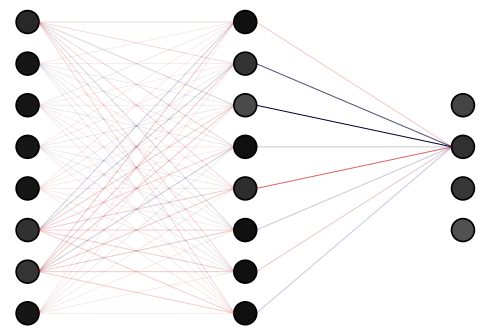
The Multi-layer Perceptron (MLP)

Last Layer

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

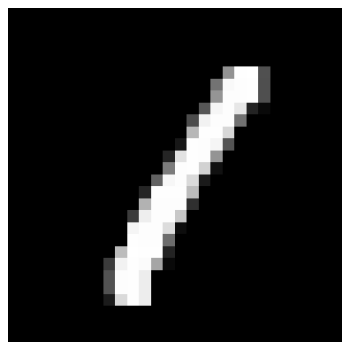
Red lines show positive values of A_i, Blue lines represent negative values of A_i. The level of transparency is proportional to the previous neurons.

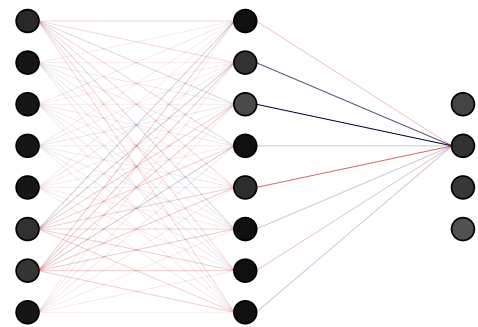




$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

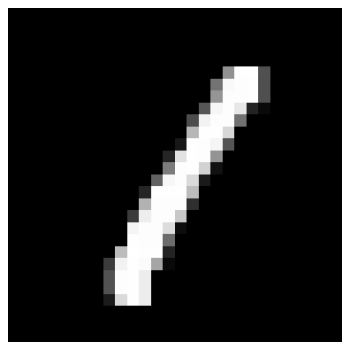
Iteratively, the neural networks improves its performance.

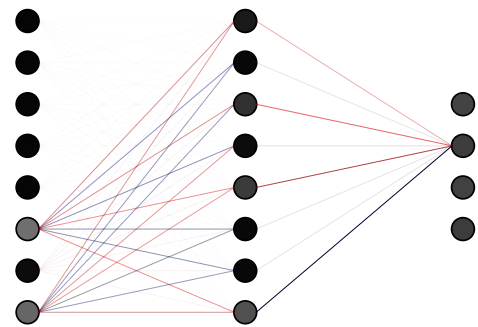




$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

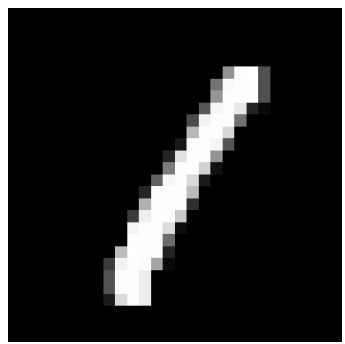
Iteratively, the neural networks improves its performance.

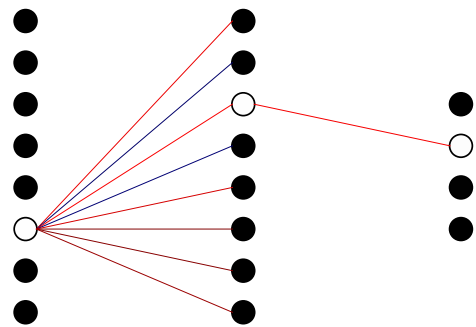




$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

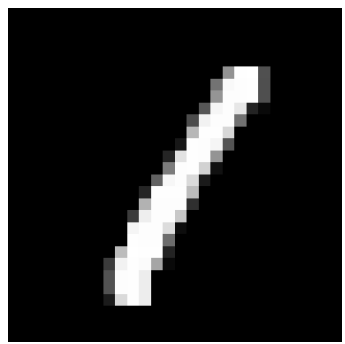
Iteratively, the neural networks improves its performance.

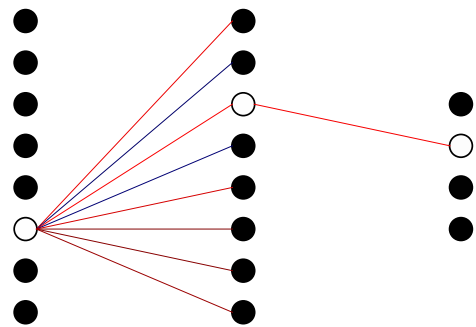




$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

Iteratively, the neural networks improves its performance.



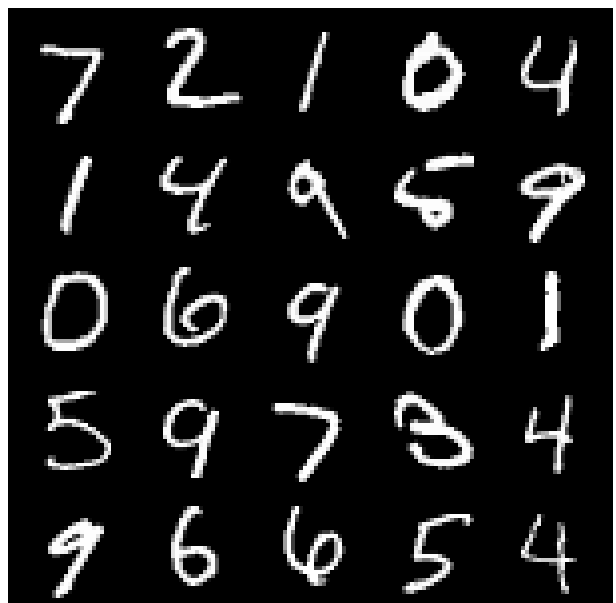


$$x_3^2 = \sigma_3\left(A_3^{2,1}x_2^1 + A_3^{2,2}x_2^2 + \cdots + A_3^{2,8}x_2^8\right)$$

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.



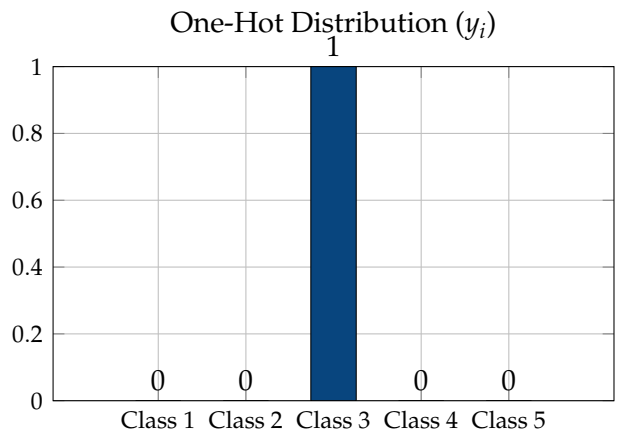
- Input shape : $1 \times 28 \times 28$.
- Number of Classes : 10.
- Number of training samples (x, y): 60000.
- Number of evaluating samples: 10000.
- Loss : cross-entropy

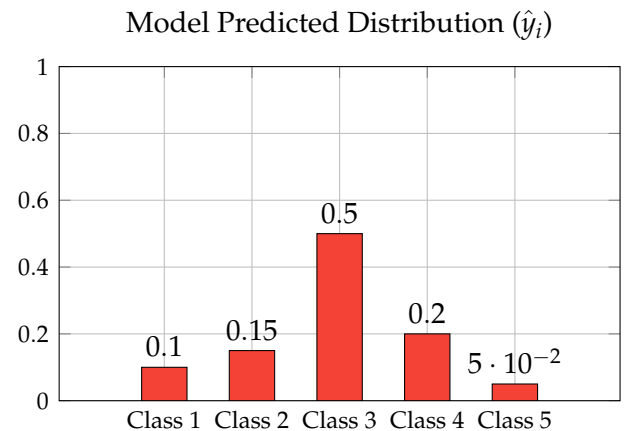
$$L(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij})$$

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:

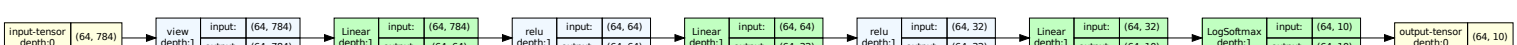
$$l(\hat{y}_i, y_i) = -\sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij}).$$

The Multi-layer Perceptron (MLP)

Example : Image classification of handwritten digits from A to Z

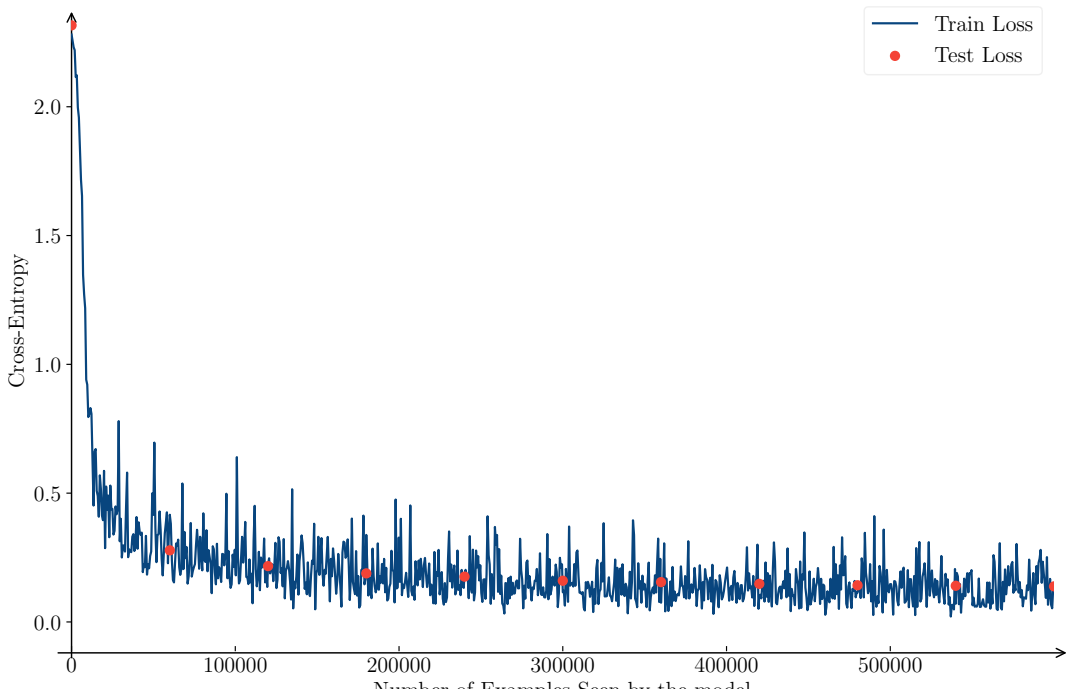
We build a 3 layers network.

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



The Multi-layer Perceptron (MLP)

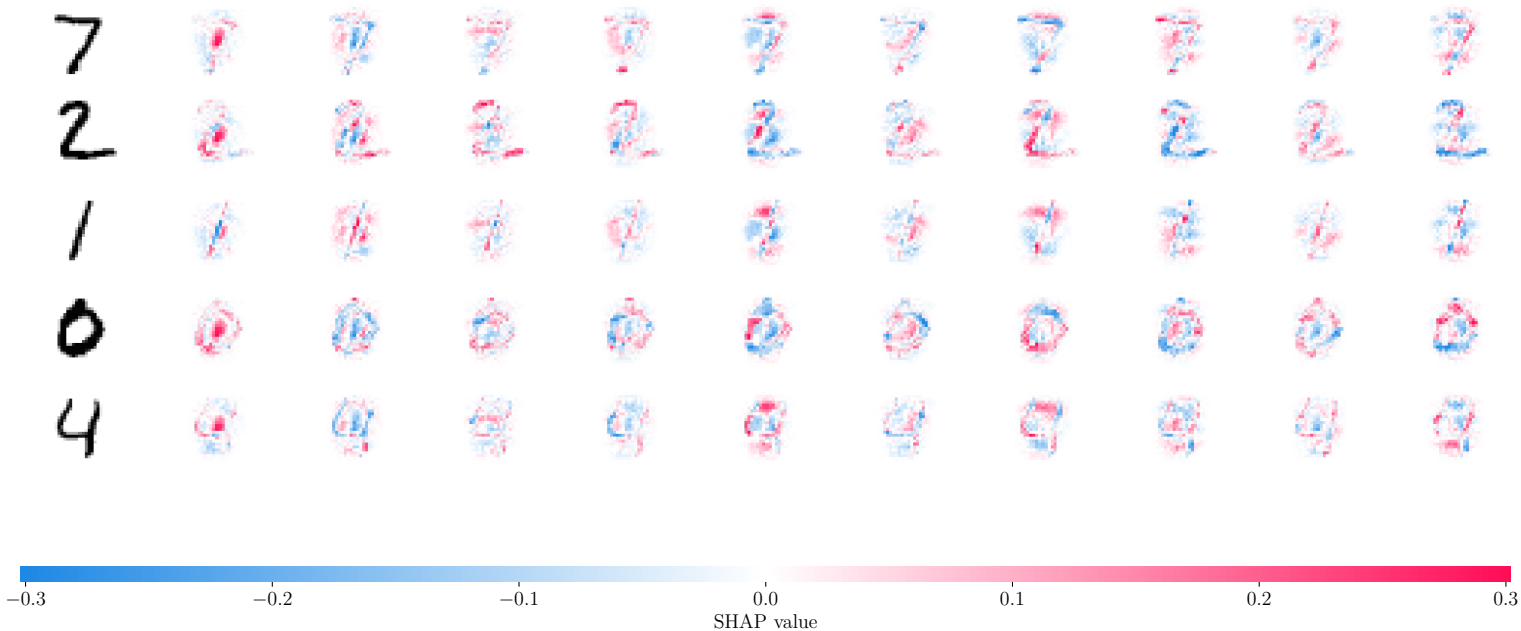
Example : Image classification of handwritten digits from A to Z



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

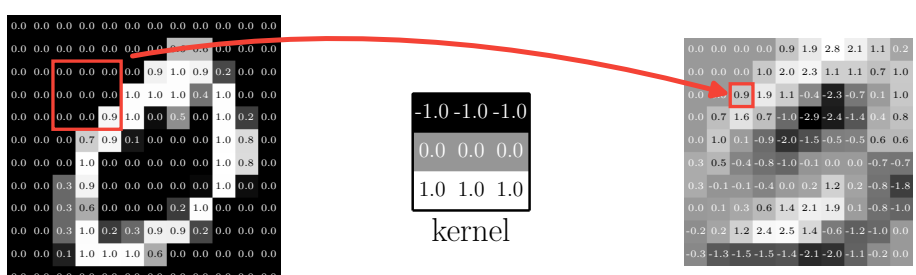
Now that we have an understanding of the training procedure for Artificial Neural Networks, we shall examine several widely-utilized structures within the literature of Neural Networks, including Convolutional Neural Networks (CNN),Resdiual Networks (ResNet), Recurrent Neural Networks (RNN), and Transformers.

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.

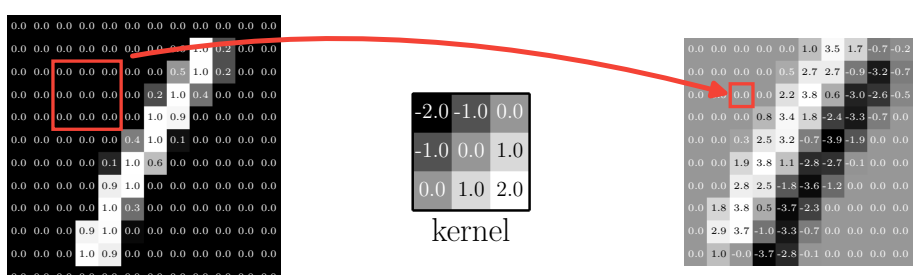
A 2D convolution in a neural network context can be mathematically represented as a sliding window operation where a filter (also called kernel) w of size $k \times k$ is applied to each $k \times k$ sub-matrix of the input matrix x. The operation can be defined as the element-wise multiplication of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

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



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.

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



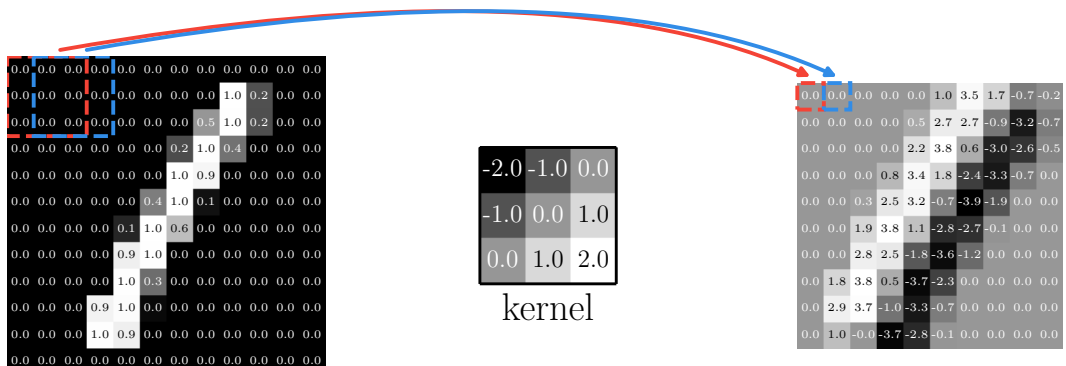
In every Deep Learning library, the Conv2D block takes three parameters in argument:

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

The size out the output is :

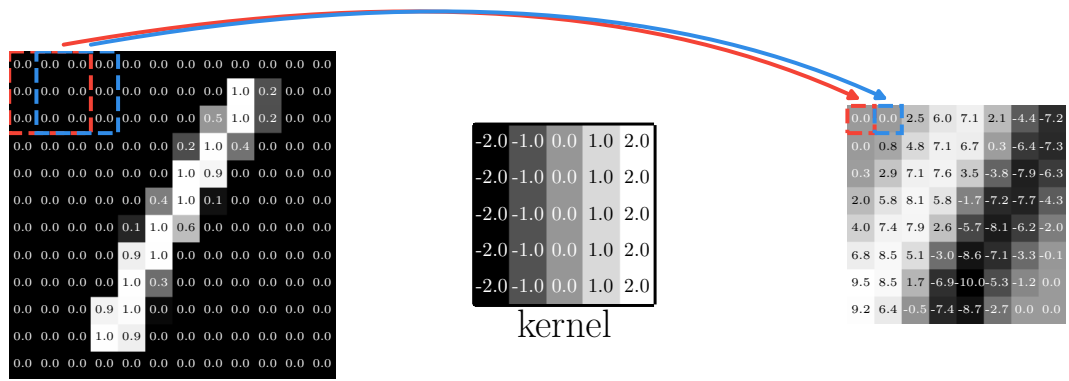
$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

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



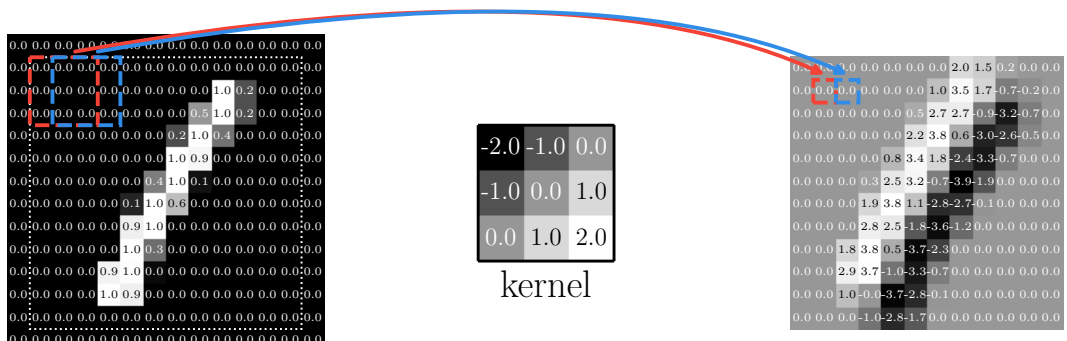
$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

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



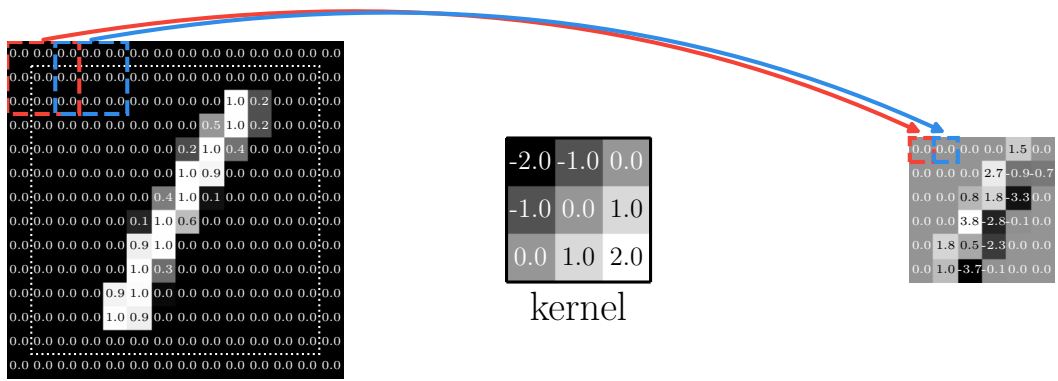
$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

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



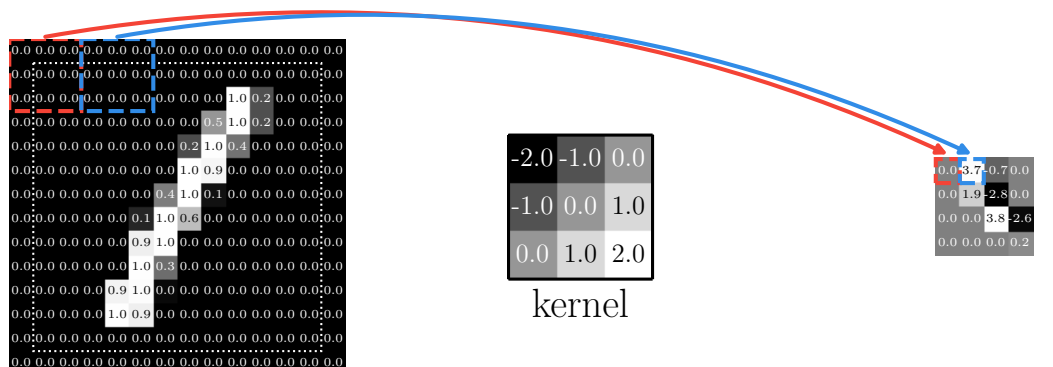
$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

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



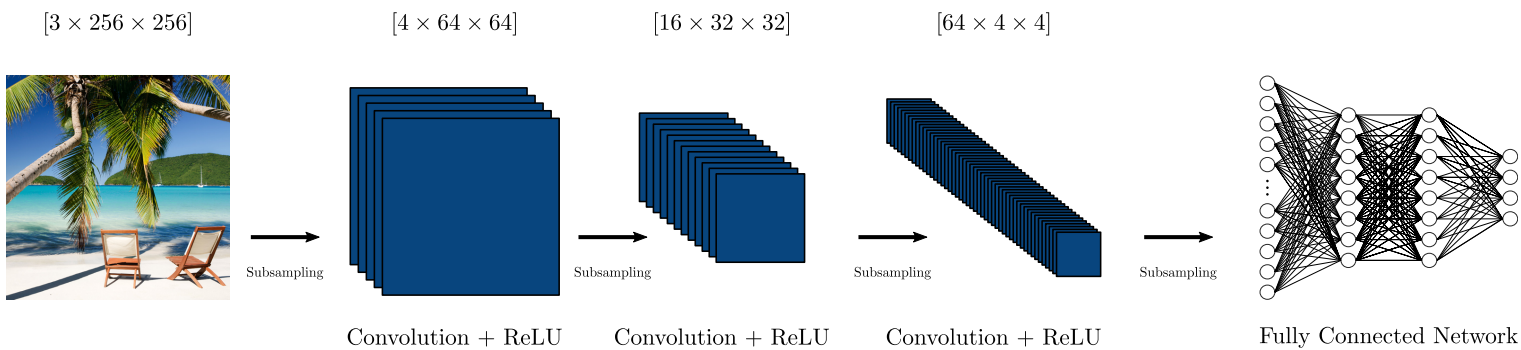
$$d_{\text{out}} = \frac{d_{\text{in}} + 2 \times \text{Padding} - \text{KernelSize}}{\text{Stride}} + 1$$

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



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

We can represent a CNN as under this form:

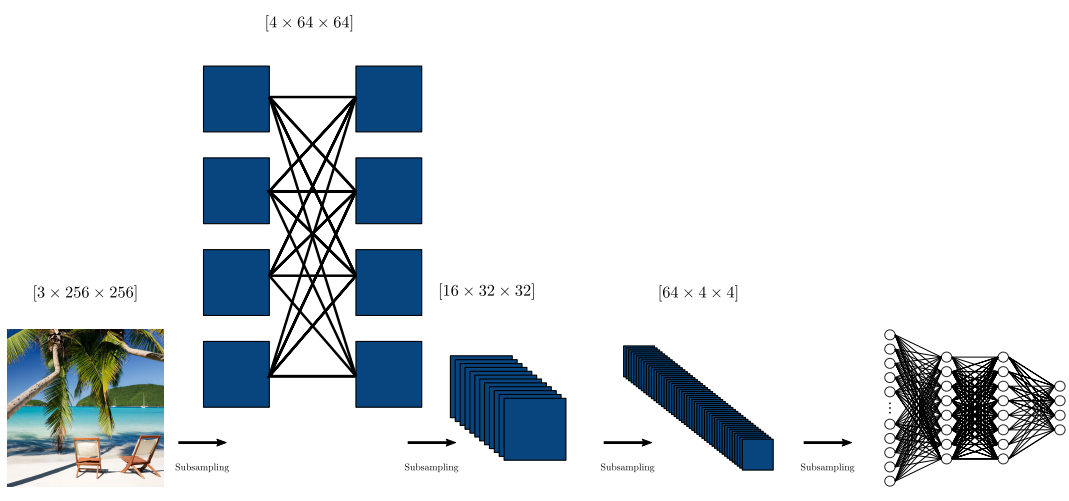


Convolutional Neural Networks

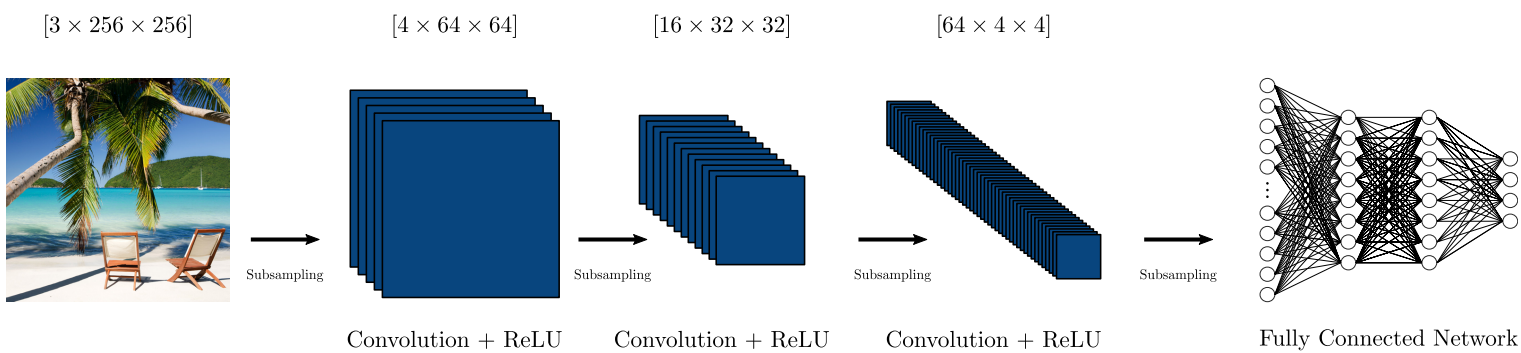
CNN : Convolutional in a network Networks

