# Deep Learning

## Introduction, simple architectures (MLPs) and autodiff

Lessons: **Kevin Scaman**
TPs: Paul Lerner

# Practical details

## Timeline

- **Dates:** Every Friday afternoon from 09/02 to 22/03 (except 23/02)
- **Room:** 2012 (lessons), 2014 (practicals)
- **Format:** 4 lessons, 2 practicals

## Validation

- 1 homework. Due date: 22/03.
- 1 final project. Due date: 29/03.
- **Final grade:** $(H + P)/2$

## Communication

- Email (`kevin.scaman@inria.fr`)

# Overview of the course

## Lessons

1. **Introduction, simple architectures (MLPs) and autodiff** 09/02
2. Training pipeline, optimization and image analysis (CNNs) 16/02
3. Sequence regression (RNNs), stability and robustness 08/03
4. Generative models in vision and text (Transformers, GANs) 15/03

## To go further

- **Dataflowr:** Pytorch implementation. `https://dataflowr.github.io/`
- **The little book of DL:** `https://fleuret.org/public/lbdl.pdf`
- **Deep Learning book:** overview of Deep Learning. `www.deeplearningbook.org/`
- **Distill journal:** Nice visualizations. `https://distill.pub/`

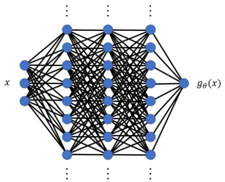
# What is Deep Learning?

# What is Deep Learning?

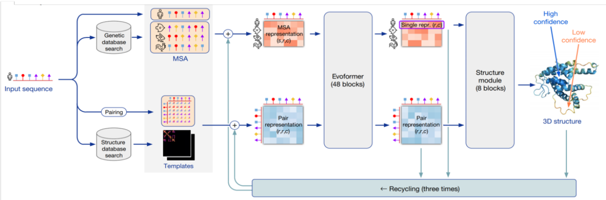
## First, what are neural networks?

- The notion changed over the last 8 decades...!
- From early neural networks imitating real neurons...
- To highly complex architectures with multiple sub-modules.