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

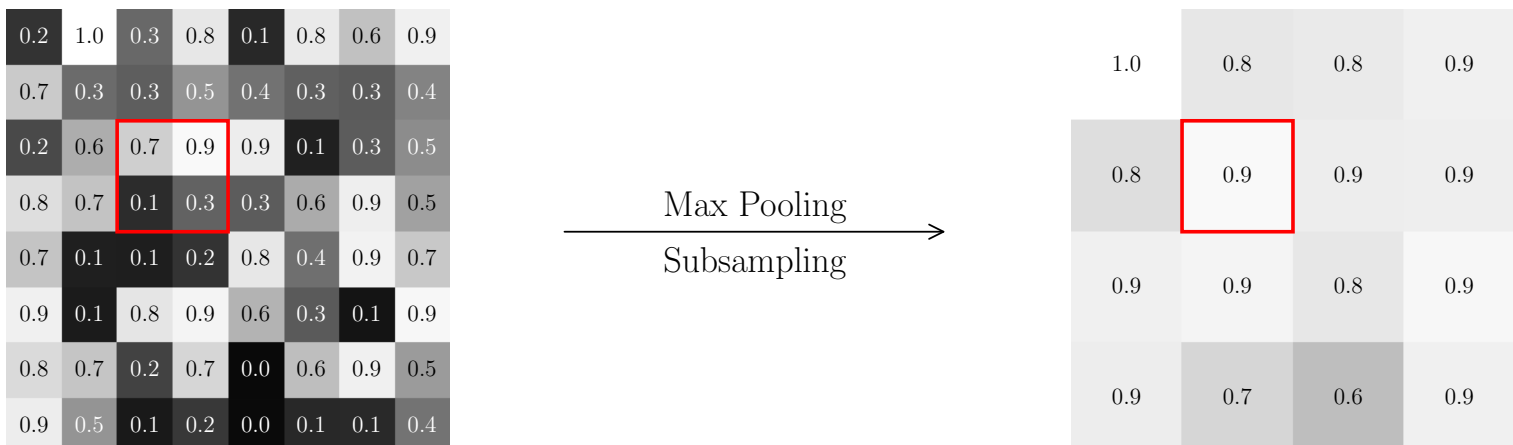
$$\text{out}_{i,j}(c_{\text{out}}) = \text{bias}(c_{\text{out}}) + \sum_{k=0}^{|c_{\text{in}}|-1} \text{Conv}(\text{input}(k), \text{kernel}_k)_{i,j}$$



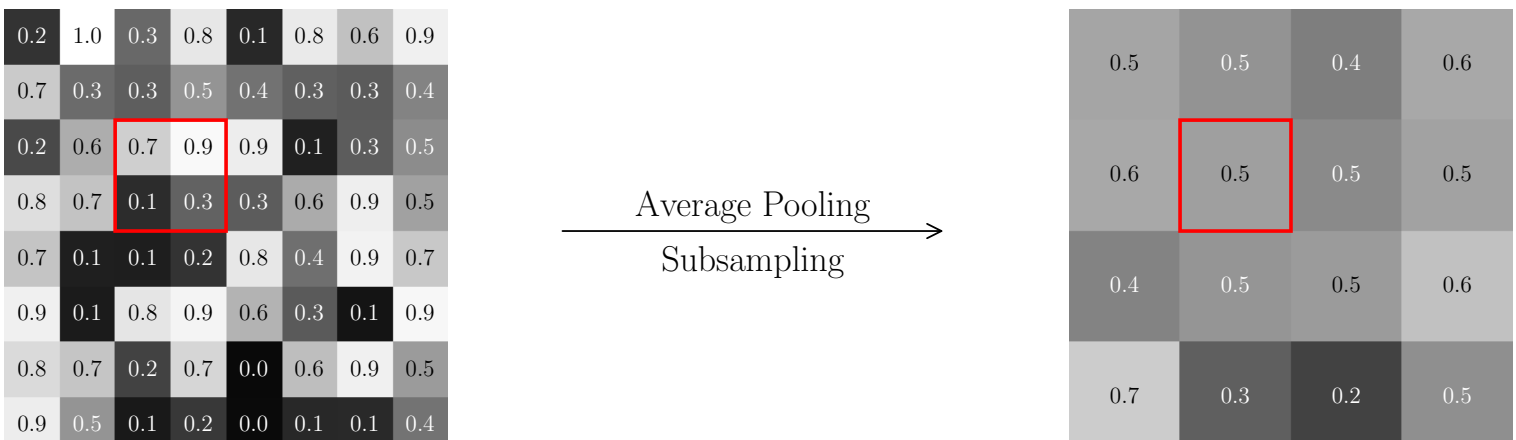
In practice, we split the image into multiple channels : the three channels RGB to begin with. Then we apply convolutional operation on different scales and then we use a fully connected tail. To change the scale we can use different sub-sampling : Max pooling, Average pooling or Invertible pooling.



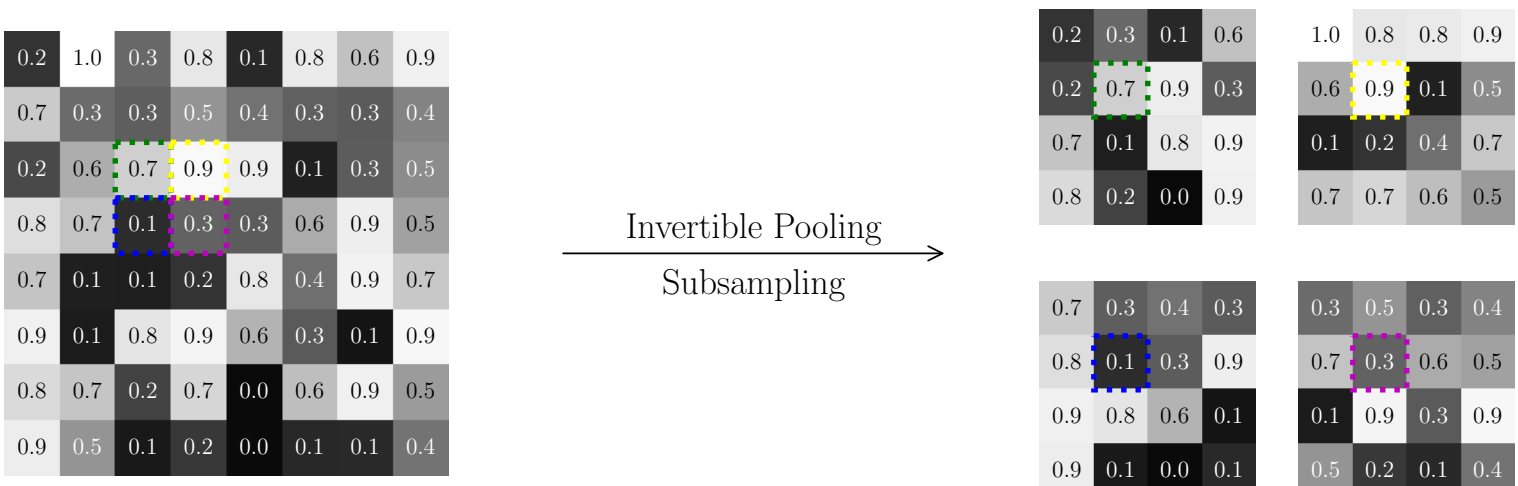
Max Pooling take the maximum within a given sized sub-matrix. In practice, the matrix is size 2×2 in order to reduce the dimension by 4 and doubling the scale.



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



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.

We will compare three different models:

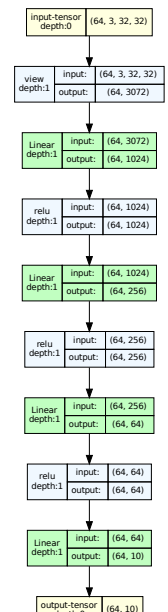
- Model 1 : Fully Connected Neural Network with 3.4 million parameters.
- Model 2 : CNN with 62 thousand parameters.
- Model 3 : Wider and longer CNN with 5.8 million parameters.

Convolutional Neural Networks

Model 1

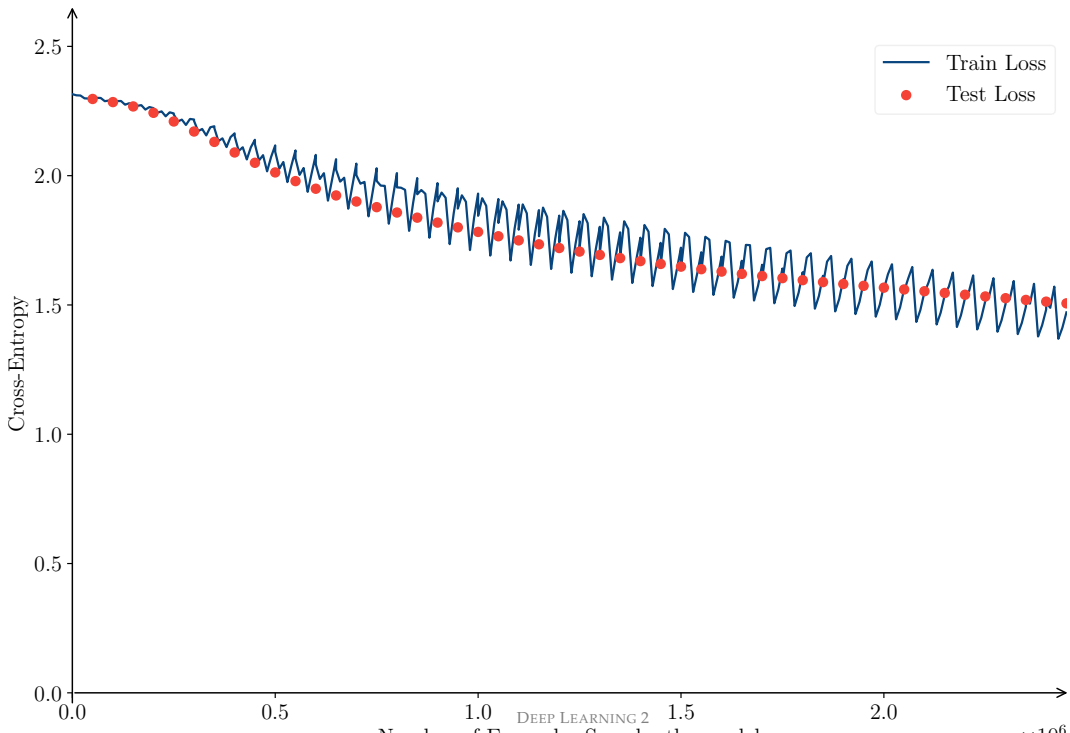
The Net in composed of 4 linear layers with ReLU activations:

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



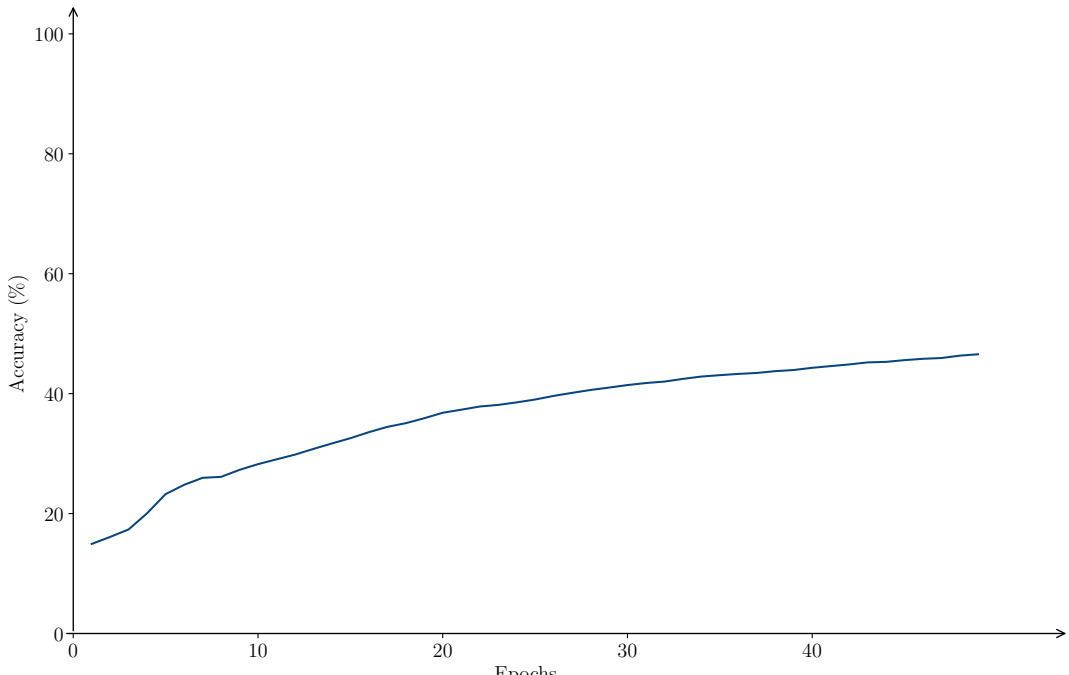
Convolutional Neural Networks

Model 1



Convolutional Neural Networks

Model 1

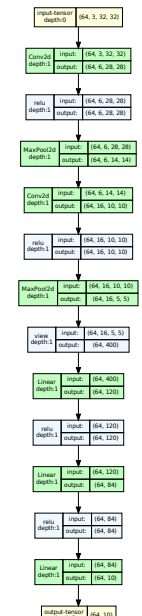


Convolutional Neural Networks

Model 2

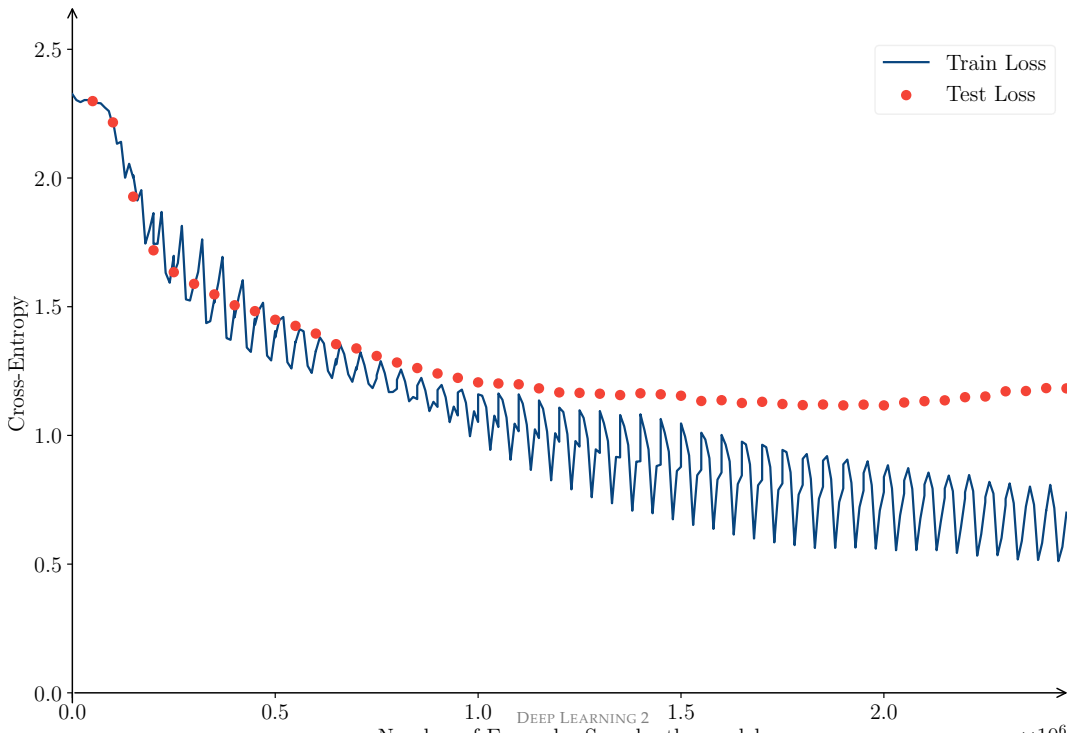
The Net is composed 2 convolutional layers and 2 linear layers:

- 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



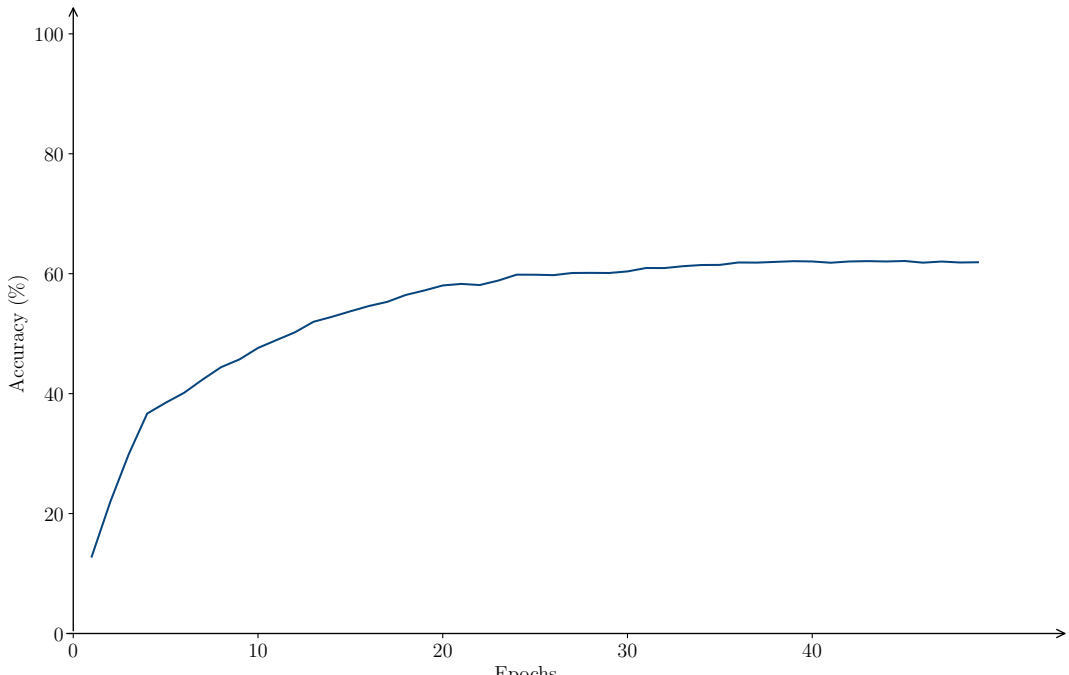
Convolutional Neural Networks

Model 2



Convolutional Neural Networks

Model 2



Convolutional Neural Networks

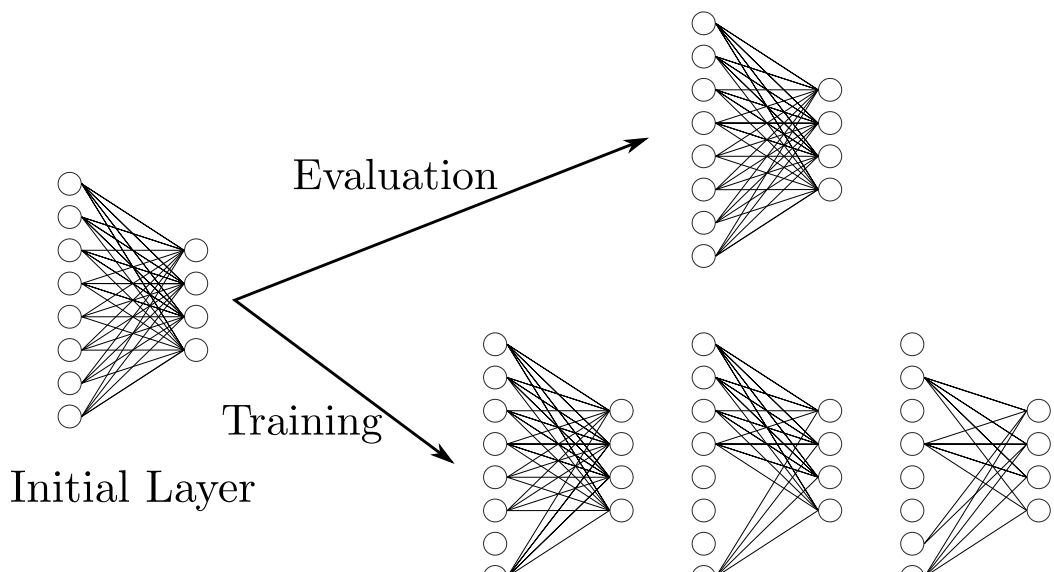
Model 3

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.

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 μ_B and variance σ_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 μ_B and σ_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 γ and β respectively.

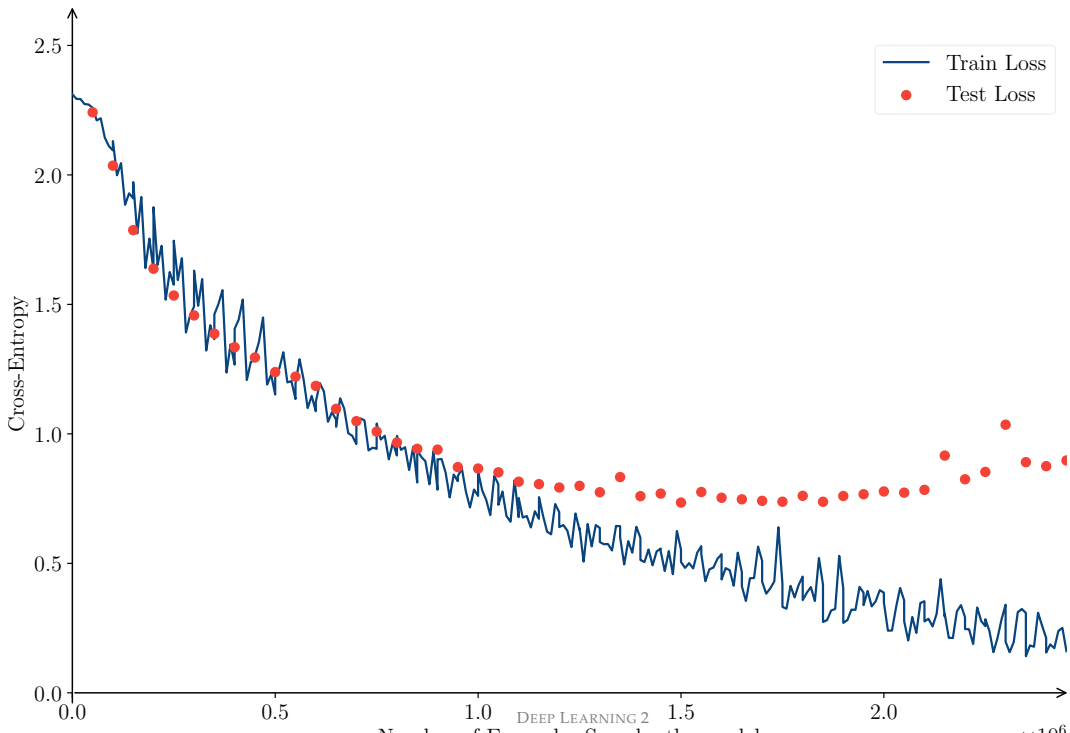
$$y_i = \gamma \bar{x}_i + \beta$$

5: **end for**

Algorithm 1: Batch Normalization

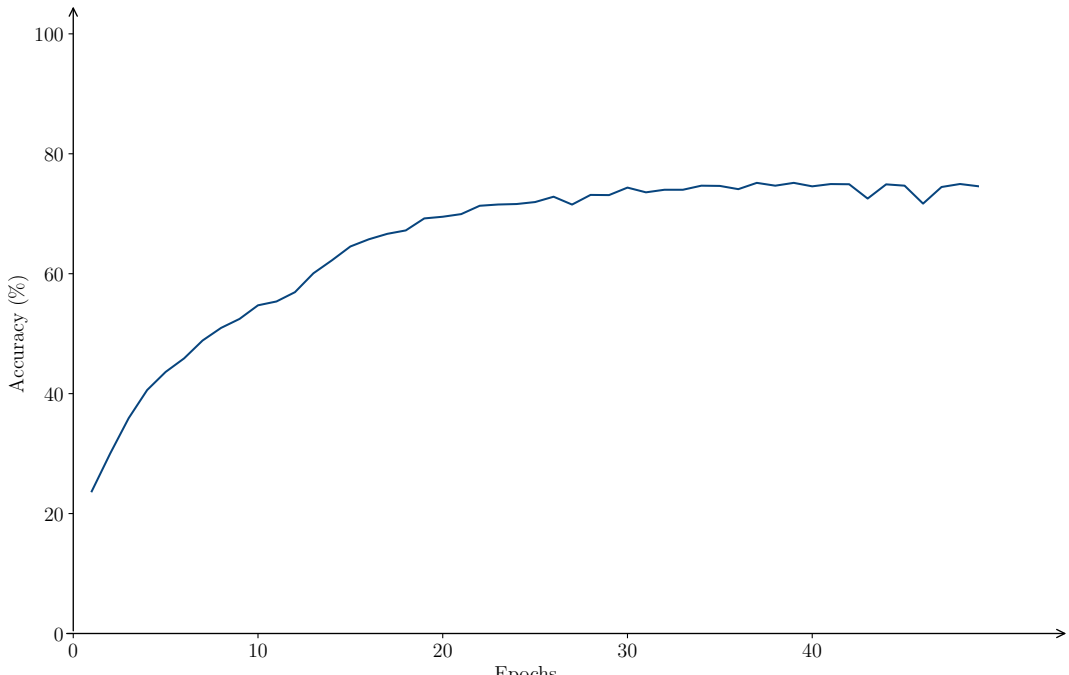
Convolutional Neural Networks

Model 3



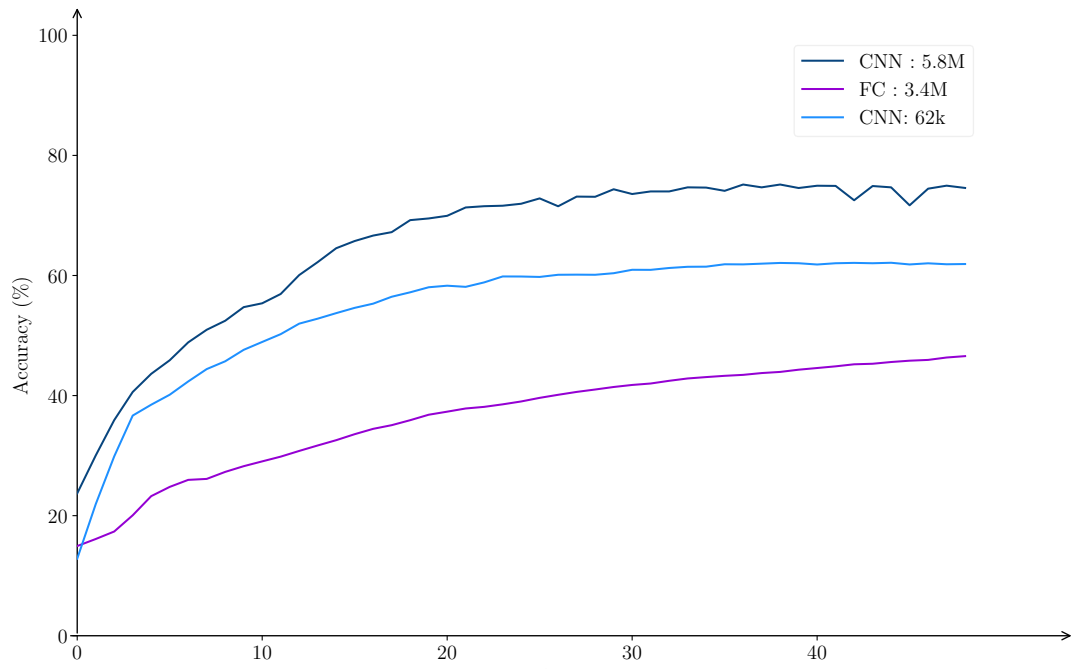
Convolutional Neural Networks

Model 3



Convolutional Neural Networks

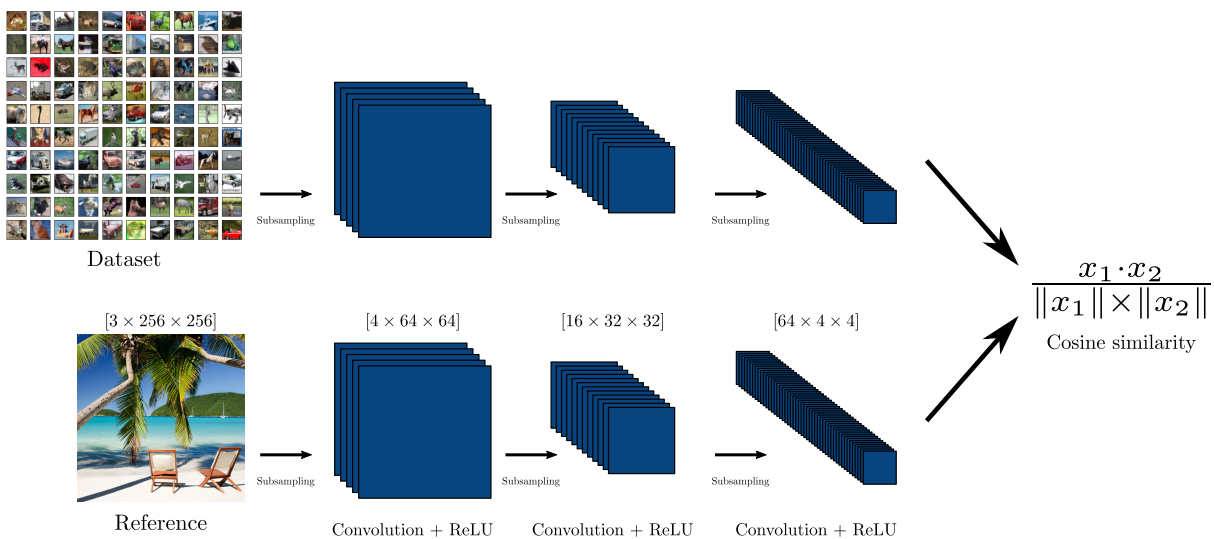
CNN in practice: CIFAR 10



Convolutional Neural Networks

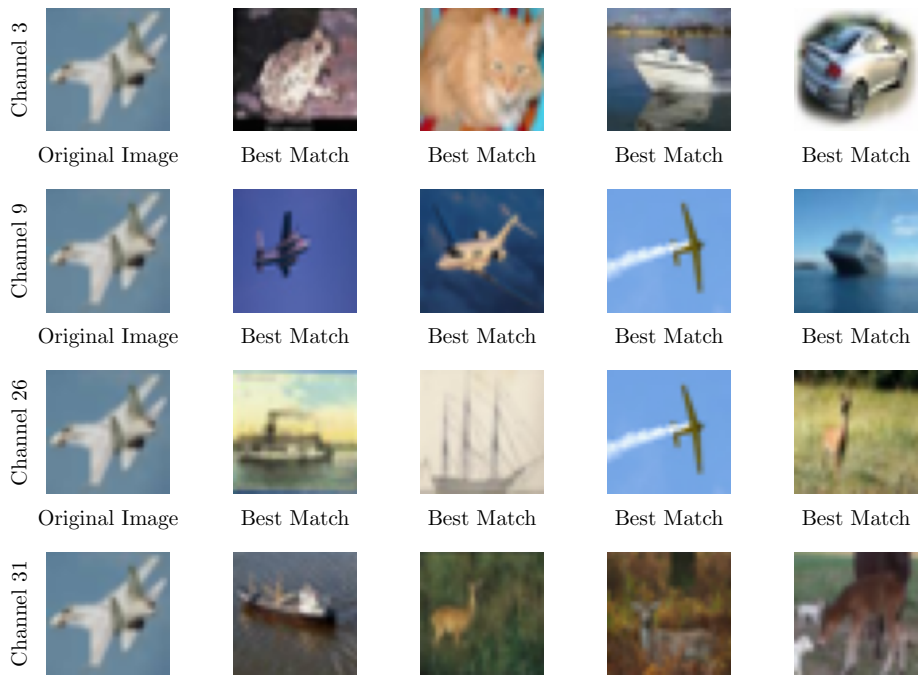
Intuition behind channels

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.



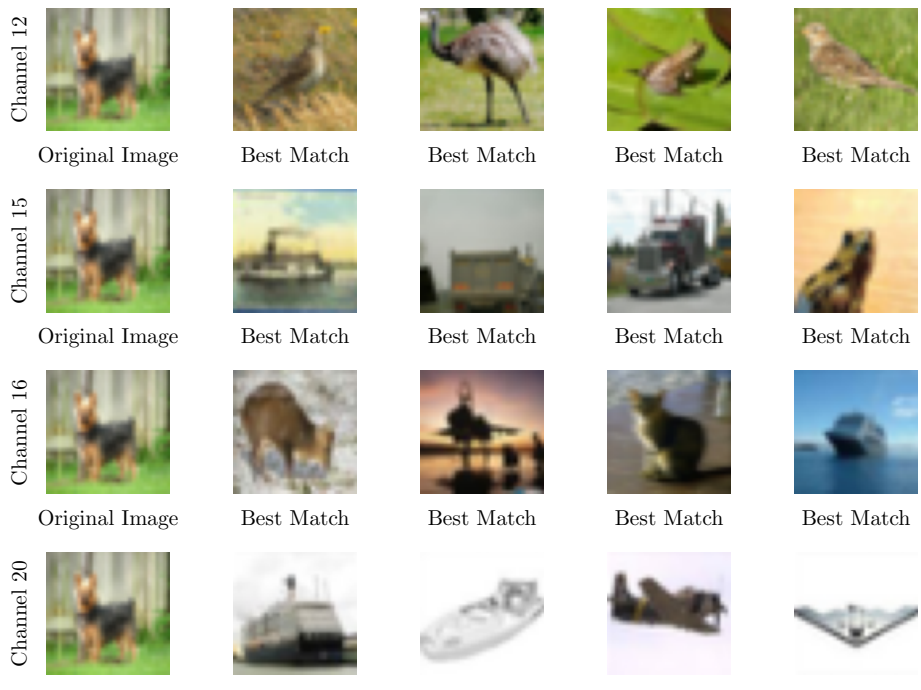
Convolutional Neural Networks

Intuition behind channels



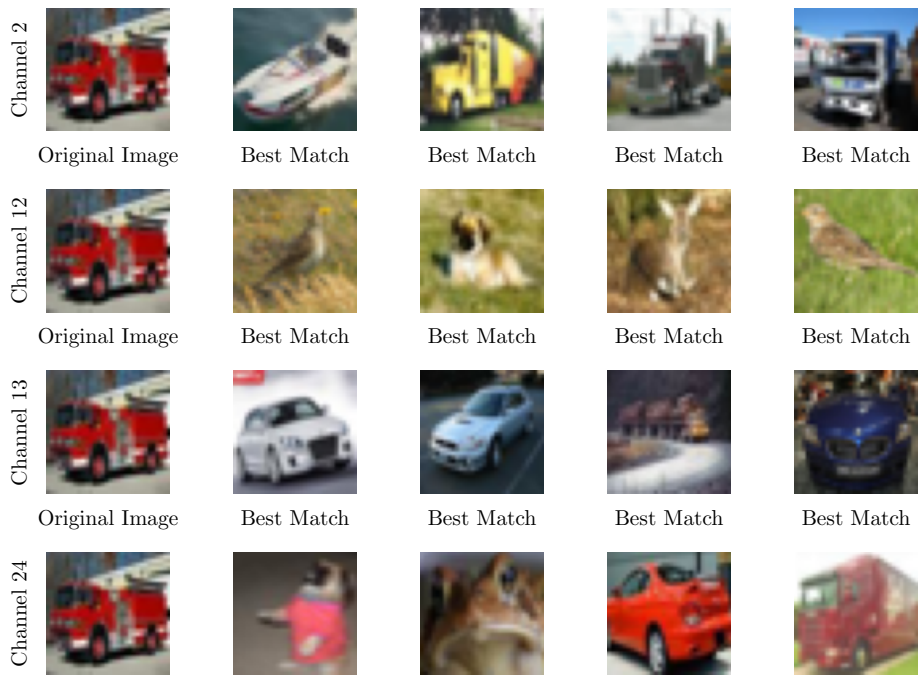
Convolutional Neural Networks

Intuition behind channels



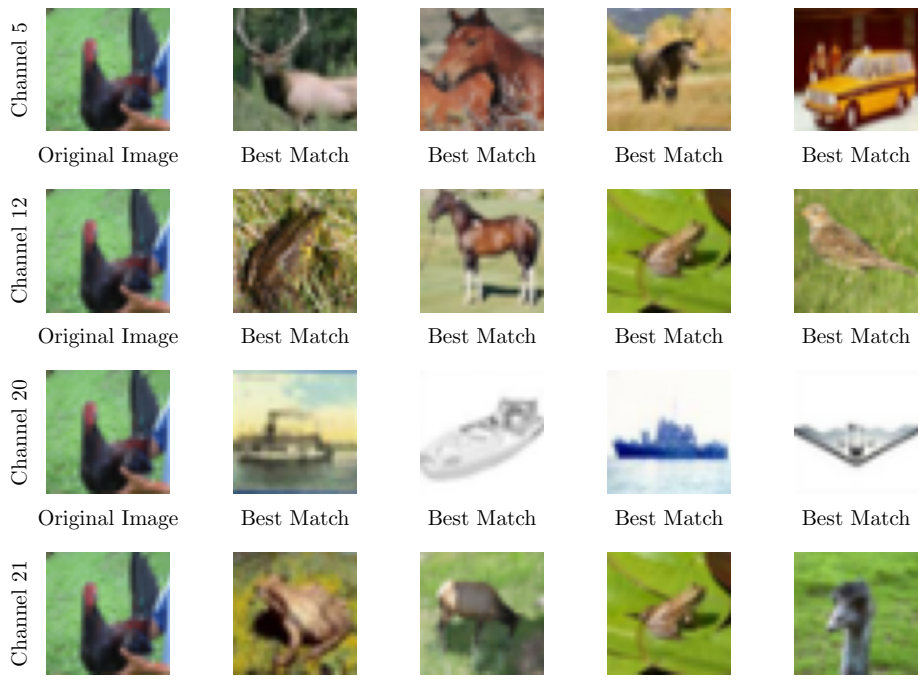
Convolutional Neural Networks

Intuition behind channels



Convolutional Neural Networks

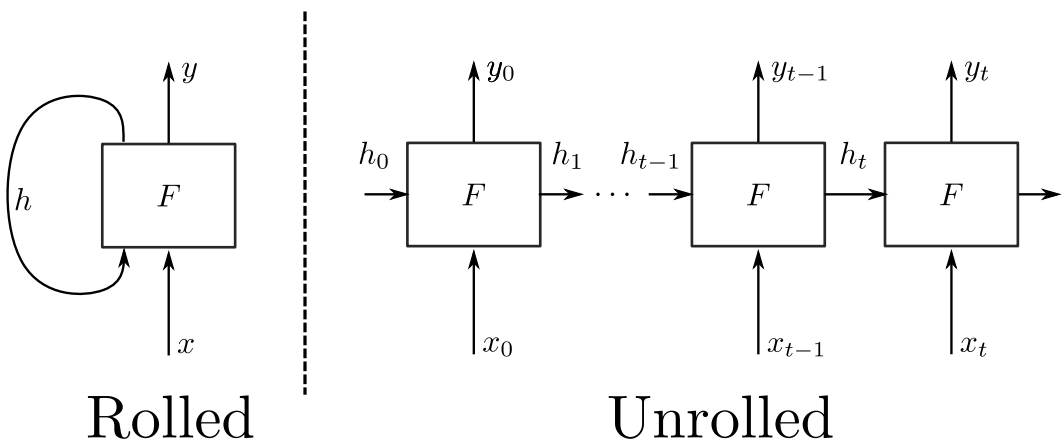
Intuition behind channels



Recurrent Neural Networks

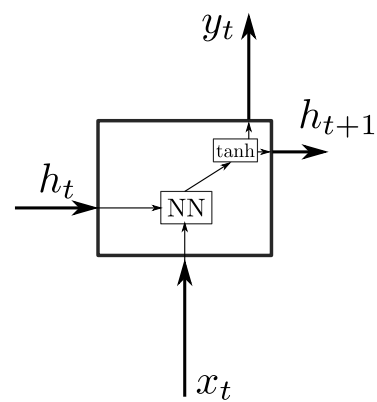
Recurrent Networks (RNNs) are a type of neural network that are specifically designed to handle sequential data, whereas CNNs are more suited for image and grid-like data. The main difference between RNNs and CNNs lies in the way they process data, with RNNs considering the sequence of elements and their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

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.



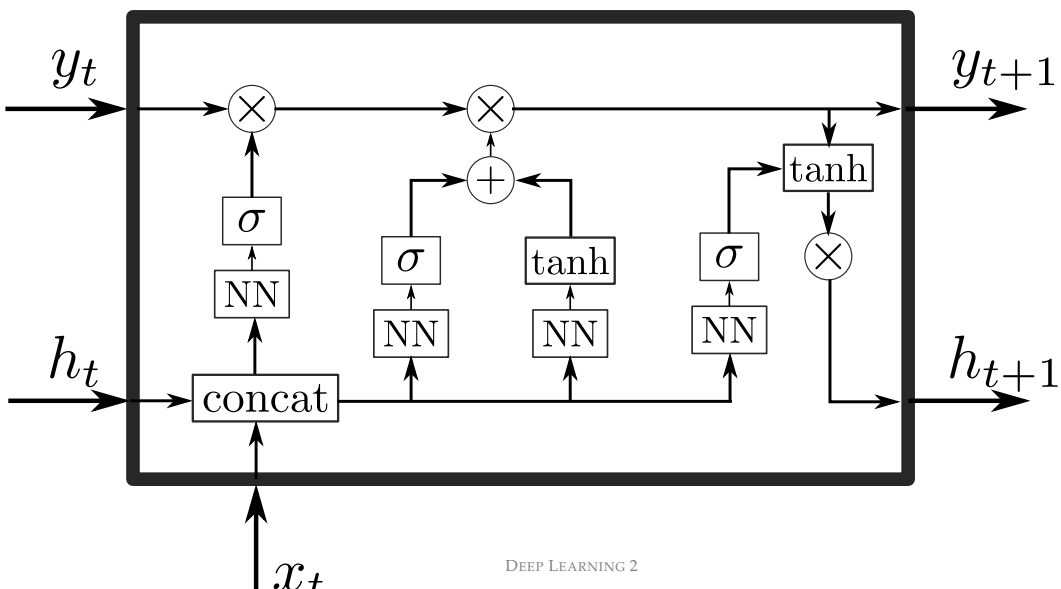
Some of the limitations of Vanilla RNNs:

- 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.

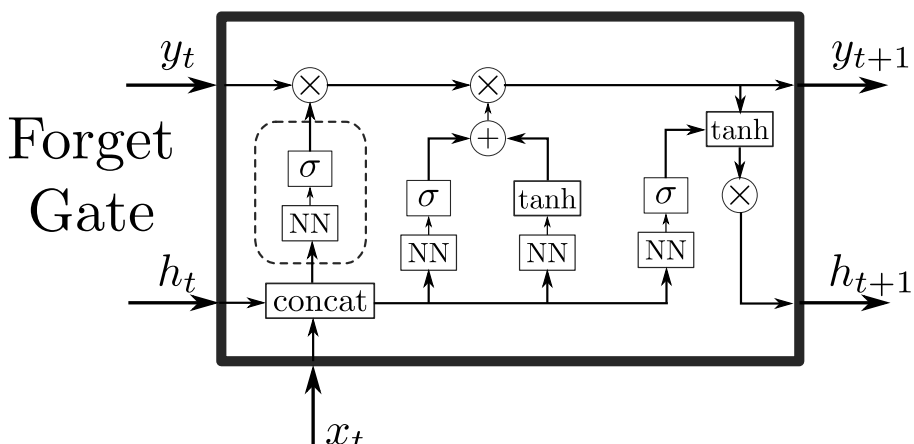


LSTM

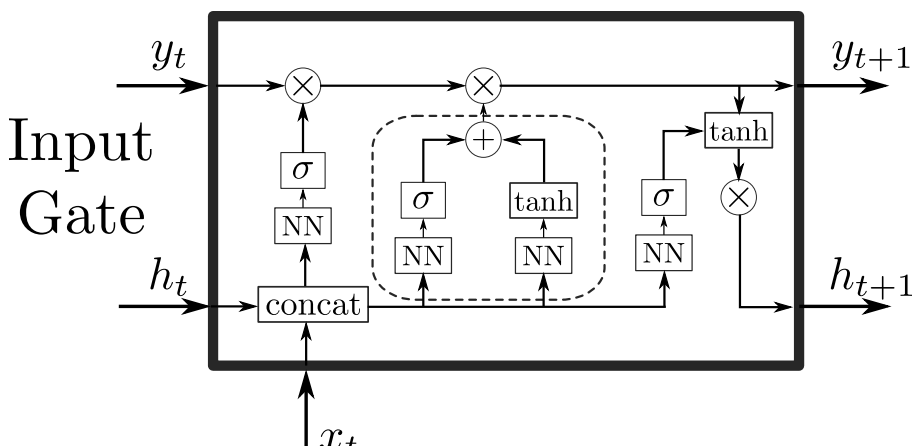
Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the vanishing gradient problem and difficulty in learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process for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



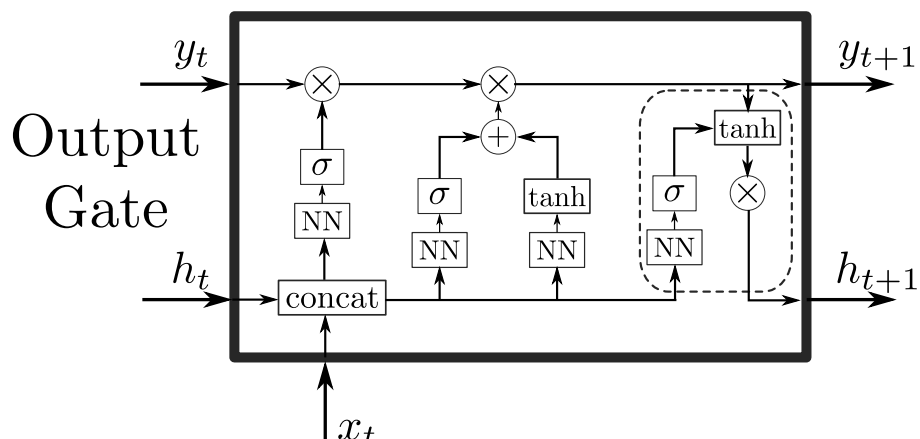
Long Short-Term Memory (LSTM) networks are a variant of recurrent neural networks (RNNs) that overcome some of the limitations of traditional RNNs, such as the problem of vanishing gradients and the difficulty of learning long-term dependencies. LSTM networks introduce memory cells, gates, and a process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



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.



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.

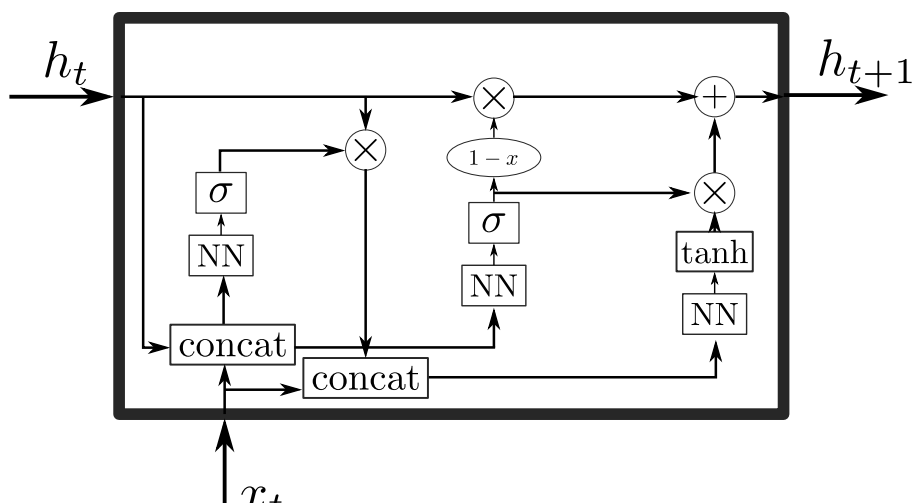


Limitations of LSTM RNNs:

- 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.

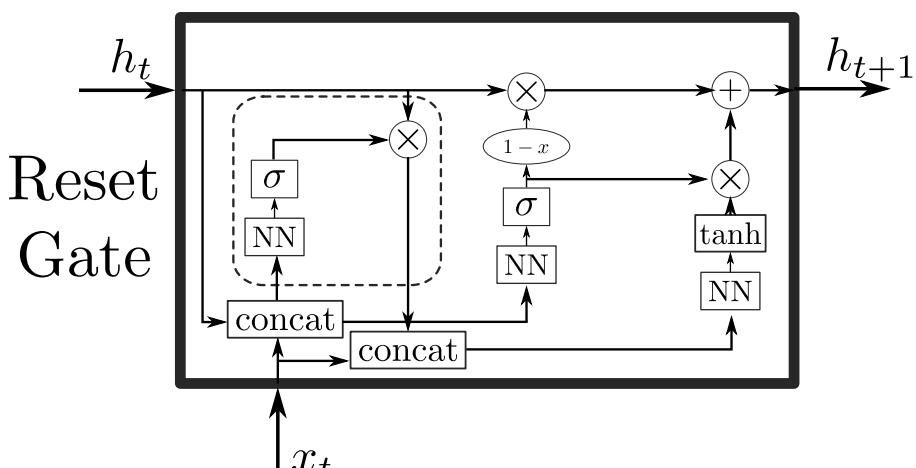
GRU

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.



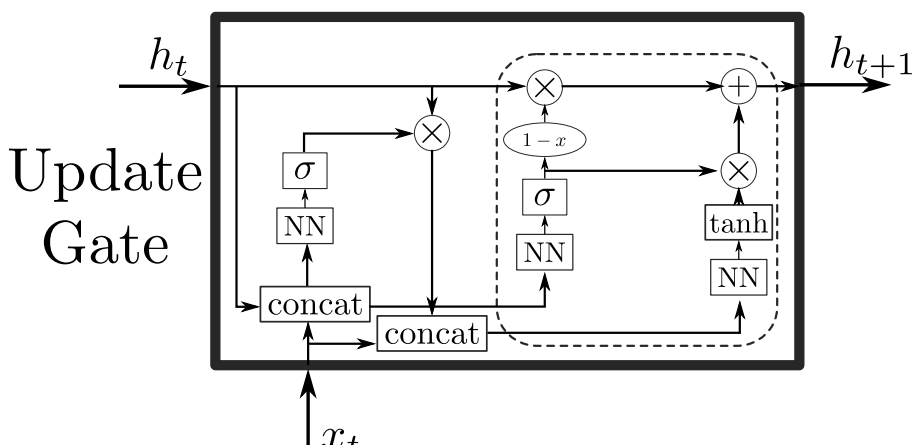
GRU

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.



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.

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

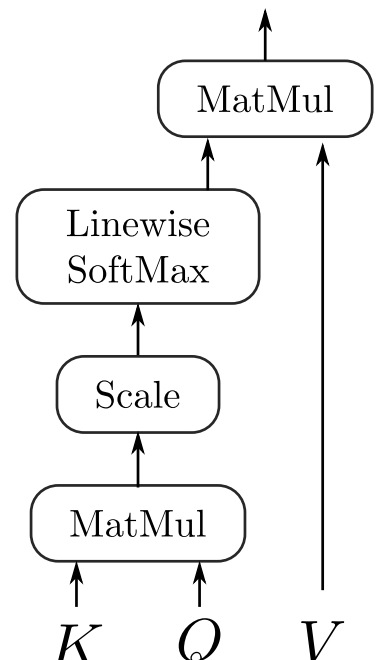
Transformers and Attention Mechanisms are relatively recent developments in the field of deep learning, which have become popular for processing sequential data, such as natural language processing (NLP) tasks. Unlike Recurrent Neural Networks (RNNs) which process sequential data by repeatedly applying 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.

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.

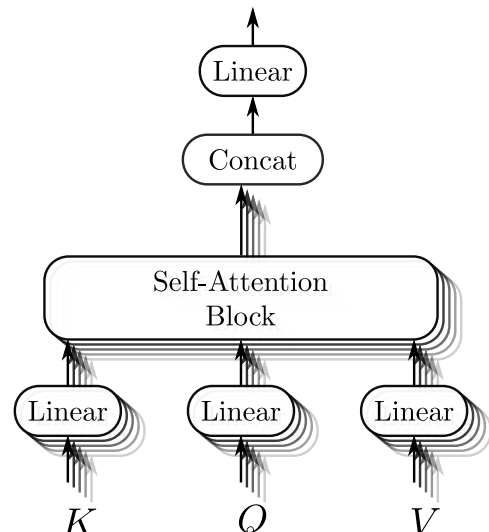
$$\text{Attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad \text{where } Q \in \mathbb{R}^{m \times d_k},\ K \in \mathbb{R}^{n \times d_k},\ V \in \mathbb{R}^{n \times d_v}$$

- 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.

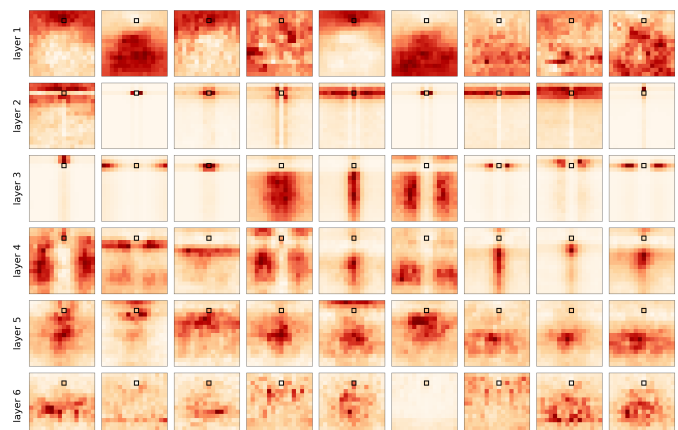
The resulting weighted sum of the values represents the output of the self-attention mechanism, capturing the relationships between different parts of the input sequence.



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.



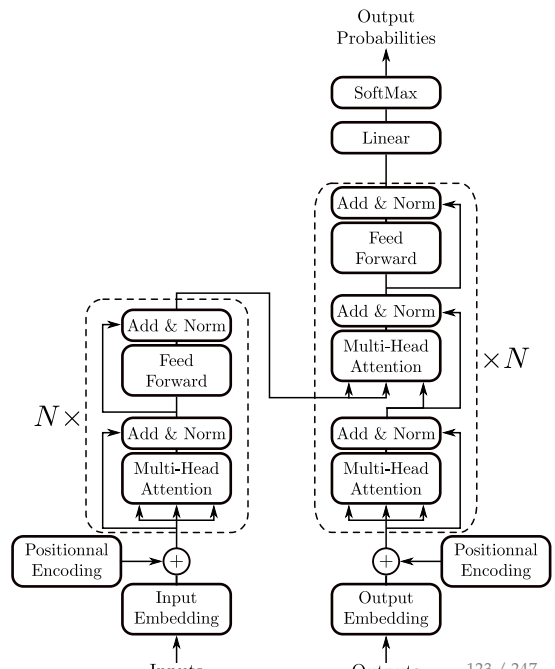
Visualizing Self-Attention for Image:
Link



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.



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

Definition

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

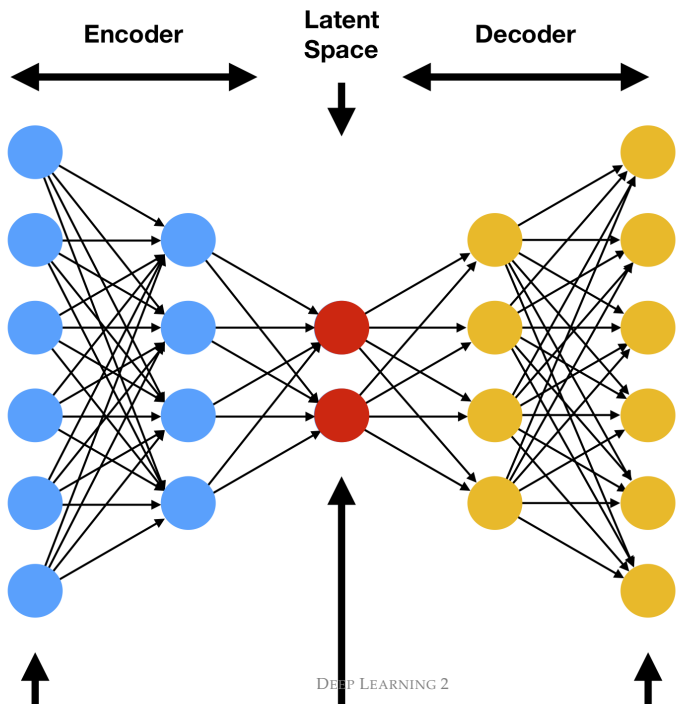
- An encoder function: $encoder(x) : \mathbb{R}^d \to \mathbb{R}^m$
  Maps an input x from the input space $\mathbb{R}^d$ to a hidden representation space $\mathbb{R}^m$.
- A decoder function: $decoder(z) : \mathbb{R}^m \to \mathbb{R}^d$
  Maps the hidden representation z back to the original input space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

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.

TP2: 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, θ 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:

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

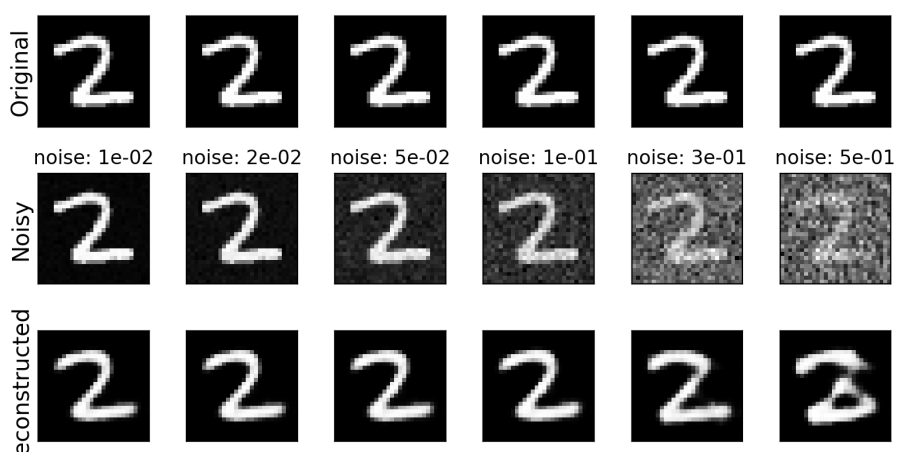
where x is the input data, θ represents the parameters of the encoder and decoder and $\hat{x} = \text{decoder}_\theta(\text{encoder}_\theta(x))$ in the reconstruction.

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.

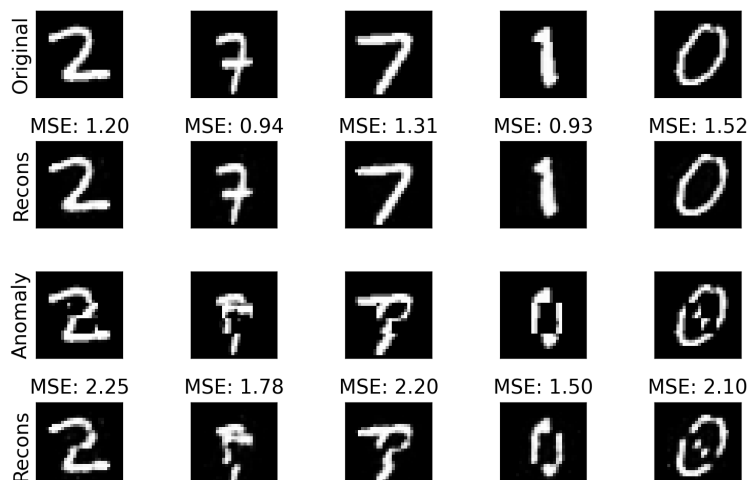
Applications of Autoencoders

- **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.



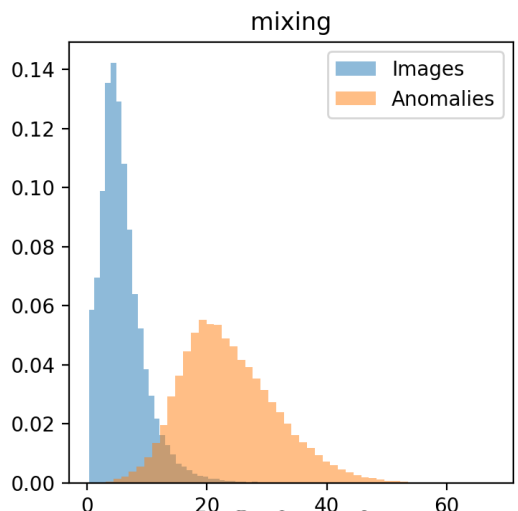
Applications of Autoencoders

- **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

Part III

Techniques to Improve Deep Learning Training

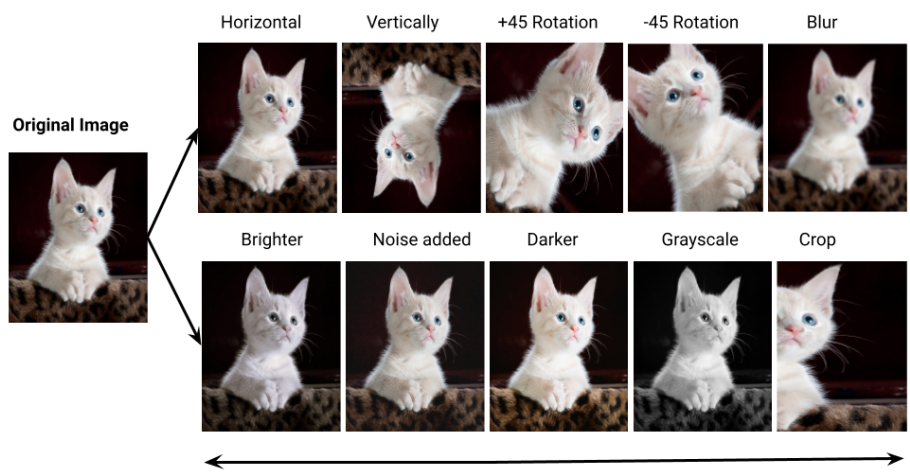
Techniques to Improve Deep Learning Training

- **Batch Normalization** - Seen
- **Dropout** - Seen
- **Data Augmentation**
- **Learning Rate Scheduling**
- **Early Stopping**
- **Gradient Clipping**
- **Weight Initialization**
- **Regularization**
- **GPU Acceleration**

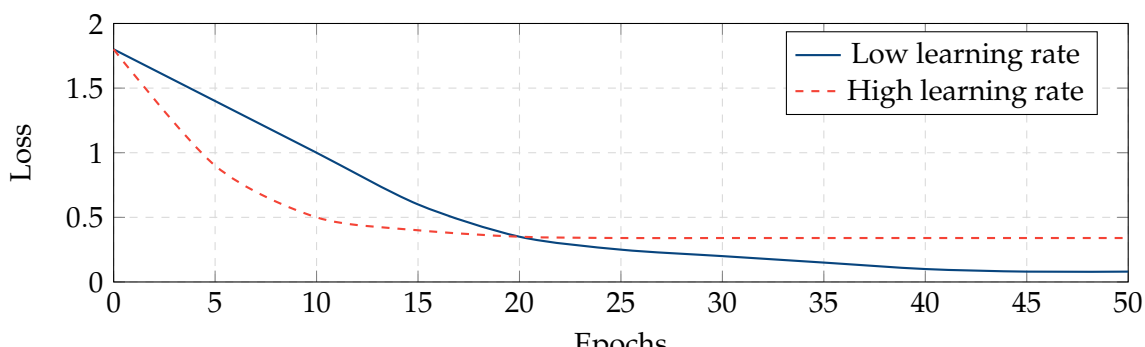
Techniques to Improve Deep Learning Training

Data Augmentation

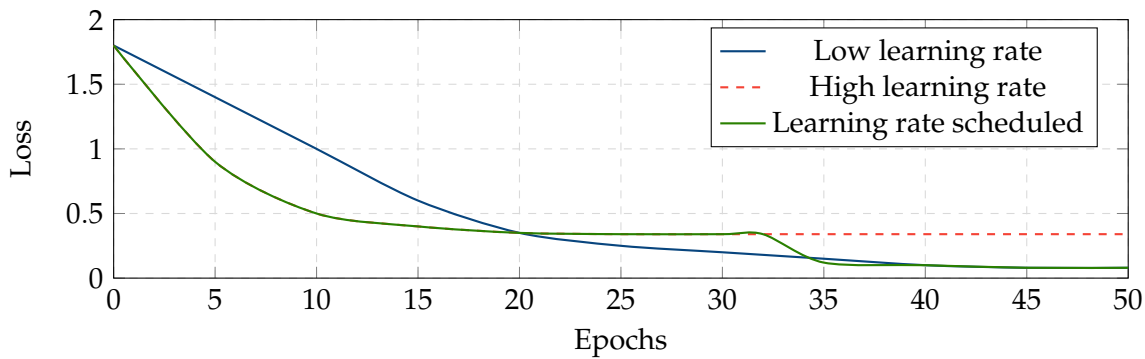
- Data Augmentation is a technique to increase the diversity of your training set by applying random (but realistic) transformations to the training images.
- The goal is to train a model that is robust to these transformations.
- For example, you can randomly rotate, scale, and flip the images in your training set.
- This helps expose the model to different aspects of the data and reduce overfitting.



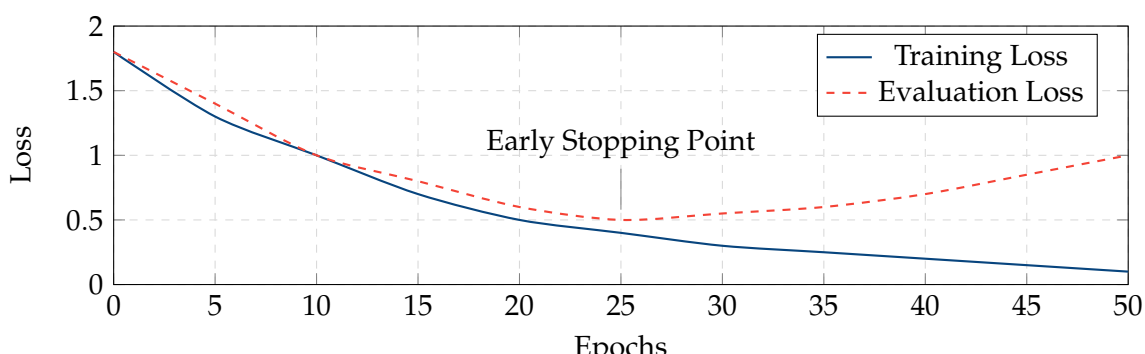
- The learning rate is one of the most important hyperparameters to tune for your deep learning model. The learning rate determines how quickly the model learns the optimal weights and how refined the gradient descent process is.
- If the learning rate is too high, the model may not converge or converge to a higher loss. If it is too low, the model may take too long to train.



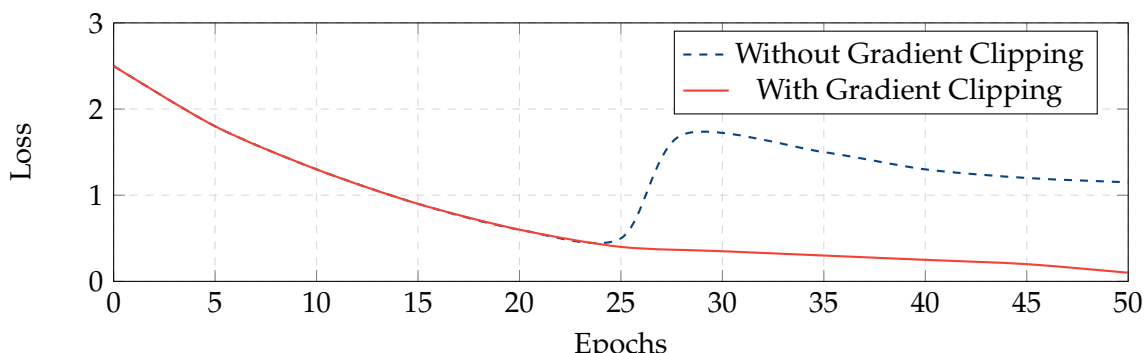
- The learning rate is one of the most important hyperparameters to tune for your deep learning model. The learning rate determines how quickly the model learns the optimal weights and how refined the gradient descent process is.
- If the learning rate is too high, the model may not converge or converge to a higher loss. If it is too low, the model may take too long to train.
- Learning rate scheduling is a technique to adjust the learning rate during training. For example, you can start with a high learning rate and then decrease it over time.



- Early stopping is a technique to prevent overfitting by stopping the training process when the model's performance on the validation set starts to degrade.
- The idea is to monitor the validation loss during training and stop training when the validation loss stops decreasing.



- Gradient clipping is a technique to prevent exploding gradients during training.
- Exploding gradients occur when the gradients of the loss function with respect to the model's parameters are too large.
- This can cause the model to diverge and fail to learn.
- Gradient clipping involves scaling the gradients if their norm exceeds a certain threshold.



- Weight initialization is a technique to set the initial values of the weights in the model.
- The initial values of the weights can have a significant impact on the training process and the final performance of the model.
- If the weights are initialized too small, the model may not learn effectively. If they are initialized too large, the model may not converge.
- Common weight initialization techniques include Xavier/Glorot initialization and He initialization.

- Xavier/Glorot initialization: The weights are initialized from a normal distribution with mean 0 and variance $2/(n_{\text{in}} + n_{\text{out}})$, where n_{in} and n_{out} are the number of input and output units, respectively. It helps prevent the gradients from vanishing or exploding during training by ensuring that the gradients have a similar scale.
- He initialization: The weights are initialized from a normal distribution with mean 0 and variance $2/n_{\text{in}}$, where n_{in} is the number of input units. It is commonly used for ReLU activation functions.

- Regularization is a technique to prevent overfitting by adding a penalty term to the loss function that discourages the model from learning complex patterns that may not generalize well.
- L1 regularization adds a penalty term to the loss function that is proportional to the absolute value of the weights. It encourages sparsity in the weights.
- L2 regularization adds a penalty term to the loss function that is proportional to the square of the weights. It encourages the weights to be small.

CPUs and GPUs are very different in terms of architecture and performance. CPUs are more suited for general-purpose computing tasks, while GPUs are optimized for parallel processing of simple operations, making them ideal for deep learning tasks.

- GPUs are much faster than CPUs for deep learning tasks because they have many more cores and can perform many more operations in parallel.
- Deep learning frameworks like PyTorch and TensorFlow are designed to take advantage of GPUs to accelerate the training process.
- Only the forward and backward passes of the model are executed on the GPU. The data loading and preprocessing are still done on the CPU.

Part IV

Deep Learning and Applications

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:

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

where $r(s_t, a_t)$ is the reward at time step t, γ 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:

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

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:

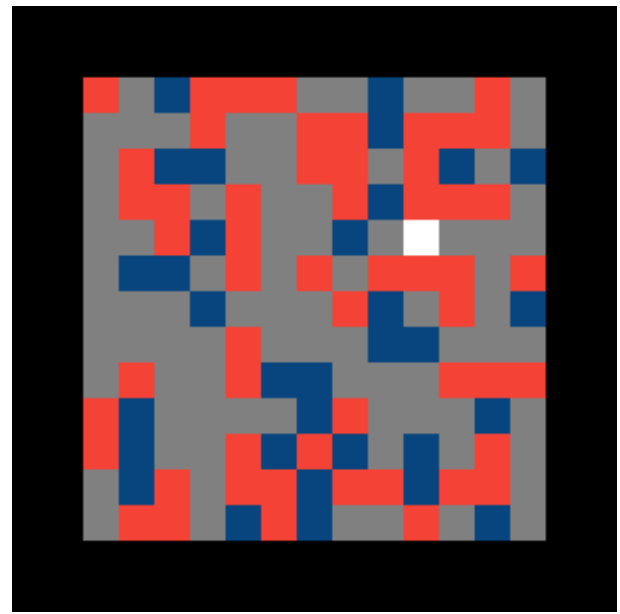
1. At the state s_t, select the action a_t with best reward using Q_t and store the results;
2. Obtain the new state s_{t+1} from the environment p;
3. Store (s_t, a_t, s_{t+1});
4. Obtain Q_{t+1} by minimizing $\mathcal{L}$ from a batch recovered from previously stored results.

Learning to act with Deep Reinforcement Learning

The Mouse Game

- 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.

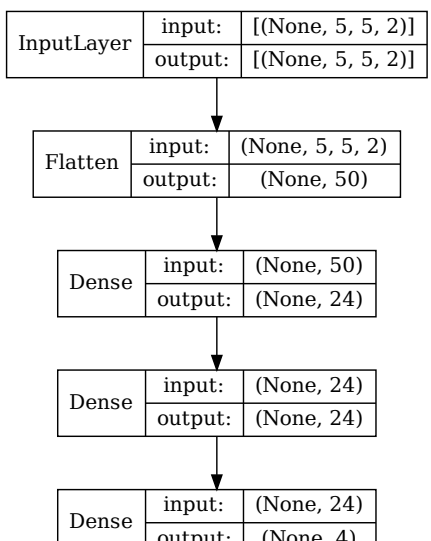
On this example, the mouse behaves randomly.

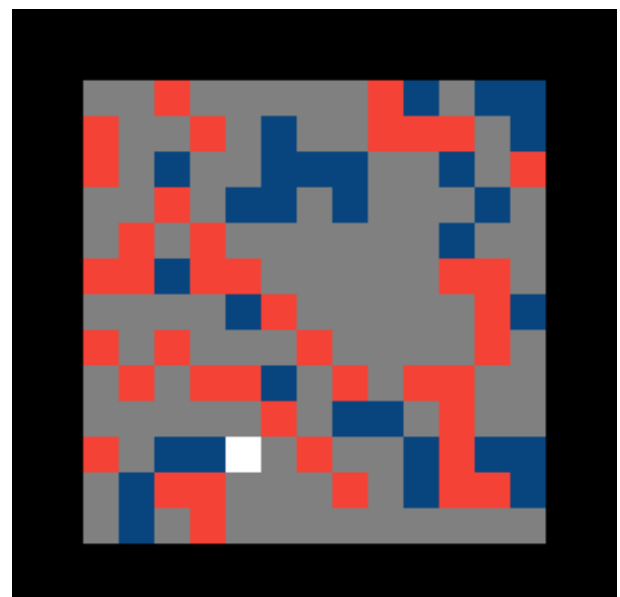


Learning to act with Deep Reinforcement Learning

Model 1

Fully connected network:

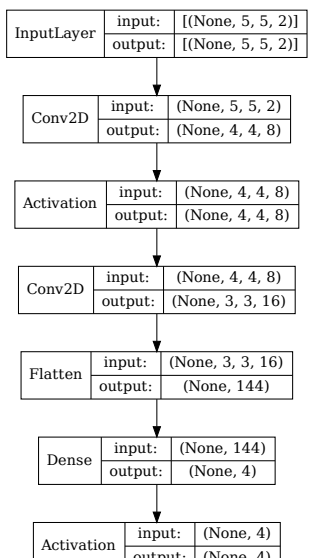


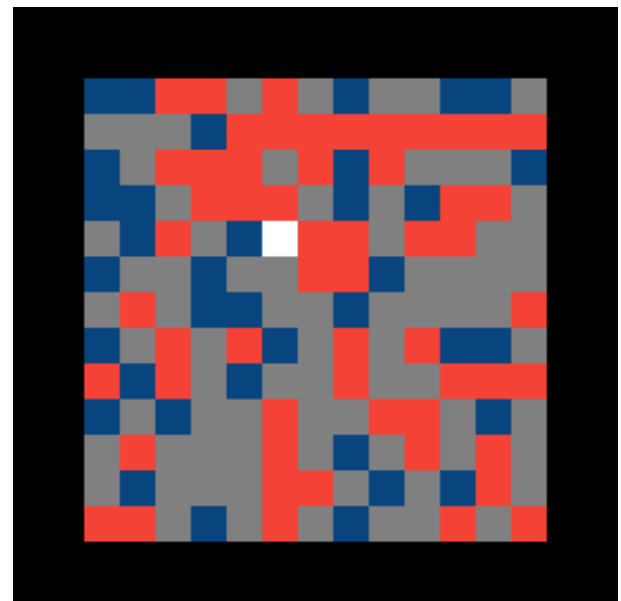


Learning to act with Deep Reinforcement Learning

Model 2

Convolutional network:



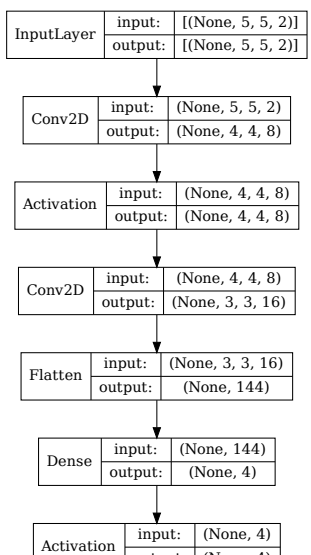


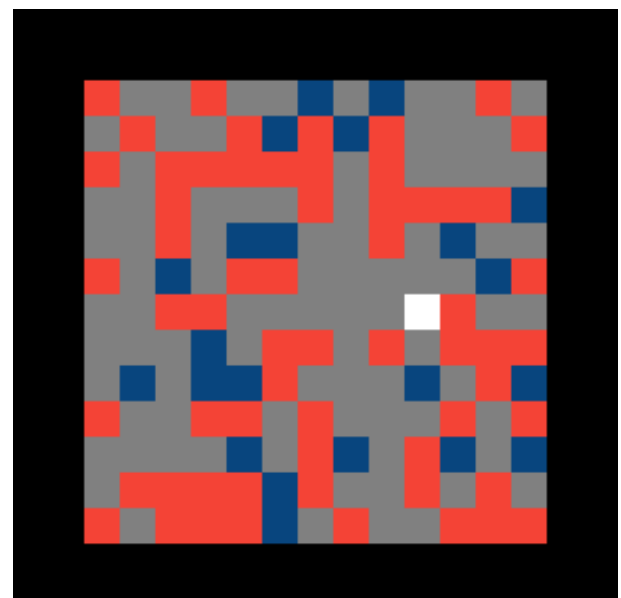
The ϵ-greedy algorithm is a common exploration strategy used in Deep Q learning. Balances exploration, where the agent tries out new actions and collects new data, and exploitation, where the agent uses the information it already has to select the action with the highest expected reward. The algorithm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

Learning to act with Deep Reinforcement Learning

Model 2+ Incorporated ϵ-exploration

Convolutional network + ϵ-greedy Algorithm during training:





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.

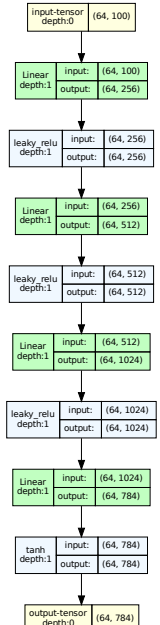
The loss of a (GAN) is defined as a minimax game between the generator and discriminator models. The generator aims to generate samples that are indistinguishable from real samples, while the discriminator aims to distinguish the generated samples from real samples. The loss function for the generator 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.

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

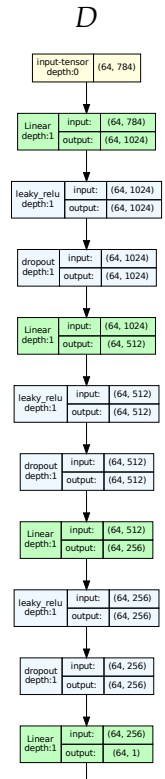
Synthetic Data Generation with Generative Adversarial Networks

MNIST Generation

G

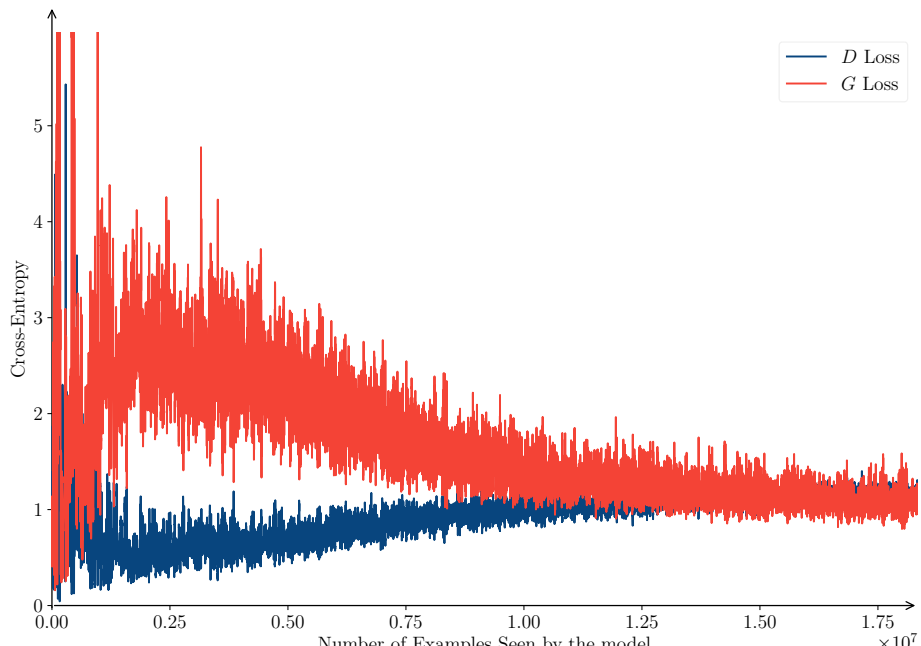


D



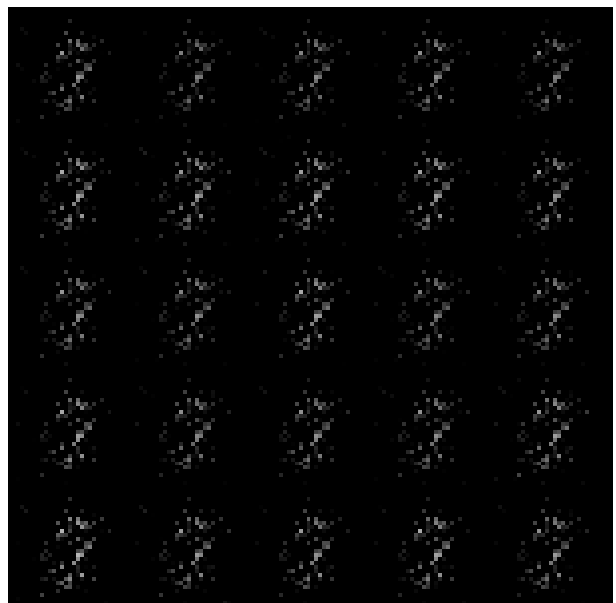
Synthetic Data Generation with Generative Adversarial Networks

MNIST Generation



Synthetic Data Generation with Generative Adversarial Networks

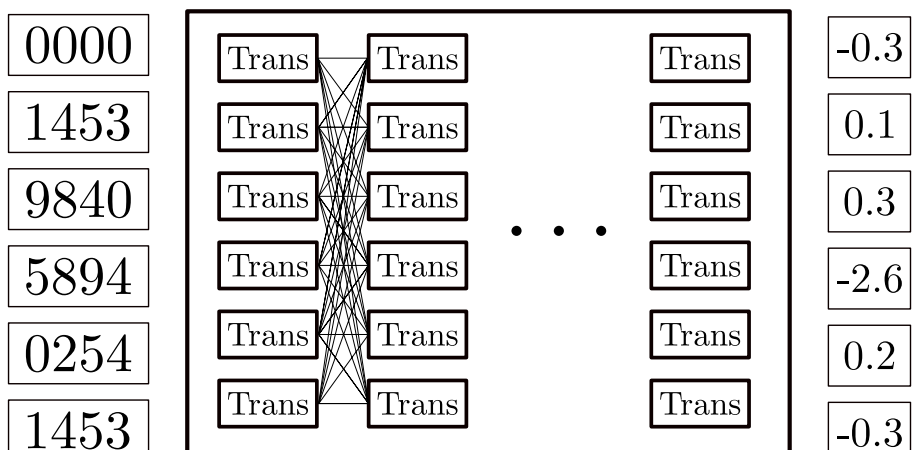
MNIST Generation



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.

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	

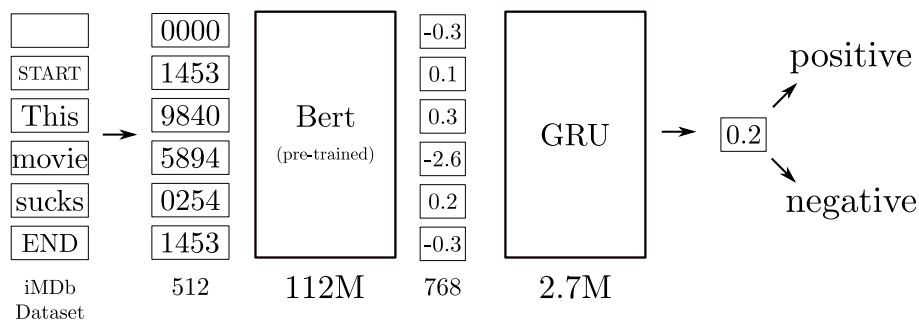
Entity-level masking

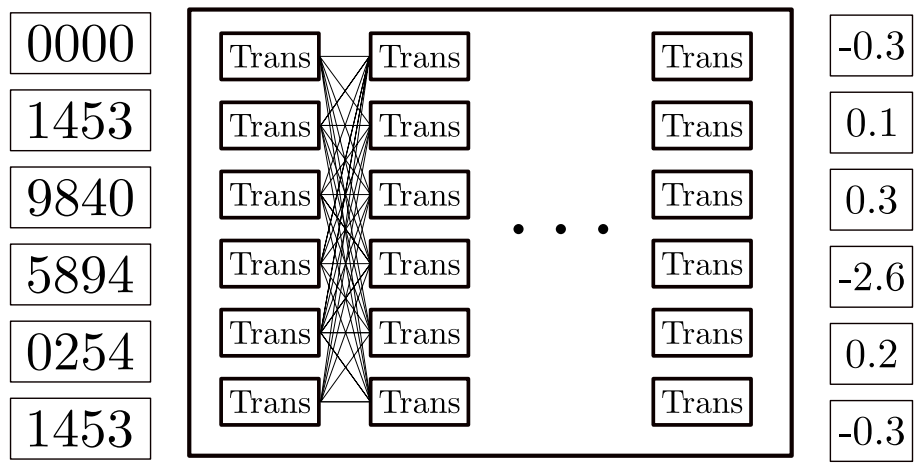
There	is			in	Elvish,	Entish,	or	the				for	this	treachery.

Phrase-level masking

			curse	in	Elvish,	Entish,	or					for	this	treachery.

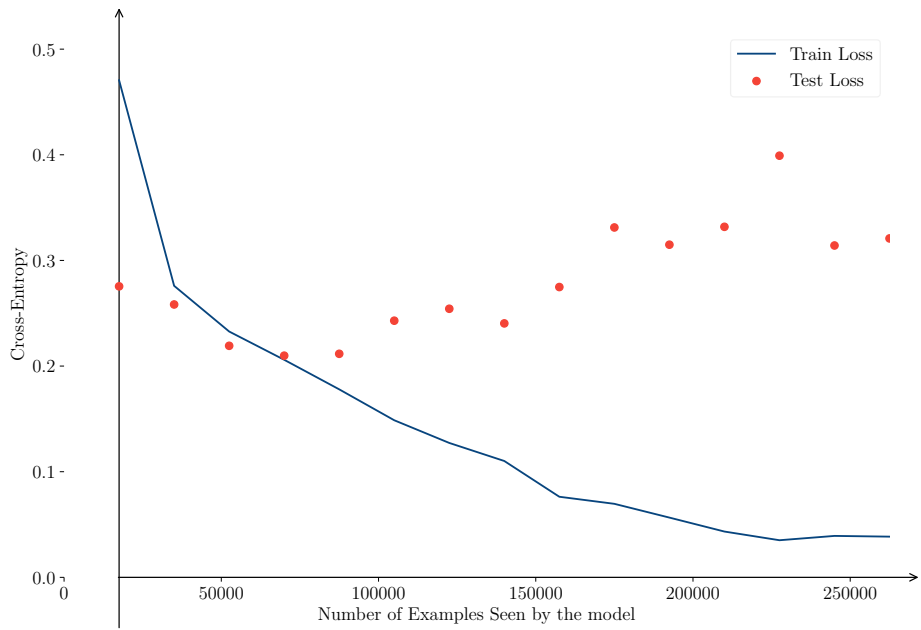
Bert can be fine-tuned for sentiment analysis by adding a classifier layer on top of the pretrained Bert model. The layer is trained on a labeled sentiment analysis dataset to predict the sentiment of a given input sequence, which can be a sentence, paragraph, or document. Fine-tuning the model allows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




0000
1453
9840
5894
0254
1453
Trans
-0.3
0.1
0.3
-2.6
0.2
-0.3
Bert
(pre-trained)

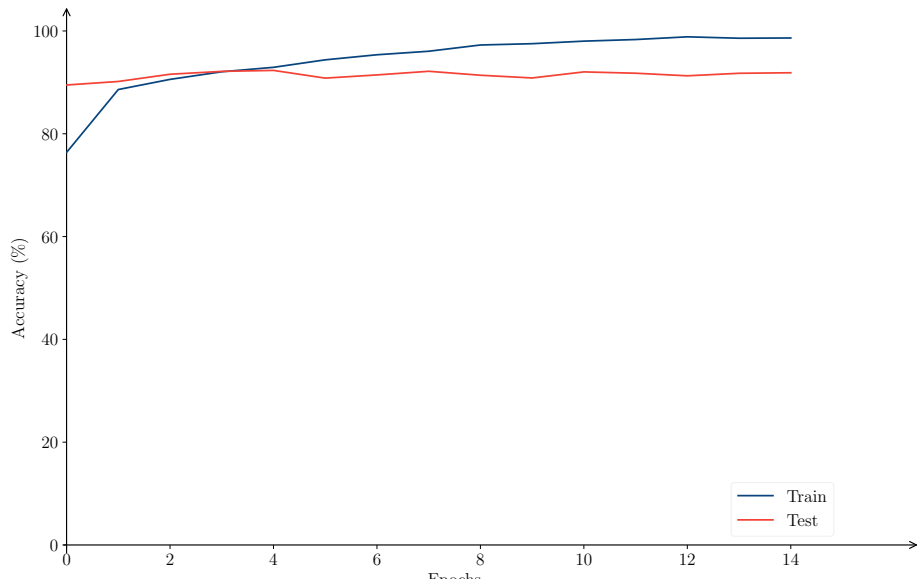
Sentiment Analysis with Transformers and GRU

Sentiment Analysis



Sentiment Analysis with Transformers and GRU

Sentiment Analysis



Sentiment Analysis with Transformers and GRU

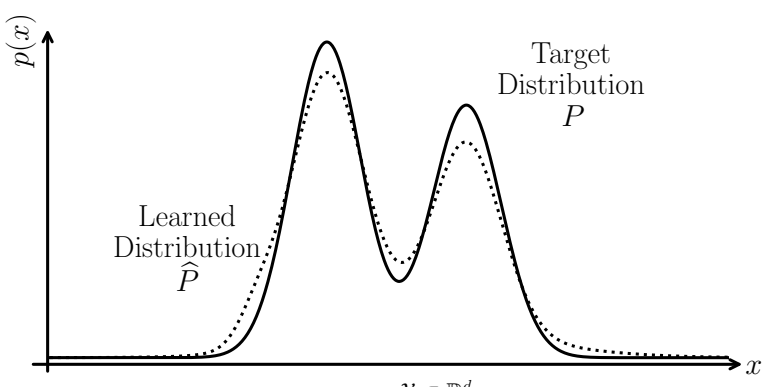
Sentiment Analysis

```
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
```

Density Estimation with Normalizing Flows

Estimating Density

In Machine Learning, the task of *density estimation* consists in estimating the probability density function of a random variable from a set of observations. This is a fundamental problem in statistics and machine learning, with applications in a wide range of fields, including anomaly detection, clustering, and generative modeling. There are several methods for density estimation, including parametric models, non-parametric models, and deep learning models. In this section, we will focus on a specific type of deep learning model called Normalizing Flows, which is used for density estimation and generative modeling.



Density Estimation with Normalizing Flows

Overview

A Normalizing Flow is usually seen as:

- a *generative model*,
- a *bijective mapping*,
- an *invertible neural network*,
- a *density estimator*.

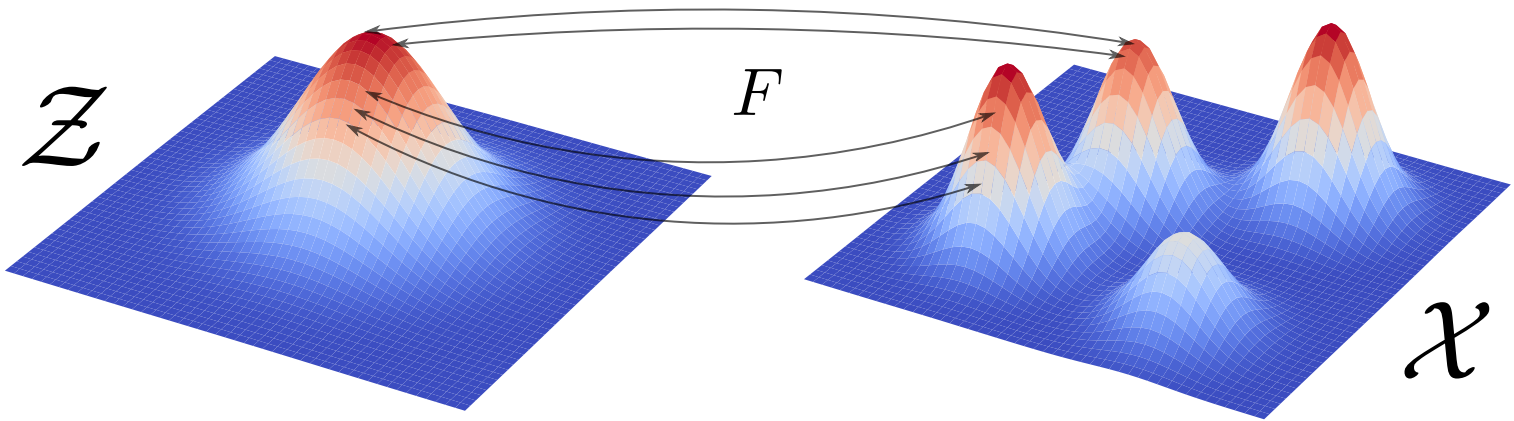


Figure. A mapping between two probability distributions
Point to point

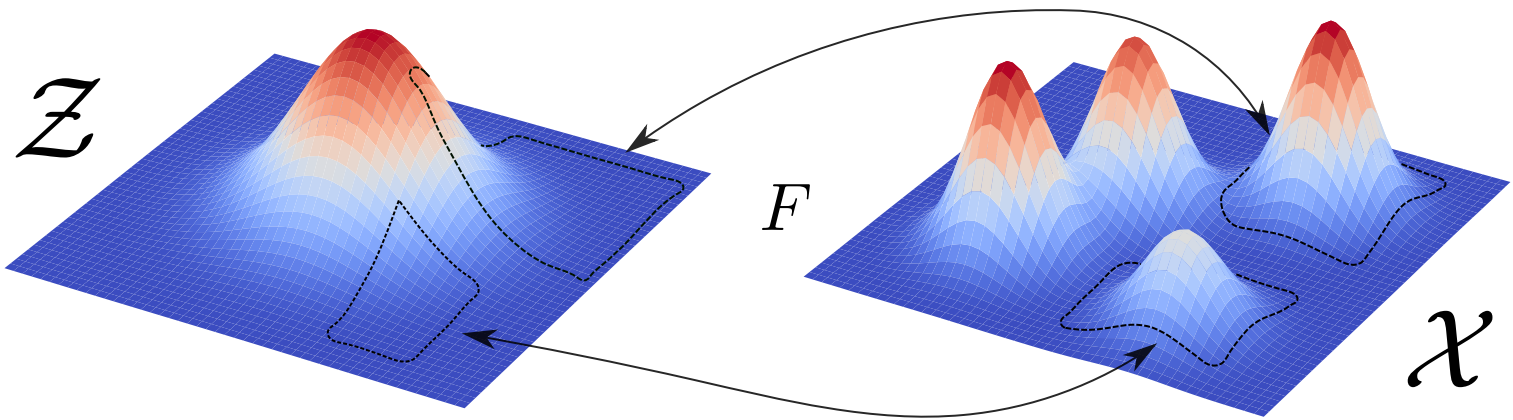


Figure. A mapping between two probability distributions
Subset to subset

Normalizing Flow

A Normalizing Flow is a bijective function between a data space $\mathcal{X}$ and a latent space $\mathcal{Z}$, both subset of $\mathbb{R}^d$.

$$\begin{array}{rccc} F: & \mathcal{X} & \longmapsto & \mathcal{Z} \\ & x & \longmapsto & z = F(x) \end{array}$$

Data and Latent Distributions

In theory, a NF maps a target distribution P, ie the data distribution to a simple latent distribution Q.

Usually, Q is set to be a Normal Gaussian multivariate distribution $\mathcal{N}(0_d, I_d)$. p and q are respectivelly the probability densities of P and Q.

Density Estimation with Normalizing Flows

How does it work ?

In practice, the mapping is *not perfect*. P^* induces a distribution Q and similarly, the latent distribution Q induces $\widehat{P}$, which is the learned distribution. The forward pass F is called the *Normalizing* direction while the inverse pass F^{-1} is called the *Generative* direction.

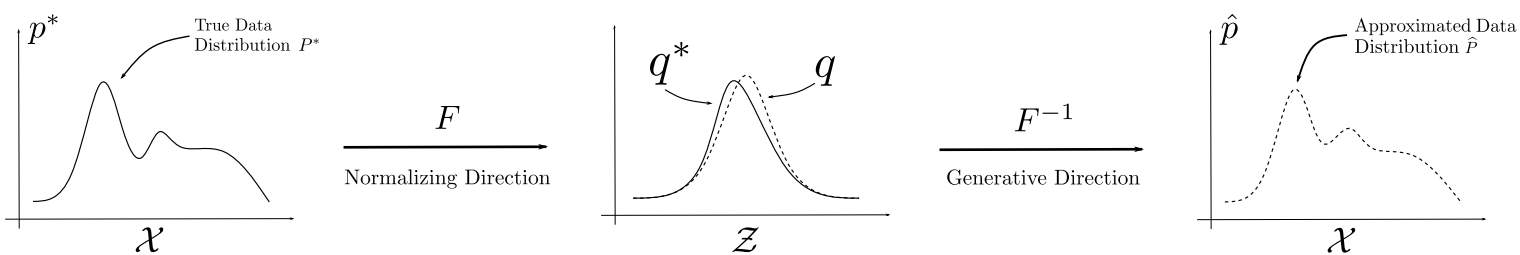


Figure. 1D Normalizing Flow process.

Change of Variable Formula

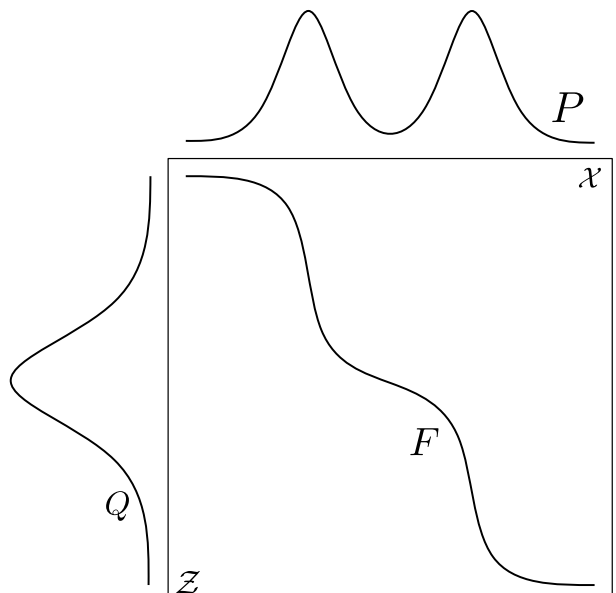
For a bijective and continuous fonction F and a latent distribution Q, the distribution induced by Q and F is defined through the *change of variable formula*:

$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \text{Jac}_F(x)|\, q(F(x)). \tag{1}$$

Density Estimation with Normalizing Flows

Induced Probabilities ?

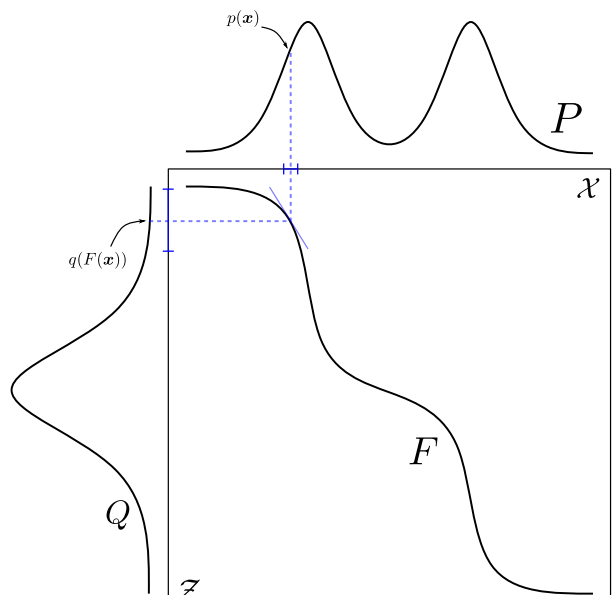
$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \mathrm{Jac}_F(x)|\, q(F(x)).$$



Density Estimation with Normalizing Flows

Induced Probabilities ?

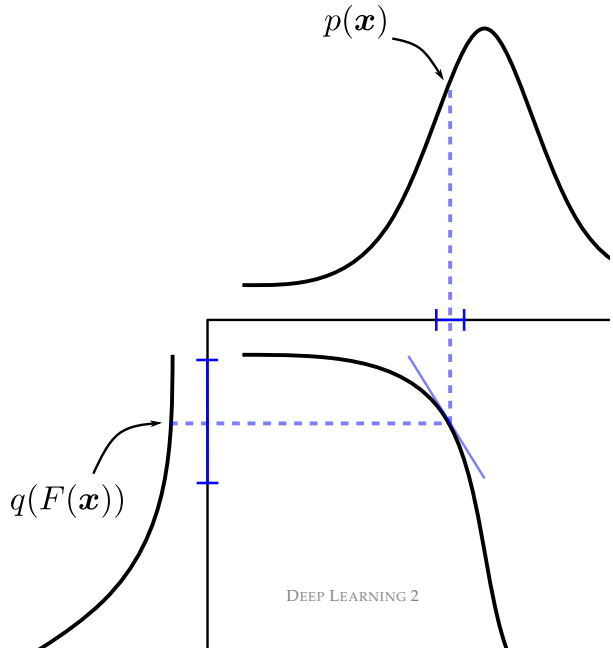
$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \mathrm{Jac}_F(x)|\, q(F(x)).$$



Density Estimation with Normalizing Flows

Induced Probabilities ?

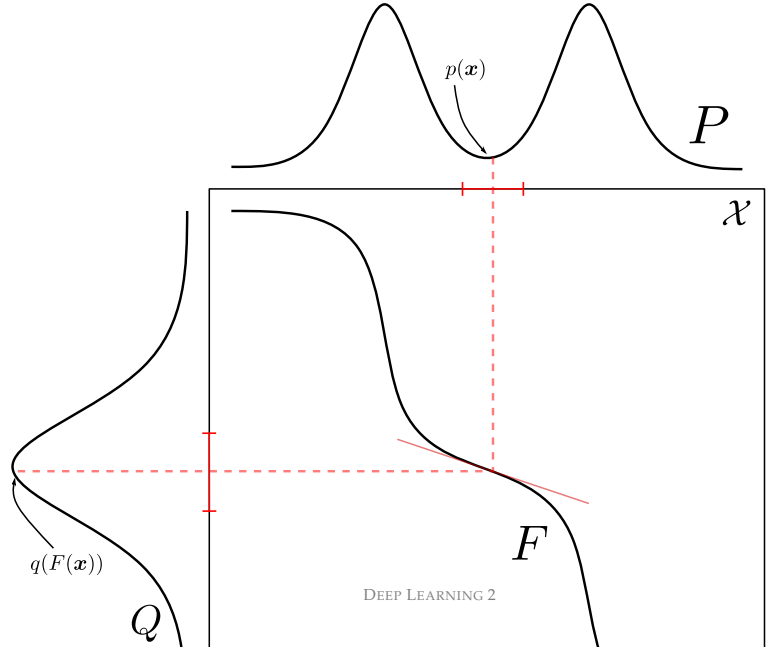
$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \mathrm{Jac}_F(x)|\, q(F(x)).$$



Density Estimation with Normalizing Flows

Induced Probabilities ?

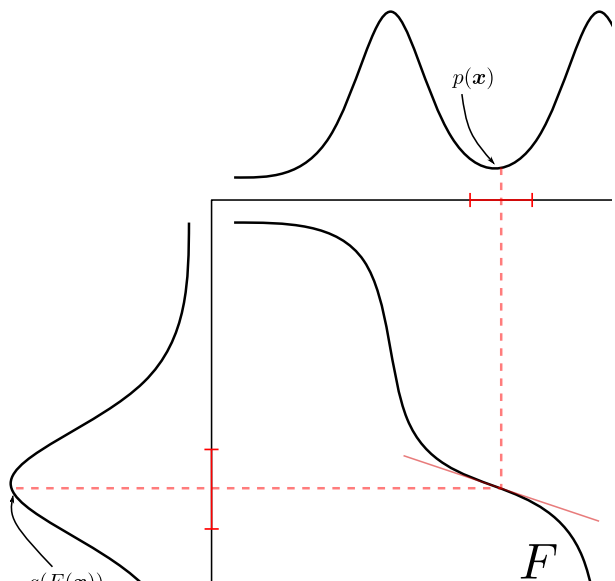
$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \mathrm{Jac}_F(x)|\, q(F(x)).$$



Density Estimation with Normalizing Flows

Induced Probabilities ?

$$\forall x \in \mathcal{X}, \quad \hat{p}(x) = |\det \mathrm{Jac}_F(x)|\, q(F(x)).$$



To perfom density estimation:

1. Draw $x \sim P^*$,
2. Compute $F(x)$ and $|\det \text{Jac}_F(x)|$,
3. Compute $\widehat{p}(x) = q(F(x))|\det \text{Jac}_F(x)|$.

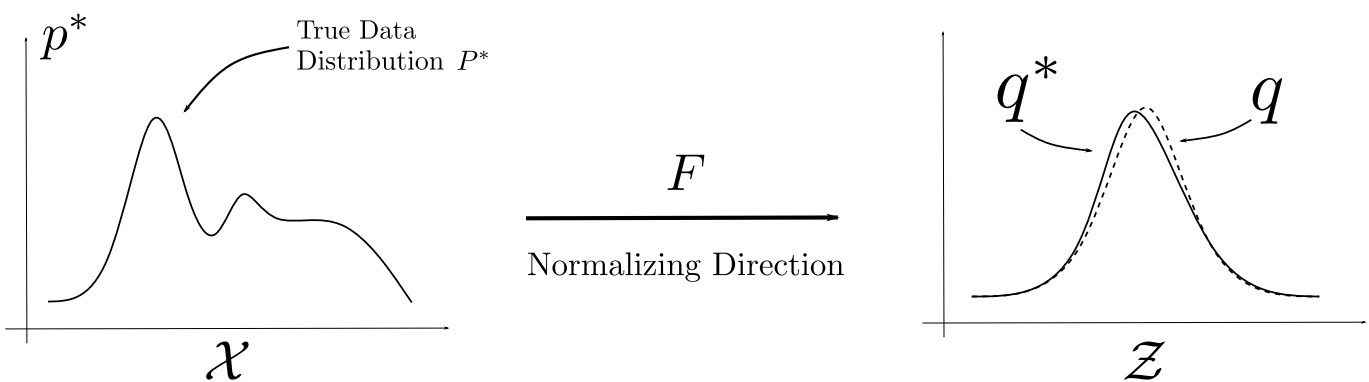


Figure. 1D Normalizing Process of Density Estimation.

To perform data generation:

1. Draw $z \sim Q$,
2. Compute $x = F^{-1}(x)$.

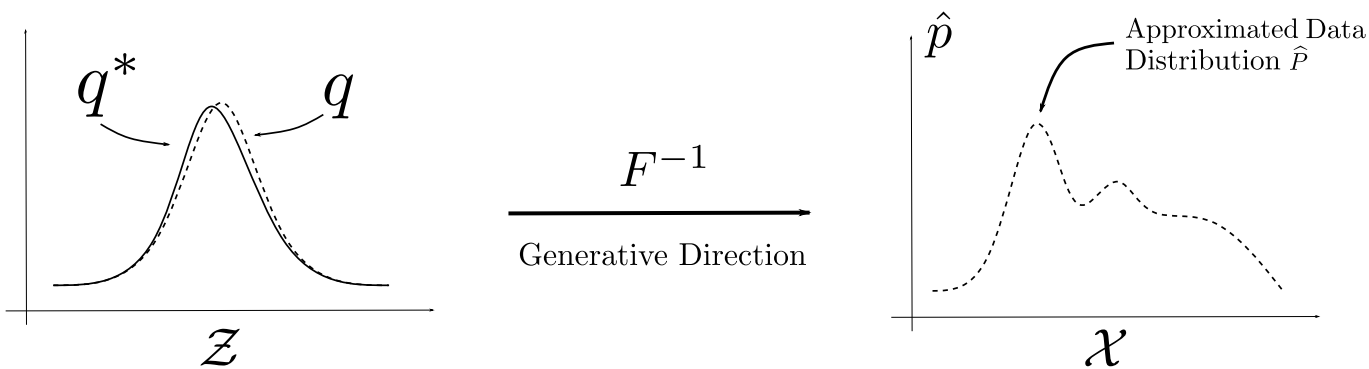


Figure. 1D Normalizing Flow process of Generation.

Loss

The objective is to approximate P^* with $\widehat{P}$. We can minimize the Kullback-Leiber Divergence :

$$\theta = \arg\min_{\theta} \mathcal{D}_{\mathrm{KL}}(P^* \| \widehat{P}).$$

This is equivalent to maximizing the log likelihood :

$$\theta = \arg\max_{\theta} \mathbb{E}_{x \sim \mathcal{X}} \left[\log \widehat{p}(x) \right].$$

$$\mathcal{D}_{\mathrm{KL}}(P^* \| \widehat{P}) = \int_{\mathcal{X}} p^*(x) \log \left(\frac{p^*(x)}{\widehat{p}(x)} \right) dx$$

$$\mathrm{nll} = -\mathbb{E}_{x \sim \mathcal{X}} \left[\log \widehat{p}(x) \right]$$

Density Estimation with Normalizing Flows

Learning steps

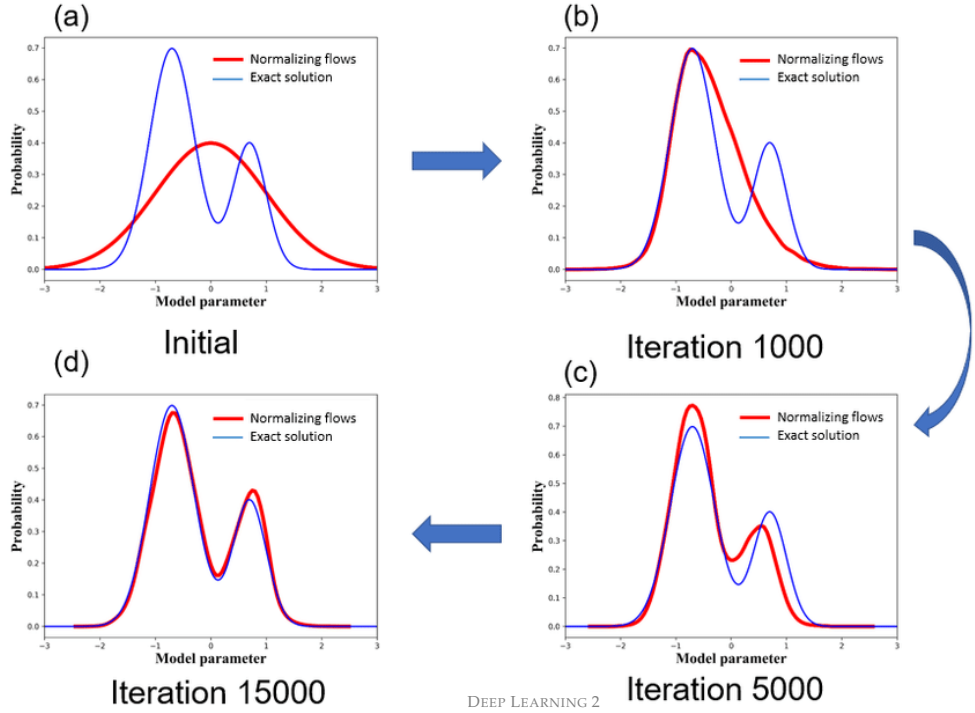


Figure. Learning Process for a 1D Normalizing Flow.

Image Segmentation with U-Net

Image Segmentation

Image segmentation is the process of partitioning an image into multiple segments or regions based on the characteristics of the pixels. It is a fundamental task in computer vision and has applications in various fields, including medical imaging, autonomous driving, and satellite image analysis. There are several methods for image segmentation, including thresholding, clustering, and deep learning-based approaches. In this section, we will focus on a deep learning model called U-Net, which is commonly used for image segmentation tasks.

Image segmentation has a wide range of applications in computer vision and image processing, including:

- Medical Imaging: Segmentation of organs, tumors, and other structures in medical images.
- Autonomous Driving: Segmentation of objects such as cars, pedestrians, and road signs in images captured by autonomous vehicles.
- Satellite Image Analysis: Segmentation of land cover types, buildings, and other features in satellite images.
- Object Detection: Segmentation of objects in images to localize and classify them.
- Image Editing: Segmentation of objects for image editing tasks such as background removal and image compositing.

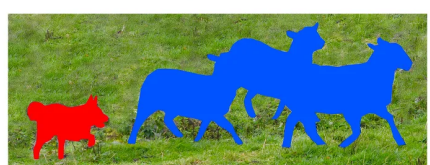
Image Segmentation with U-Net

Type of Segmentations

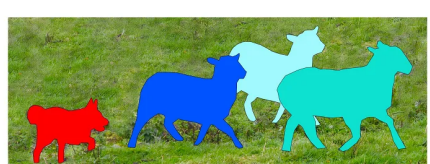
Image segmentation can be broadly classified into two types: semantic segmentation and instance segmentation.

- **Semantic Segmentation:** Semantic segmentation assigns a class label to each pixel in an image, such as road, car, person, etc. The goal is to partition the image into semantically meaningful regions.
- **Instance Segmentation:** Instance segmentation goes a step further than semantic segmentation by distinguishing between different instances of the same class. It assigns a unique label to each object instance in the image.

Original Image



Semantic Segmentation

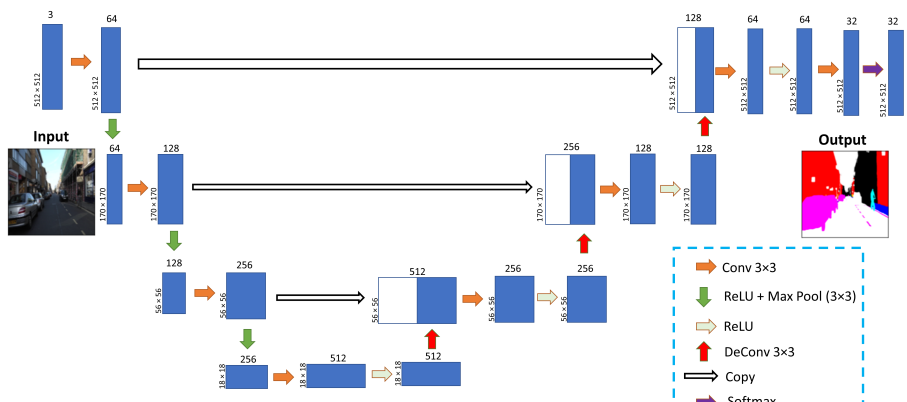


Instance Segmentation

Image Segmentation with U-Net

U-Net Architecture

U-Net is a convolutional neural network architecture designed for image segmentation tasks. It consists of an encoder-decoder structure with skip connections that allow the model to capture both local and global features in the input image.



3 main loss functions are used for image segmentation tasks:

- **Categorical Cross-Entropy Loss:** Used for multi-class segmentation tasks where each pixel is classified into one of multiple classes.
- **Dice Loss:** A similarity-based loss function that measures the overlap between the predicted segmentation mask and the ground truth mask.
- **Shaped aware Loss:** A loss function that penalizes the model for making errors near the object boundaries, where the segmentation is most critical.

3 main loss functions are used for image segmentation tasks:

- **Categorical Cross-Entropy Loss:** Used for multi-class segmentation tasks where each pixel is classified into one of multiple classes.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

- **Dice Loss:** A similarity-based loss function that measures the overlap between the predicted segmentation mask and the ground truth mask.
- **Shaped aware Loss:** A loss function that penalizes the model for making errors near the object boundaries, where the segmentation is most critical.

3 main loss functions are used for image segmentation tasks:

- **Categorical Cross-Entropy Loss:** Used for multi-class segmentation tasks where each pixel is classified into one of multiple classes.
- **Dice Loss:** A similarity-based loss function that measures the overlap between the predicted segmentation mask and the ground truth mask.

$$\mathcal{L} = 1 - \frac{2\sum_{i=1}^{N} y_i \hat{y}_i}{\sum_{i=1}^{N} y_i + \sum_{i=1}^{N} \hat{y}_i}$$

- **Shaped aware Loss:** A loss function that penalizes the model for making errors near the object boundaries, where the segmentation is most critical.

3 main loss functions are used for image segmentation tasks:

- **Categorical Cross-Entropy Loss:** Used for multi-class segmentation tasks where each pixel is classified into one of multiple classes.
- **Dice Loss:** A similarity-based loss function that measures the overlap between the predicted segmentation mask and the ground truth mask.
- **Shaped aware Loss:** A loss function that penalizes the model for making errors near the object boundaries, where the segmentation is most critical.

$$\mathcal{L} = -w(x) \left[\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}) \right]$$

with $w(x)$ a weight function that assigns higher weights to pixels near the object boundaries.

Part V

Managing a Deep Learning Project

Why Notebooks are not enough

Where Notebooks Fall Short

Jupyter or Colab Notebooks are great for prototyping, showcasing, and data exploration. However, they are not enough for managing a deep learning project. In particular, they lack the following features:

- **Reproducibility:** Notebooks are not reproducible. They are not designed to be run multiple times with the same results.
- **Versioning:** Notebooks are not versioned. It is hard to track changes and revert to versions.
- **Modularity:** Notebooks are not modular. It is hard to reuse code across different notebooks.
- **Scalability:** Notebooks are not scalable. They are not designed to run on multiple machines or in the cloud.
- **Monitoring:** Notebooks are not monitored. It is hard to track the progress of a long-running experiment.
- **Collaboration:** Notebooks are not collaborative. It is hard to work with multiple people on the same notebook.
- **Deployment:** Notebooks are not deployable. They are not designed to be run in production.
- **Performance:** Notebooks are not performant, depending on the amount of log, they can be up to 50% slower than a script.

Why Notebooks are not enough

Why Scripts are Better

Directly running Python script is a better alternative to Jupyter Notebooks for a deep learning project. They allow:

- **Support for Scheduling and Automation:** Scripts can be scheduled to run at specific times or intervals.
- **Support for Versioning:** Scripts can be versioned using Git or other version control systems.
- **Support for Scalability:** Scripts can be run on multiple machines or in the cloud.
- **Support for Command Line Arguments:** Scripts can take command line arguments for customization.
- **Support for Ressource Management:** Scripts can be run in the background without blocking the terminal or integrate cluster management tools (e.g. SLURM).
- **Support for Logging:** Scripts can log information to files for monitoring and debugging.
- **Support for Modularity:** Scripts can be modularized into functions and classes.

Why Notebooks are not enough

Program

In this lecture, we will learn how to manage a deep learning project using Python scripts. We will cover the following topics:

- **Project Structure:** We will learn how to organize a deep learning project into directories and files for better modularity and reusability.
- **Dependency Management:** We will learn how to manage dependencies using a virtual environment and a requirements file.
- **Running code as a script:** We will learn how to run Python code as a script from the command line.
- **Logging:** We will learn how to log information to files for monitoring and debugging.

Modularity

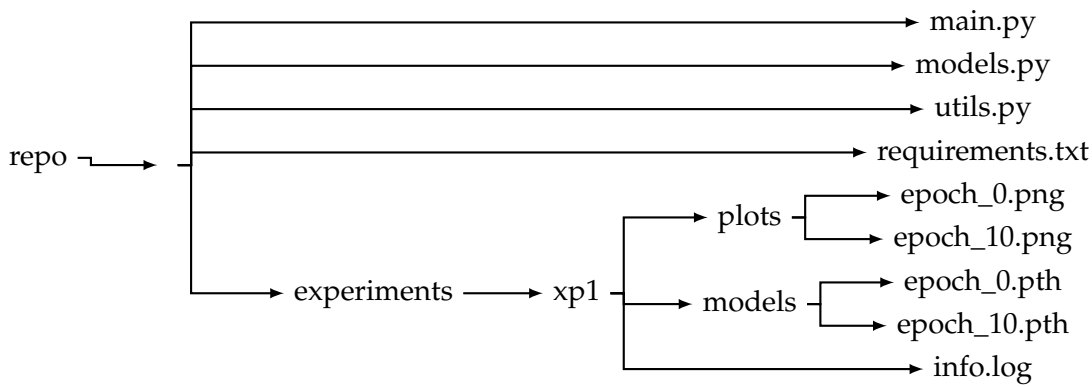
Why Code Should Be Modular

Writing modular code is essential for managing complex deep learning projects. It offers several advantages:

- **Reusability:** Modular code allows functions and classes to be reused across different parts of the project, reducing redundancy.
- **Maintainability:** With a clear structure, modular code is easier to maintain and update, especially in large projects.
- **Testing and Debugging:** Smaller, independent modules make it easier to test and debug specific parts of the codebase.
- **Collaboration:** A modular design enables multiple team members to work on different parts of the codebase simultaneously without conflicts.
- **Scalability:** Modular code can scale more effectively, as each component can be optimized or extended individually.

Modularity

Organizing Your Project



Explanation:

- `main.py`: Entry point for running experiments
- `models.py`, `agents.py`, etc.: Modular files for specific tasks
- `utils.py`: Utility functions and classes
- `requirements.txt`: List of dependencies
- `experiments/`: Separate folder for experiment outputs, logs, and saved models

Modularity

Key Principles of Modularity

To build a maintainable and scalable project, follow these modularity principles:

- **Separation of Concerns:** Each module should have a single responsibility (e.g., separate data processing, model architecture, and training code).
- **Encapsulation:** Keep functionality within defined boundaries to avoid cross-module dependencies.
- **Configurable Parameters:** Use configuration files for parameters, ensuring that code can be adapted to new experiments without modification.
- **Interface Design:** Define clear inputs and outputs for each module, making it easier to swap or update components.

Best practices for maintaining modularity in Python:

- **Use Functions and Classes:** Encapsulate related functionality in functions and classes for better organization.
- **Separate Configurations:** Store experiment parameters and settings in YAML or JSON files instead of hardcoding.
- **Follow a Naming Convention:** Consistent, descriptive names for modules, functions, and variables improve readability.
- **Documentation:** Document modules with docstrings and comments to clarify each part's role and dependencies.

Modularity

Defining the NeuralNet Class

Code example:

```
import torch.nn as nn

class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.block = None

    def forward(self, x):
        x = self.block(x)
        return x
```

Modularity

Extending NeuralNet for Different Architectures

Modularity allows for easy extension of base classes. Here, we extend `NeuralNet` to create specialized architectures:

```
class DenseNet(NeuralNet):
    def __init__(self, input_size, hidden_size, output_size):
        super(DenseNet, self).__init__()
        self.blocks = nn.ModuleList([nn.Linear(input_size, hidden_size),
                                     nn.ReLU(),
                                     nn.Linear(hidden_size, output_size)])

class ConvNet(NeuralNet):
    def __init__(self, input_channels, num_classes):
        super(ConvNet, self).__init__()
        self.conv1 = nn.Conv2d(input_channels=16,
                               kernel_size=3,
                               stride=1,
                               padding=1)
        self.fc = nn.Linear(16 * 28 * 28, num_classes)
        self.blocks = nn.ModuleList([self.conv1, nn.ReLU(), self.fc])
```

Local Environment Setup

Why Set Up a Local Environment?

Setting up a local environment is crucial for deep learning projects for several reasons:

- **Dependency Isolation:** Keeps project-specific dependencies separate from system-wide packages, preventing version conflicts.
- **Reproducibility:** Ensures consistent dependencies, allowing for reproducible experiments and easier collaboration.
- **Project Portability:** Makes it simpler to share the project with others, as they can replicate the exact environment.

Tools for Creating Local Environments:

- `virtualenv`: Lightweight environment creation tool
- `conda`: Environment management with additional package management capabilities

Local Environment Setup

Why Use Virtualenv?

`virtualenv` offers several advantages over `conda` for local environments:

- **Lightweight:** `virtualenv` is faster to create and has a smaller footprint compared to `conda`.
- **Python Version Agnostic:** Works with different versions of Python, regardless of the system's Python version.
- **Flexible Integration:** Compatible with `pip` and various Python packaging tools, making it easy to install dependencies from `requirements.txt`.

Creating a Virtual Environment:

```
# Install virtualenv if not already installed
pip install virtualenv

# Create a new virtual environment
virtualenv my_env
```

Local Environment Setup

Essential Commands for Virtualenv

Once the virtual environment is created, here are the key commands to manage it:

- **Activate the Environment:**

```
# On Windows
my_env\Scripts\activate

# On macOS/Linux
source my_env/bin/activate
```

- **Deactivate the Environment:**

```
deactivate
```

- **Install Dependencies:**

```
# Install a package
pip install <package_name>

# Install from a requirements file
pip install -r requirements.txt
```

Tip: Always activate the environment before installing packages.

`conda` provides several advantages over `virtualenv` for local environments:

- **Built-In Package Management:** `conda` can install packages directly.
- **Cross-Language Support:** Supports packages beyond Python, such as R or C libraries, making it versatile for data science projects.
- **Easier Dependency Resolution:** `conda` handles complex dependencies automatically, reducing conflicts.

Creating a Conda Environment:

```
# Create a new environment
conda create --name my_env python=3.8
```

Local Environment Setup

Essential Commands for Conda

Once the conda environment is created, here are the key commands to manage it:

- **Activate the Environment:**

```
conda activate my_env
```

- **Deactivate the Environment:**

```
conda deactivate
```

- **Install Dependencies:**

```
# Install a specific package
conda install <package_name>

# Install from an environment file
conda env update --file environment.yml
```

Tip: Use `environment.yml` files for reproducible setups when sharing your project.

The `argparse` library in Python allows for flexible command-line argument parsing, which is essential for configuring scripts without modifying code.

- **Customizable Scripts:** Enable users to specify parameters at runtime.
- **Improves Code Reusability:** Parameters can be adjusted dynamically, making the script adaptable across tasks.
- **User-Friendly Interface:** Provides help messages and default values, making it easier to use scripts.

Example:

```
import argparse
parser = argparse.ArgumentParser(description="My script description")
```

Running Experiments

Adding Arguments

Using `argparse`, we can add different types of arguments with various options:

```
# Adding arguments
parser.add_argument("--epochs", type=int, default=10,
                                help="Number of epochs")
parser.add_argument("--lr", type=float, default=0.001,
                                help="Learning rate")
parser.add_argument("--batch-size", type=int, default=32,
                                help="Batch size")
args = parser.parse_args()
```

Explanation:

- `-epochs`: Integer argument with a default value of 10.
- `-lr`: Float argument with a default value of 0.001.
- `-batch-size`: Integer argument to specify the batch size.

Running Experiments

Example Usage of Argparse

Once arguments are defined, we can access them via `args`:

```
# Accessing arguments
print(f"Train for {args.epochs} epochs with batch size {args.batch_size}")
```

Example of running the script with arguments:

```
python my_script.py
```

```
python my_script.py --epochs 20 --lr 0.01 --batch-size 64
```

```
python my_script.py --epochs 200 --lr 1e-5
```

This allows the script to be configured dynamically without changing the code.

Running Python scripts from the terminal on Mac and Linux provides flexibility and resource control:

- **Basic Command:**

```
python my_script.py --epochs 10 --lr 0.001
```

- **Background Execution:** Running a script in the background frees up the terminal for other tasks.

```
python my_script.py --epochs 10 --lr 0.001 &
```

- **Monitoring Background Jobs:** Use `jobs` to list background jobs and `fg` to bring them to the foreground.
- **Importance of Background Execution:** Ideal for long-running tasks, especially on servers or remote machines.

Running Python scripts from the Command Prompt or PowerShell on Windows is straightforward:

- **Basic Command:**

```
python my_script.py --epochs 10 --lr 0.001
```

- **Running in Background:**
  - Use `pythonw` to launch a new command window:

```
pythonw my_script.py --epochs 10 --lr 0.001
```

  - Alternatively, use `start /B` for background execution without opening a new window:

```
start /B python my_script.py --epochs 10 --lr 0.001
```

- **Importance of Background Execution:** Allows other terminal tasks and is useful for long-running processes.

Cron is a task scheduler for Unix-based systems that runs scripts or commands at scheduled intervals.

- **Cron Syntax:** The format is `min hour day month day_of_week command`.
- Example:

```
# Run a script every day at midnight
0 0 * * * /usr/bin/python /path/to/my_script.py --epochs 10
```

Alternative Tools:

- **MacOS:** Use `launchd`, a built-in tool for scheduling tasks.
- **Windows:** Use the Task Scheduler, which provides a GUI for managing scheduled tasks.

Why Scheduling Matters: Automating tasks saves time and allows for consistent, timely execution of regular jobs (e.g., retraining a model).

Monitoring Experiments

Why Monitoring Matters

Monitoring experiments is essential in deep learning projects for several reasons:

- **Track Progress:** Provides insights into training dynamics, allowing for better optimization and tuning.
- **Diagnose Issues Early:** Identify potential issues (e.g., vanishing gradients, overfitting) quickly, saving time and resources.
- **Reproducibility and Accountability:** Logs and metrics enable reproducibility and allow other collaborators to understand the experiment setup and results.
- **Model Comparisons:** Facilitates comparison of different runs and hyperparameters to identify the best-performing model.

Monitoring Experiments

Tools for Monitoring Experiments

There are various tools for effective monitoring and tracking:

- **Basic Logging:** Use the `logging` module to track metrics, errors, and messages within code.
- **JSONL Files:** Store logs as structured data, allowing for detailed tracking and easy parsing.
- **TensorBoard:** Provides a visual interface for monitoring metrics, loss curves, and other parameters in real-time.
- **Weights & Biases:** A powerful tool for experiment tracking and visualization, allowing for collaborative analysis and sharing.

These tools support different needs, from lightweight solutions (logging) to more robust experiment tracking (TensorBoard, Weights & Biases).

The `logging` module in Python is a flexible way to capture and track key information during training:

- **Purpose:** Logs provide structured, timestamped messages about code execution, helping with debugging and monitoring.
- **Logging Levels:** `logging` supports different log levels:
  - **INFO:** General messages about code execution and key steps.
  - **WARNING:** Messages for potential issues that are not critical.
  - **ERROR:** Logs errors that may stop part of the execution.
- **Formats:** Logs can include timestamps, filenames, and message content for better traceability.

Monitoring Experiments

Setting Up a Logger

Here's how to set up a basic logger in Python:

```
import logging

# Configure the logger
logging.basicConfig(
    filename='experiment.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log messages
logging.info("Training started")
logging.warning("Batch size might be too large")
logging.error("Out of memory error")
```

Explanation:

- `filename`: Saves logs to a file for permanent record.
- `level`: Sets the minimum log level (INFO, WARNING, ERROR) for capture.
- `format`: Configures the log format with timestamp, level, and message.

Monitoring Experiments

Saving Metrics and Plots

For effective monitoring, it's important to save, rather than print, key metrics and visualizations:

- **Metrics Tracking:** Save loss, accuracy, and other key metrics to log files or structured formats (e.g., JSON, JSONL) for later analysis.
- **Plotting Loss Curves:** Generate and save plots of loss, accuracy, and other metrics during training, rather than printing them in the terminal.
- **Tools for Visualization:** Use tools like `matplotlib` to save plots, or real-time dashboards such as:
  - **TensorBoard:** Visualize training metrics live for a smoother debugging experience.
  - **Weights & Biases:** Track metrics, visualize experiments, and compare runs.

Tip: Automate saving plots and metrics to ensure nothing is missed during training.

Git is essential for managing code in deep learning projects for several reasons:

- **Version Control:** Git tracks changes over time, enabling easy rollback to previous versions and enhancing reproducibility.
- **Collaboration:** Git allows multiple contributors to work on the same project, with tools to merge code and resolve conflicts.
- **Backup and Security:** Code stored on remote servers (e.g., GitHub, GitLab) provides a secure, backed-up version of the project.
- **Experiment Tracking:** Branching allows you to test experimental features or new ideas without affecting the main code.

Git is a distributed version control system where every version of the repository is saved locally and remotely (e.g., on GitHub):

- **Local Repository:** Changes are tracked locally on your machine, allowing for offline work and complete version history.
- **Remote Repository:** A server (e.g., GitHub, GitLab) stores your code and history, making it accessible for collaboration.
- **Commit History:** Every change is saved as a "commit," building a history of modifications for easy review.

Each commit is like a snapshot of the project at a specific point in time, allowing for seamless version tracking.

Versioning your Code

Essential Git Commands

Here are some fundamental Git commands to manage your repository:

- **git pull:** Fetch updates from the remote repository and integrate them into your local repository.

```
git pull origin main
```

- **git add:** Stage changes for the next commit.

```
git add <file_name>  # Add a specific file
git add .            # Add all changed files
```

- **git rm:** Remove a file from the repository.

```
git rm <file_name>
```

Committing and pushing changes are key steps in Git:

- **git commit:** Save a snapshot of the staged changes to the local repository with a message describing the change.

```
git commit -m "Describe the change"
```

- **git push:** Upload your commits from the local repository to the remote repository.

```
git push origin main
```

Regular commits with clear messages help track progress and make it easier to understand code changes.

Git branches enable experimentation and collaboration:

- **Branches:** Create separate lines of development to test features without affecting the main codebase.

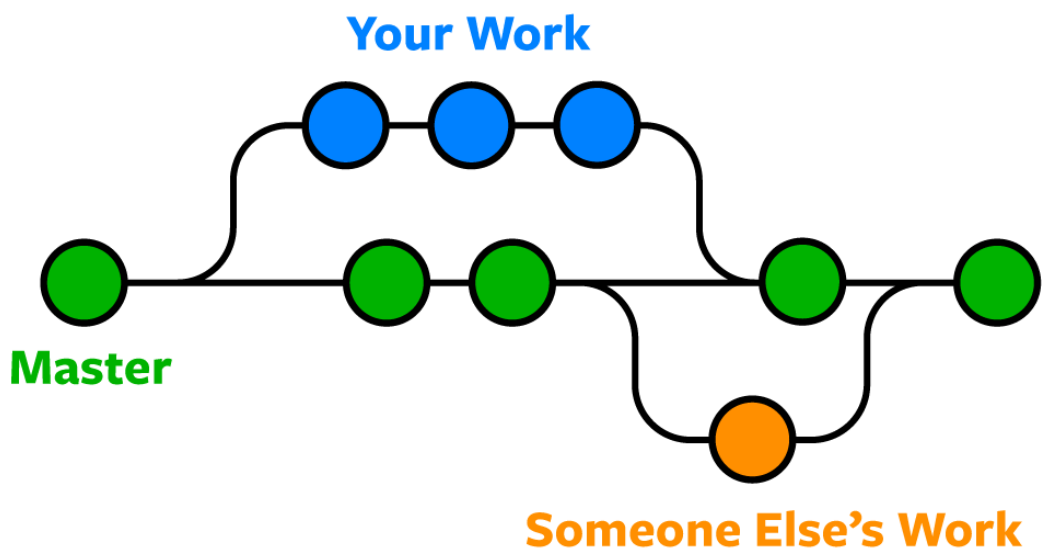
```
git branch new-feature       # Create a branch
git checkout new-feature     # Switch to the branch
```

- **Merging:** Combine changes from a branch back into the main branch once the feature is ready.

```
git checkout main
git merge new-feature
```

- **Pull Requests (PRs):** PRs are used on platforms like GitHub to review and approve code before merging it into the main branch.

The following diagram represents a typical Git workflow with branching, committing, and pushing changes.

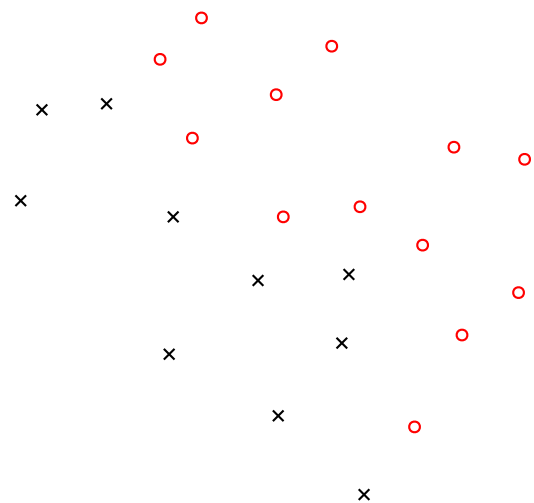


Part VI

Project Presentation

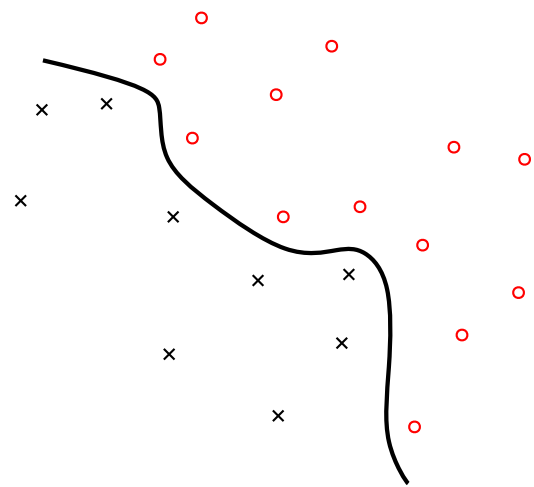
Mini Project 2: Adversarial Attacks

A dataset



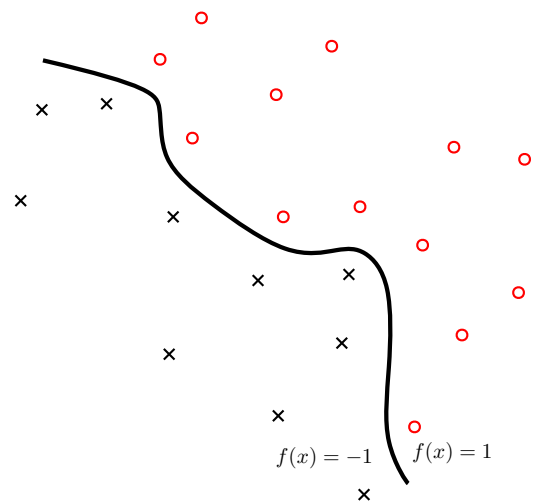
Mini Project 2: Adversarial Attacks

A decision boundary



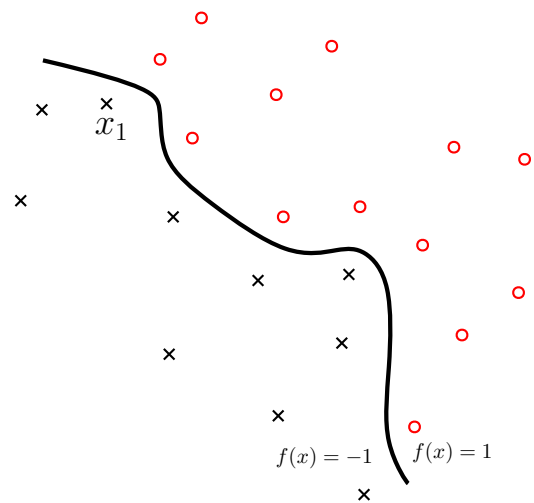
Mini Project 2: Adversarial Attacks

A classifier



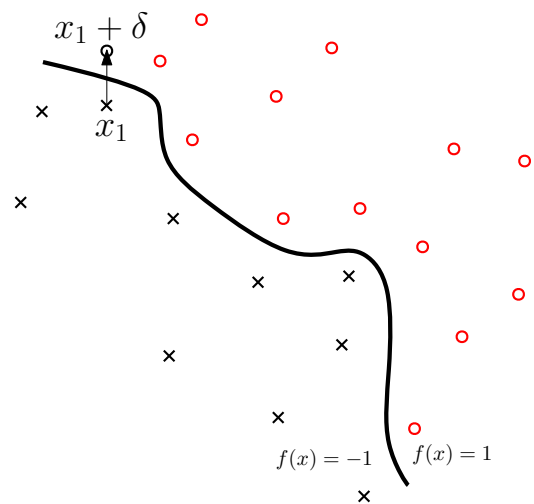
Mini Project 2: Adversarial Attacks

Choosing a data point



Mini Project 2: Adversarial Attacks

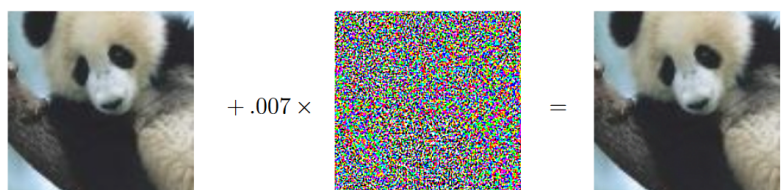
Perturbing the data point



What if δ is imperceptible ?

Mini Project 2: Adversarial Attacks

Adversarial Attacks in Image recognition



Source : *Explaining and Harnessing Adversarial Examples*, Goodfellow et al, ICLR 2015.

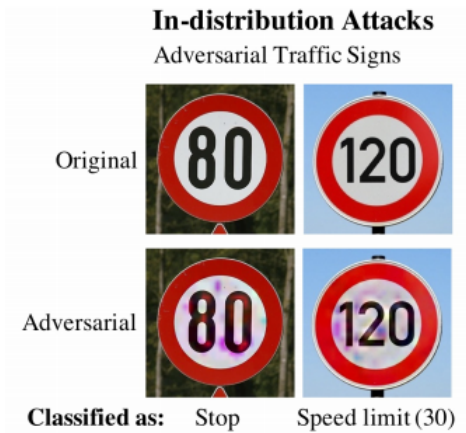


Figure. Adversarial traffic signs (Sitawarin, Bhagoji et al., 2018)

To be imperceptible, the norm of the perturbation is bounded

We define an $\epsilon \in \mathbb{R}$ such that $\|\delta\|_p \leq \epsilon$.
In practice, we use ℓ_2 and ℓ_∞ norm to bound the perturbation.

Generating a adversarial example

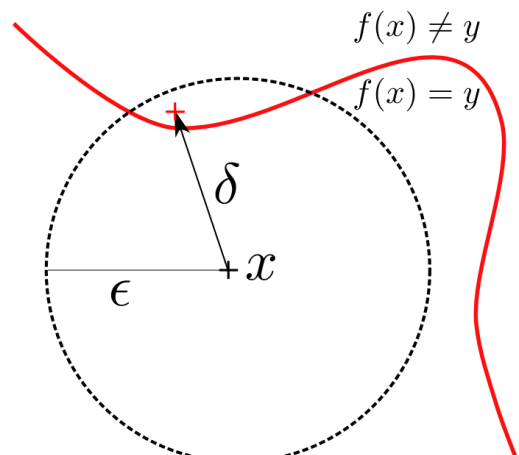
Let $f : \mathbb{R}^d \to \mathcal{Y}$ be a classifier. Given an example $x \in \mathcal{X} \subset \mathbb{R}^d$ and its true label $y \in \mathcal{Y}$, the goal is to find $\delta \in \mathbb{R}^d$ such that :

Untargeted attacks
$\|\delta\|_p \leq \epsilon$ and $f(x + \delta) \neq y$

Targeted attacks
$\|\delta\|_p \leq \epsilon$ and $f(x + \delta) = t$ with $t \neq y$

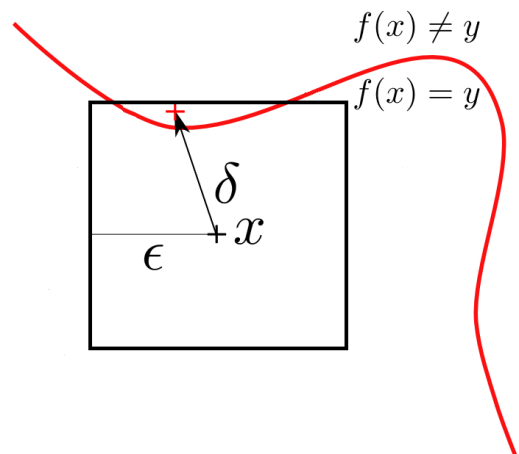
Mini Project 2: Adversarial Attacks

Generating an adversarial example with ℓ_2-norm



Mini Project 2: Adversarial Attacks

Generating an adversarial example with ℓ_∞-norm



FGSM

The Fast Gradient Sign Method (FGSM) is an attack scheme that uses the gradients of the neural network to create adversarial examples, it is defined as:

$$x_{\text{adv}} = x + \epsilon \cdot \text{sign}(\nabla_x L(\theta, x, y))$$

Source:
Explaining and Harnessing Adversarial Examples, Goodfellow et. al, ICLR 2015.

ℓ_2-PGD

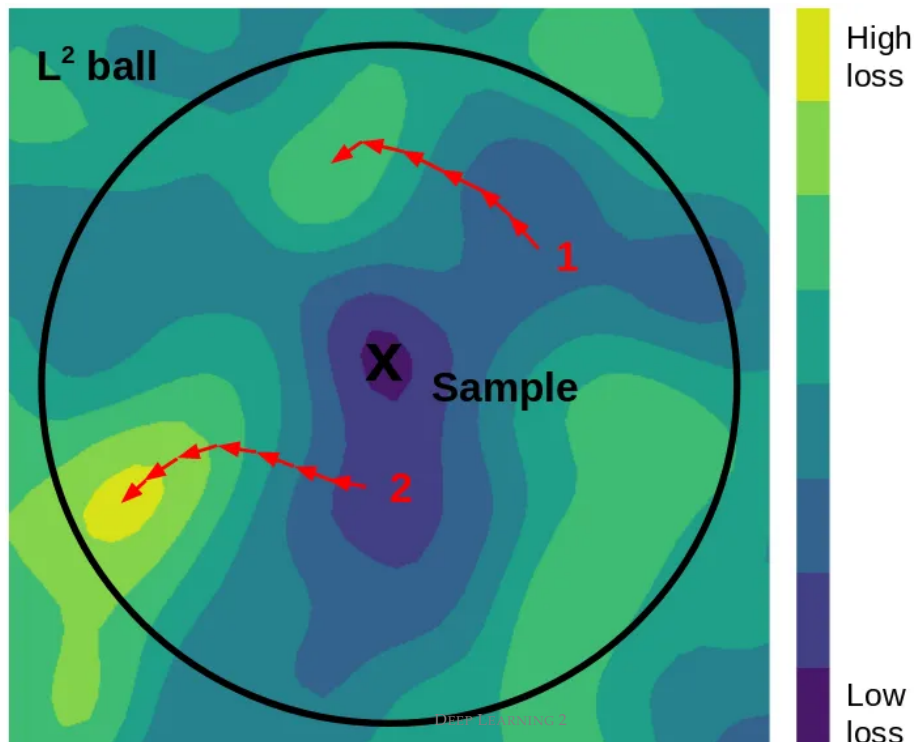
ℓ_2-PGD is an iterative method similar to ℓ_∞-PGD, but it constrains the perturbation to an ℓ_2-norm ball. The iteration is defined as follows:

1. $x_0 \leftarrow x$
2. repeat n times :
$$x_{t+1} = \Pi_{B_2(x,\epsilon)} \left(x_t + \eta \nabla_x L_\theta(x_t, y) \right)$$

Source:
Towards Deep Learning Models Resistant to Adversarial Attacks, Madry et. al, ICLR 2018.

Mini Project 2: Adversarial Attacks

ℓ_2-PGD Attack



ℓ_∞-PGD

ℓ_∞-PGD is an iterative method that constructs the perturbed data as follows :

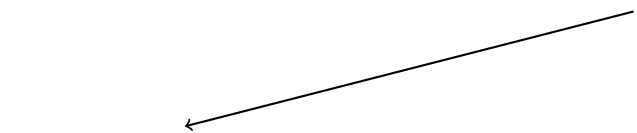
1. $x_0 \leftarrow x$
2. repeat n times :
$$x_{t+1} = \Pi_{B_\infty(x,\epsilon)}\left(x_t + \eta sign(\nabla_x L_\theta(x_t, y))\right)$$

Source:
Towards Deep Learning Models Resistant to Adversarial Attacks, Madry et. al, ICLR 2018.

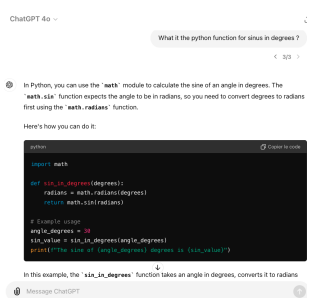
Generative Models

Context

In the past few years, generative models have made significant progress in various domains of application.



Text Generation



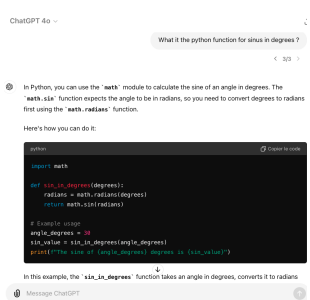
eg: Code Completion

Generative Models

Context

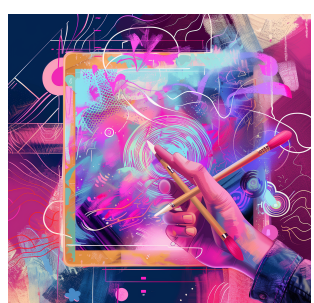
In the past few years, generative models have made significant progress in various domains of application.

Text Generation



eg: Code Completion

Image/Video Generation

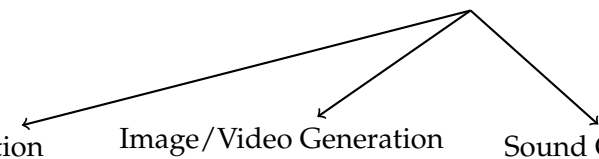


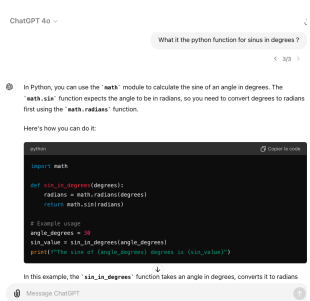
eg: Media Industry

Generative Models

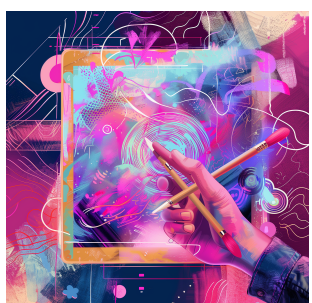
Context

In the past few years, generative models have made significant progress in various domains of application.





eg: Code Completion



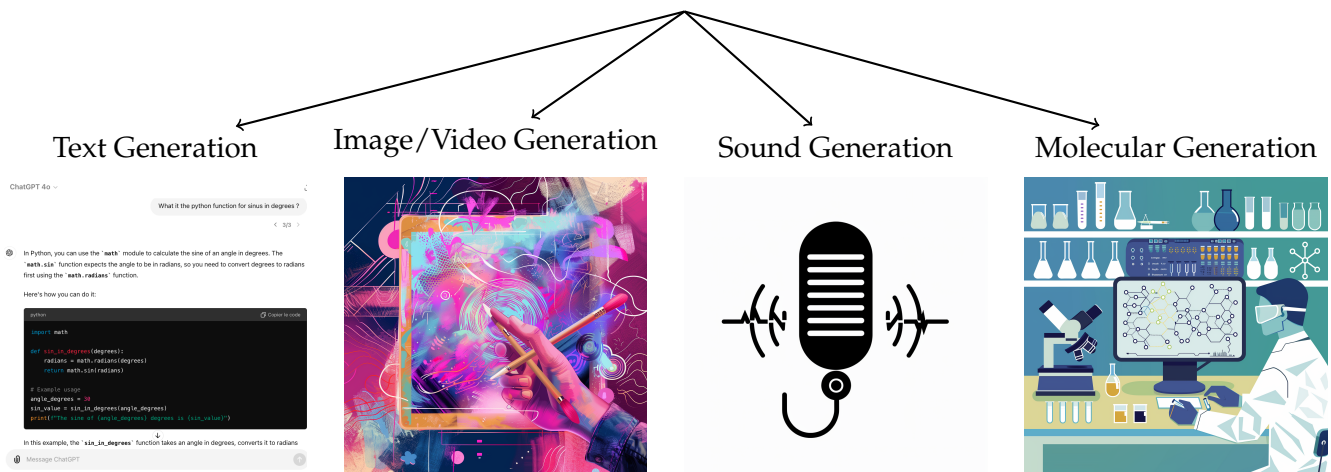
eg: Media Industry

eg: Speech Synthesis

Generative Models

Context

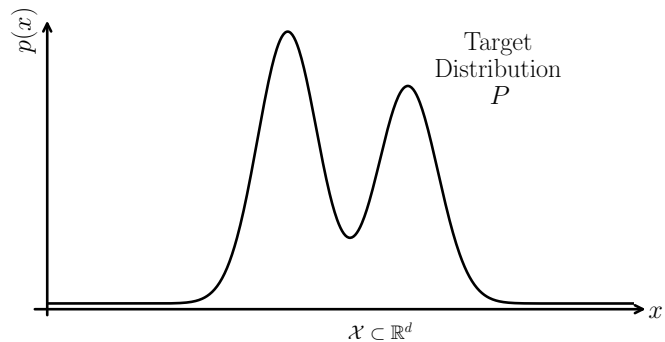
In the past few years, generative models have made significant progress in various domains of application.



eg: Code Completion | eg: Media Industry | eg: Speech Synthesis | eg: Drug Discovery

Generative Models

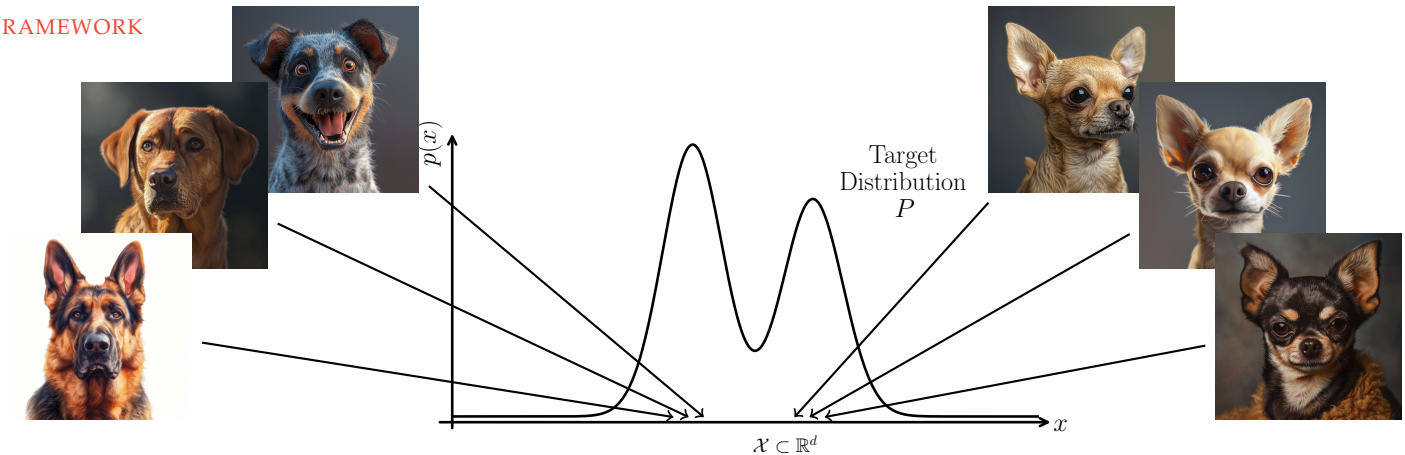
Framework



- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.

Generative Models

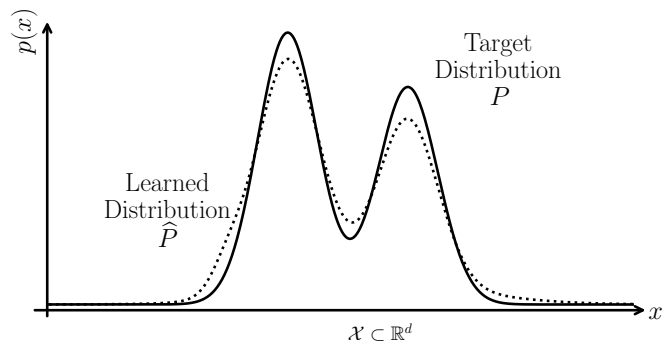
Framework



- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.

Generative Models

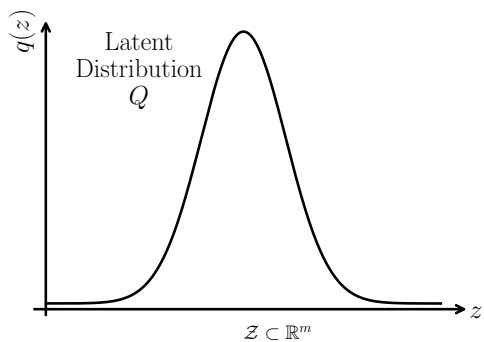
Framework

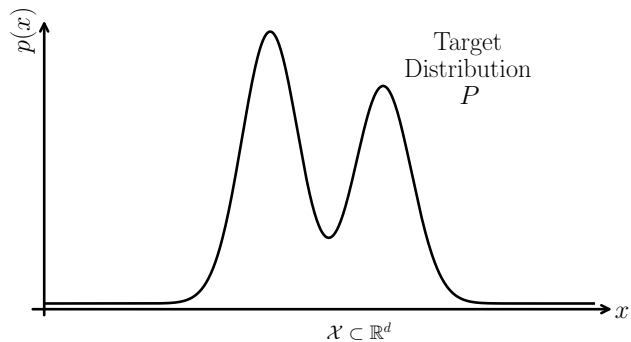


- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.
- Goal: Learn a *parameterized distribution* $\widehat{P}$ that approximate P:

Generative Models

Framework

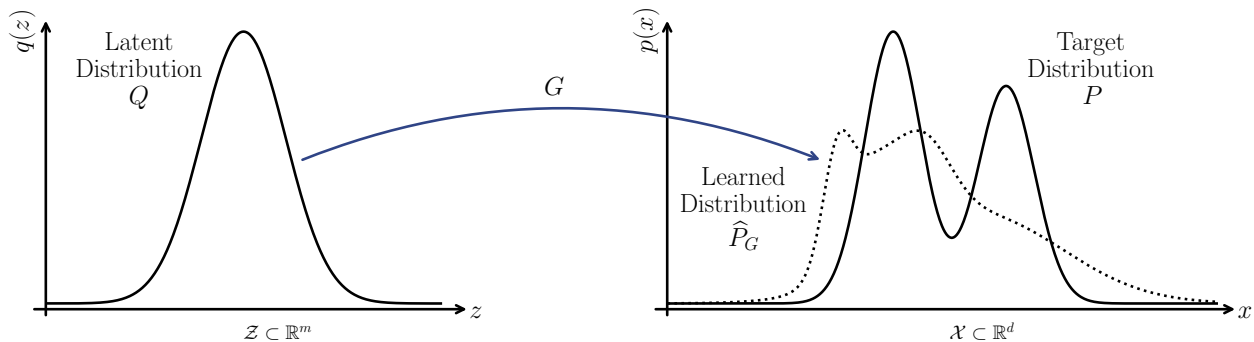




- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.
- Goal: Learn a *parameterized distribution* $\widehat{P}$ that approximate P:
  1. Consider a distribution Q in *a latent space* $\mathcal{X} \subset \mathbb{R}^m$, usually $\mathcal{N}(0, I_m)$.

Generative Models

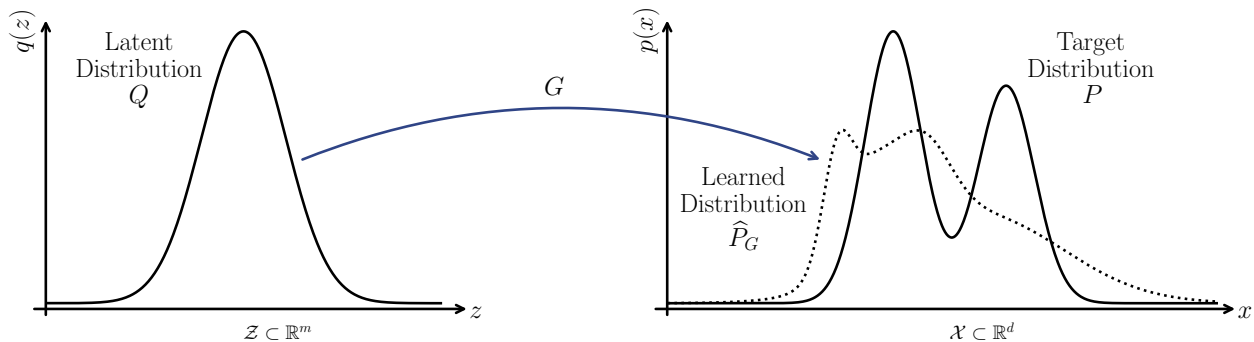
Framework



- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.
- Goal: Learn a *parameterized distribution* $\widehat{P}_G$ that approximate P:
  1. Consider a distribution Q in *a latent space* $\mathcal{X} \subset \mathbb{R}^m$, usually $\mathcal{N}(0, I_m)$.
  2. Take *a generator model* G represented by a neural network. Take $\widehat{P}_G = G\#Q$.

Generative Models

Framework

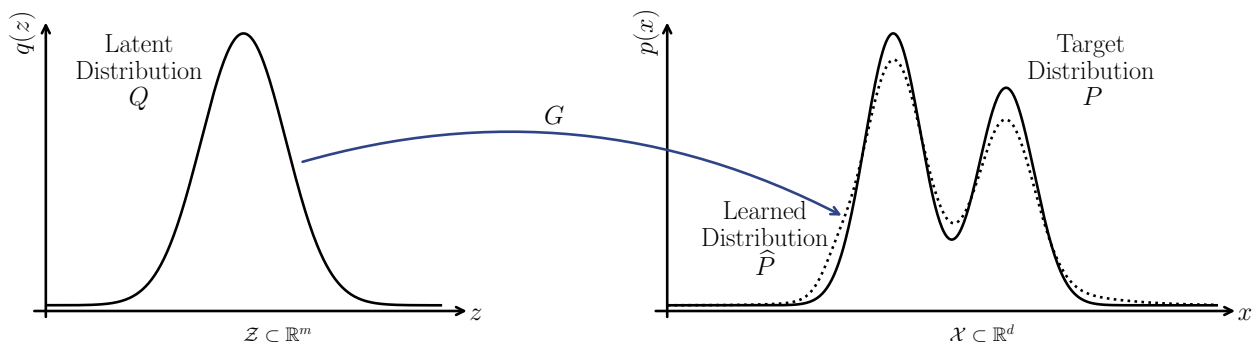


- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.
- Goal: Learn a *parameterized distribution* $\widehat{P}_G$ that approximate P:
  1. Consider a distribution Q in *a latent space* $\mathcal{X} \subset \mathbb{R}^m$, usually $\mathcal{N}(0, I_m)$.
  2. Take *a generator model* G represented by a neural network. Take $\widehat{P}_G = G\#Q$.
  3. Compute G^{opt} that minimize *a dissimilarity measure* D between P and $\widehat{P}_G$:

$$G^{\text{opt}} = \arg\min_G D(P, \widehat{P}_G)$$

Generative Models

Framework



- Assumption: There is an unknown *target distribution* P in $\mathcal{X} \subset \mathbb{R}^d$.
- Goal: Learn a *parameterized distribution* $\widehat{P}$ that approximate P:
  1. Consider a distribution Q in *a latent space* $\mathcal{X} \subset \mathbb{R}^m$, usually $\mathcal{N}(0, I_m)$.
  2. Take *a generator model* G represented by a neural network. Take $\widehat{P} = G\#Q$.
  3. Compute G^{opt} that minimize *a dissimilarity measure* D between P and $\widehat{P}$:

$$G^{\text{opt}} = \arg\min_G D(P, \widehat{P})$$

Generative Models

In Practice

DALL·E 2 (2023)

?

Midjourney v5 (2023)

Prompt: *A dog playing with a child.*

Generative Models

In Practice

DALL·E 2 (2023)

$\neq$

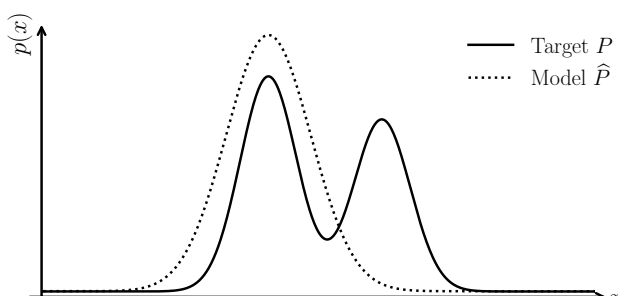
Midjourney v5 (2023)

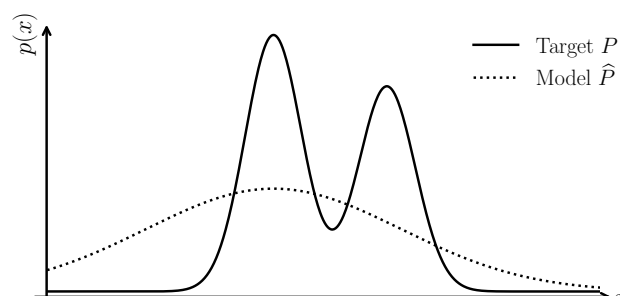
Prompt: *A dog playing with a child.*

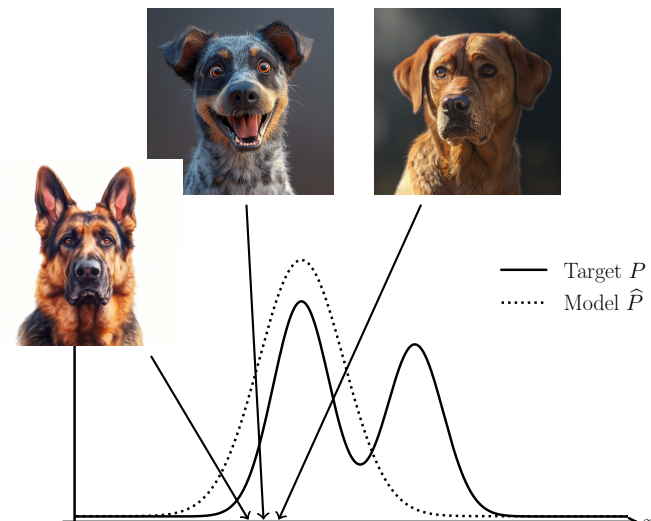
Generative Models

In Practice

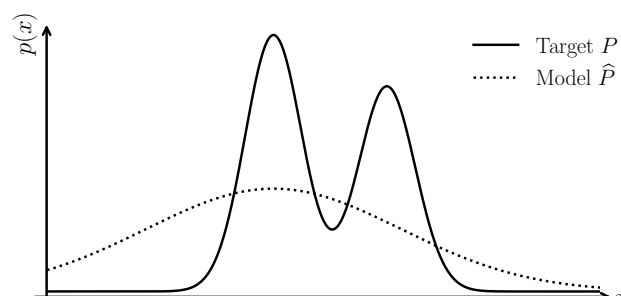
$$\widehat{P} \neq P$$



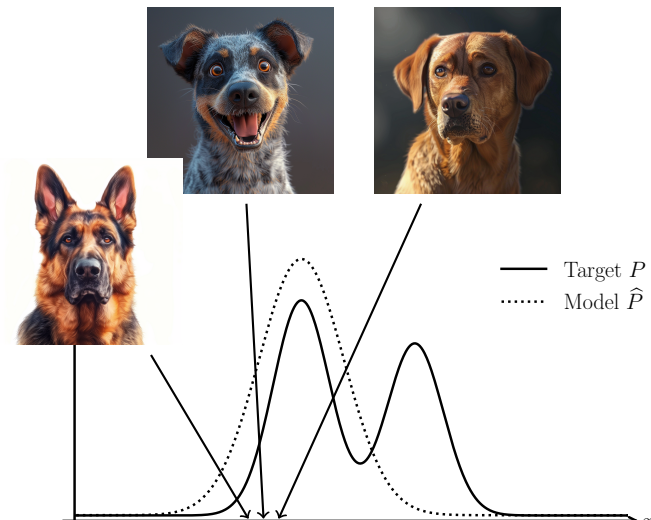




$\widehat{P} \neq P$

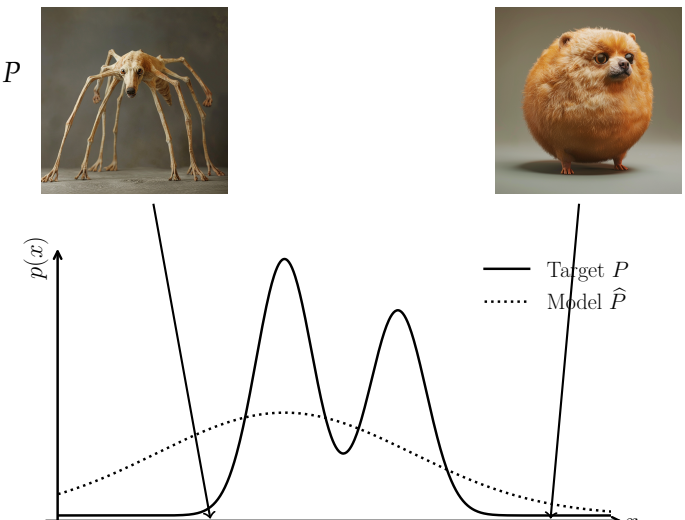


Low Diversity



Low Diversity

$\widehat{P} \neq P$



Low Quality

Precision and Recall for Generative Models

Metrics to evaluate quality and diversity

To assess models, we use the notion of Precision and Recall, inspired from Information Retrieval:

Quality

Diversity

Precision and Recall for Generative Models

Metrics to evaluate quality and diversity

To assess models, we use the notion of Precision and Recall, inspired from Information Retrieval:

Quality

$\downarrow$

Precision

$=$

What proportion of generated samples are realistic?

Diversity

Precision and Recall for Generative Models

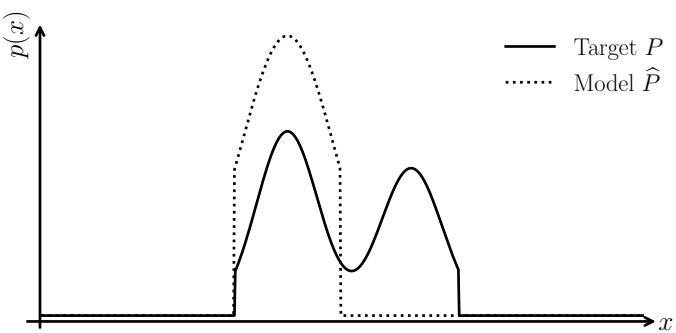
Metrics to evaluate quality and diversity

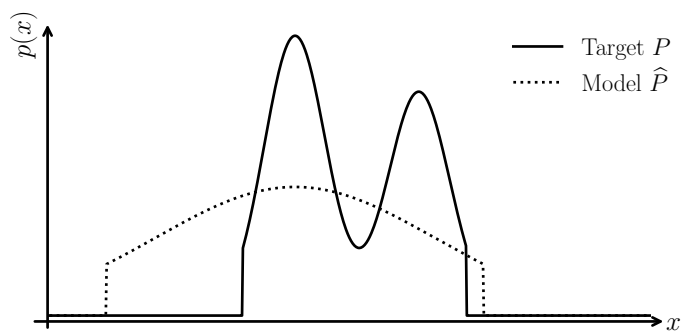
To assess models, we use the notion of Precision and Recall, inspired from Information Retrieval:

Quality	Diversity
$\downarrow$	$\downarrow$
Precision	Recall
$=$	$=$
What proportion of generated samples are realistic?	What proportion of real samples can be generated?

Precision and Recall for Generative Models

For finite support





Precision and Recall for Generative Models

For finite support

Definition 2.1 (Support-Based Precision and Recall - [6].)

For any distributions $P \in \mathcal{P}(\mathcal{X})$ and $\widehat{P} \in \mathcal{P}(\mathcal{X})$, we say that the distribution P has precision $\bar{\alpha}$ at recall $\bar{\beta}$ with respect to $\widehat{P}$ if

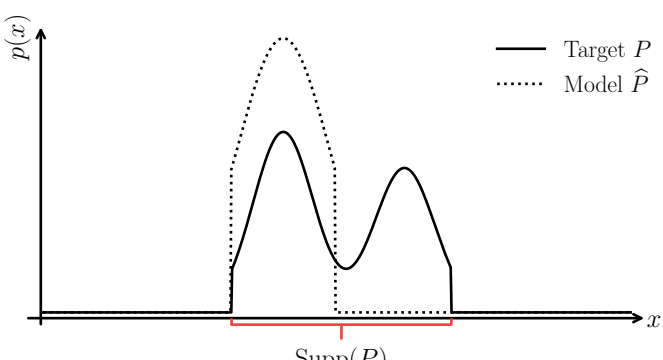
$$\bar{\alpha} = \widehat{P}(\text{Supp}(P)) \quad et \quad \bar{\beta} = P(\text{Supp}(\widehat{P})). \tag{2}$$

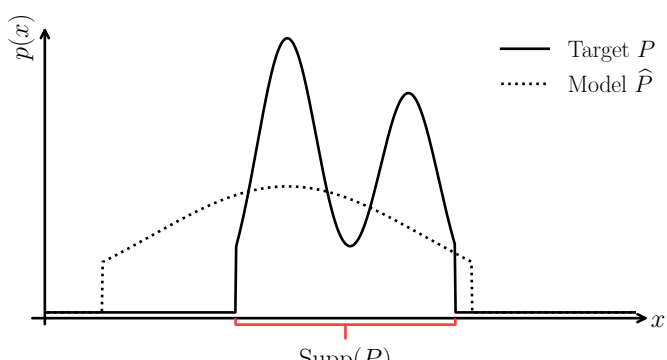
Precision and Recall for Generative Models

For finite support

Precision for finite support is the proportion of generated data that lies on the support of the real data:

$$\bar{\alpha} = \widehat{P}(\text{Supp}(P)).$$



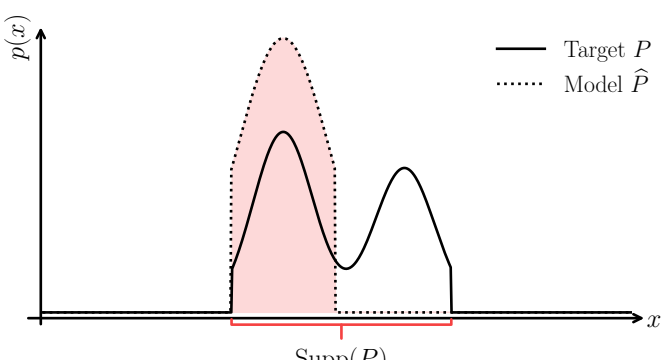


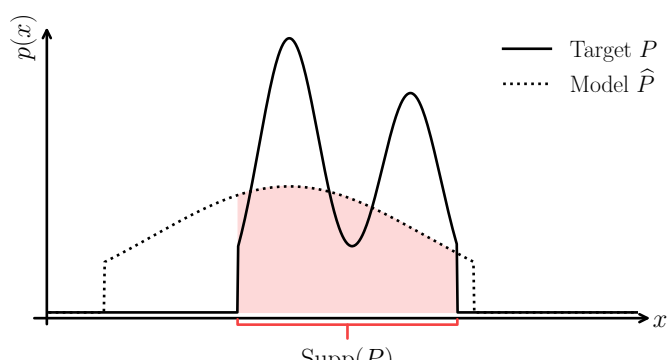
Precision and Recall for Generative Models

For finite support

Precision for finite support is the proportion of generated data that lies on the support of the real data:

$$\bar{\alpha} = \widehat{P}(\text{Supp}(P)).$$



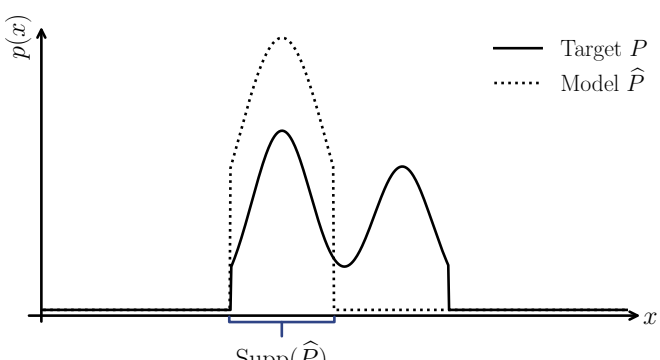


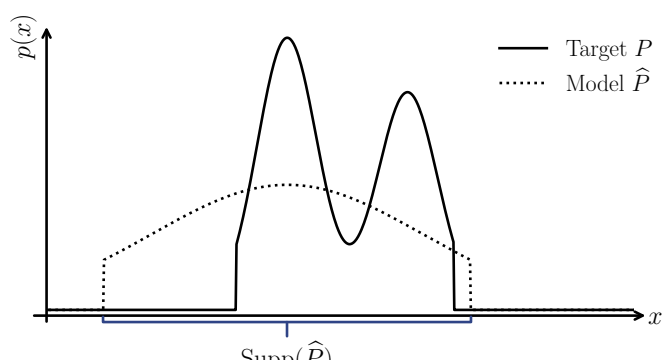
Precision and Recall for Generative Models

For finite support

Recall for finite support is the proportion of the support of the real data that is covered by the generated data:

$$\bar{\beta} = P(\text{Supp}(\widehat{P})).$$



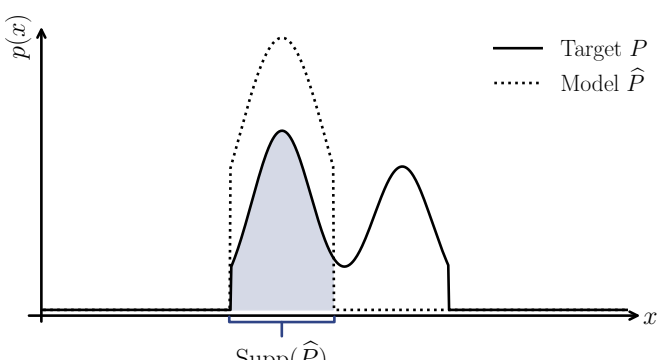


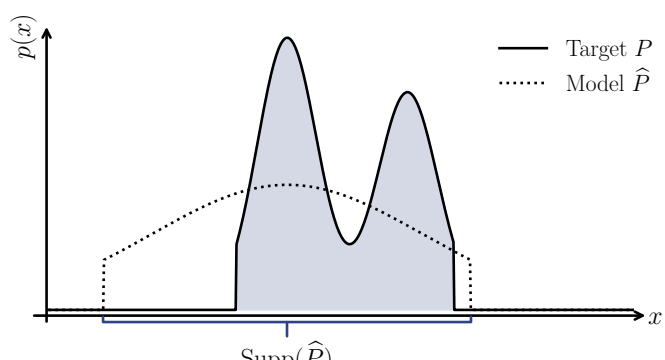
Precision and Recall for Generative Models

For finite support

Recall for finite support is the proportion of the support of the real data that is covered by the generated data:

$$\bar{\beta} = P(\text{Supp}(\widehat{P})).$$





Precision and Recall for Generative Models

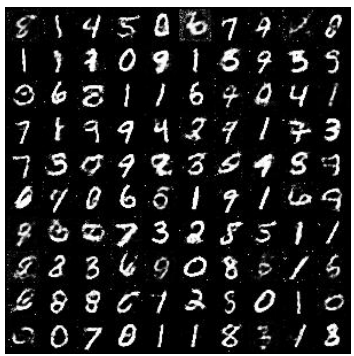
In Practice

MNIST Dataset [7]

Precision and Recall for Generative Models

In Practice

Low Precision High Recall



Precision: 0.54 Recall: 0.91

MNIST Dataset [7]

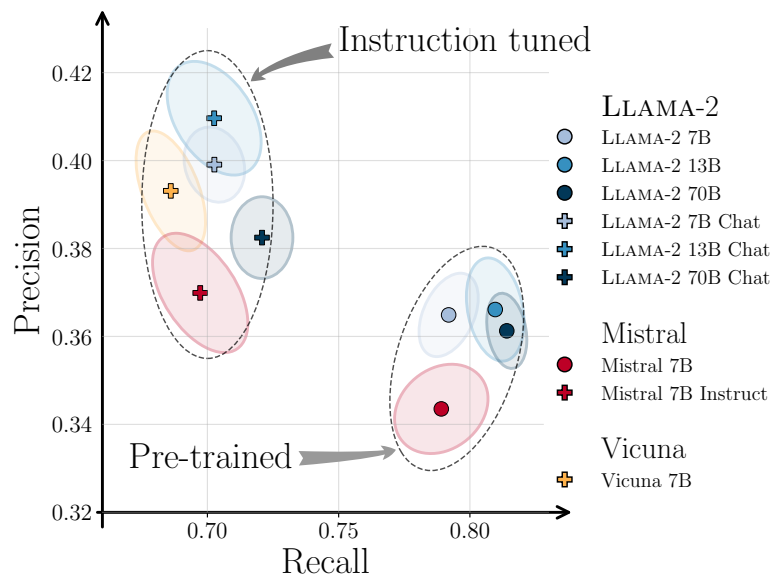
High Precision Low Recall

Precision: 0.80 Recall: 0.70

Precision and Recall for Generative Models

For LLMs

On open-ended generation, the quality and diversity of LLMs can also be evaluated using Precision and Recall: [2]



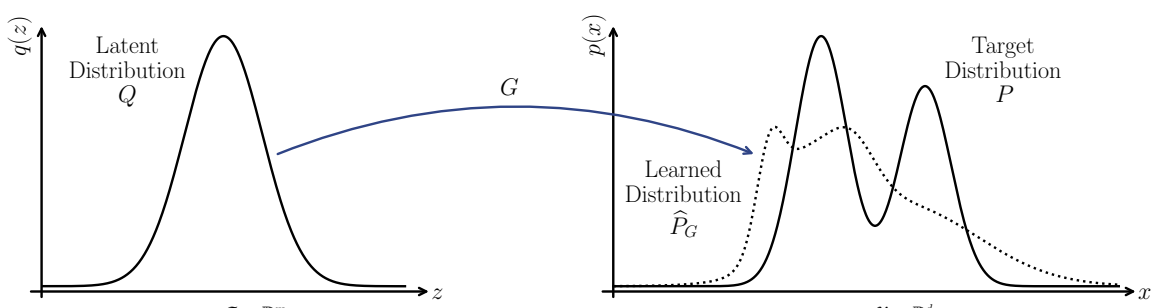
Generative Adversarial Networks

Original Framework

- Let $G : \mathcal{Z} \to \mathcal{X}$ be a generator model parameterized by a neural network.
- Let $D : \mathcal{X} \to [0, 1]$ be a discriminator model parameterized by a neural network.

The original GAN framework [3] is defined by the following optimization problem:

$$\min_G \max_D \; \mathbb{E}_{x \sim P} \left[\log D(x) \right] + \mathbb{E}_{x \sim \widehat{P}_G} \left[\log(1 - D(x)) \right]. \tag{3}$$



Generative Adversarial Networks

Original Framework

- Let $G : \mathcal{Z} \to \mathcal{X}$ be a generator model parameterized by a neural network.
- Let $D : \mathcal{X} \to [0, 1]$ be a discriminator model parameterized by a neural network.

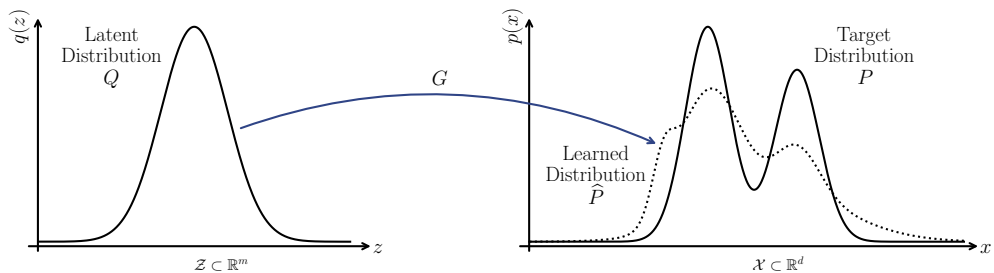
The original GAN framework [3] is defined by the following optimization problem:

$$\min_G \max_D \; \mathbb{E}_{x \sim P} \left[\log D(x) \right] + \mathbb{E}_{x \sim \widehat{P}_G} \left[\log(1 - D(x)) \right]. \tag{3}$$

Tuning Precision and Recall in Generative Models

Truncation

Hard Trunctation
Karras et al. [4]



Soft Trunctation
Kingma and Dhariwal [5]

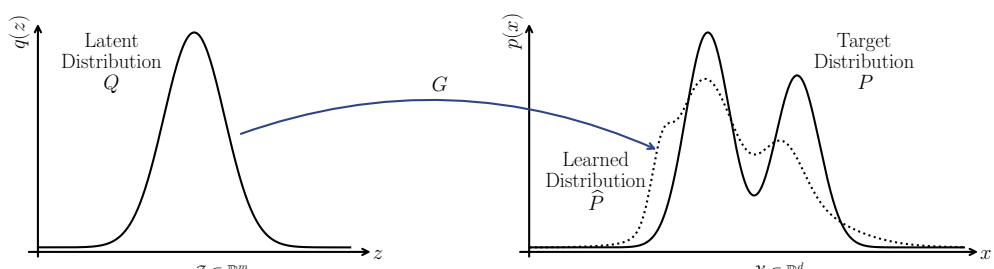


Figure. From left to right: $\psi = 0.0$, $\psi = 0.3$ $\psi = 0.7$ $\psi = 1.0$.

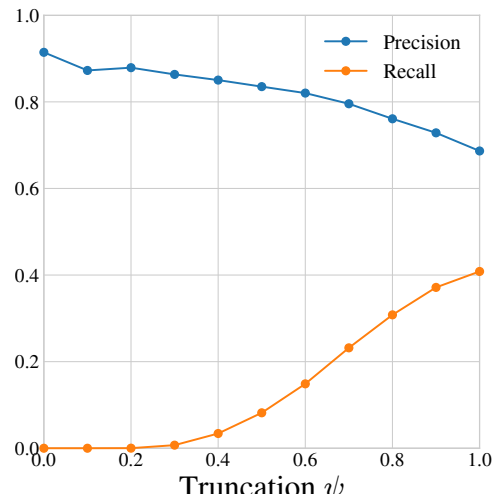


Figure. Source: [6]

(a) $\psi = 0.04$ (b) $\psi = 0.5$ (c) $\psi = 1.0$ (d) $\psi = 2.0$

Figure. Soft-Truncation on BigGAN. Source:[1].

Final Project

Tuning Quality and Diversity in GANs

- Goal: Implement a GAN architecture to generate samples from a given dataset.
- Dataset: MNIST
- Code: Use the repository given in the Project
- Enrolling: GitHub Classroom `https://classroom.github.com/a/lj2_ipKW`

Possible improvements

- f-GANs
  1. f-GAN: Training Generative Neural Samplers using Variational Divergence Minimization
  2. Precision-Recall Divergence Optimization for Generative Modeling with GANs and Normalizing Flows
- WGAN
  1. Wasserstein GAN
- Rejection Sampling
  1. Discriminator Rejection Sampling
  2. Metropolis-Hastings Generative Adversarial Networks
  3. Optimal Budgeted Rejection Sampling for Generative Models
- Latent Rejection sampling
  1. Latent reweighting, an almost free improvement for GANs
- Gradient ascent
  1. Discriminator optimal transport
  2. Refining Deep Generative Models via Discriminator Gradient Flow
  3. Your GAN is Secretly an Energy-based Model and You Should use Discriminator Driven Latent Sampling
- Classifier guidance generation
  1. MMGAN: Generative Adversarial Networks for Multi-Modal Distributions

Final Project: Generative Adversarial Networks

Final Project: Tuning Quality and Diversity in GANS

Specifics:

- For fair comparison, use the same *generator* architecture as the one in the project.
- Your code will be pulled and test every 24 hours (at 02:00 AM).
- You have to choose one of the methods (or any other method that makes sense to you) and implement it.

During the presentation, you will have to:

- Present the paper you chose.
- Present how you adapted the method to the project.
- Explain your results.
- Explain the limitations of your method.

You will be graded:

- on the quality of the presentation.
- on how well you understood the method.
- on how well you adapted the method to the project.

You will not be graded on:

- Your rank on the platform.
- The quality of your code.

Final Project: Generative Adversarial Networks

Report and Presentation

The work is:

- Individual.
- Due on the 10th of December at 8:15 AM.

The report and the presentation:

- The report should be a PDF file to push on the Github repository.
- The report should be at most 5 pages long. (excluding references and title page)
- The report should be written in English.
- The presentation should be a PDF file to push on the Github repository.
- The presentation must be 15 minutes long (hard limit) + 5 minutes for questions.
- The presentation should be in English or French.

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.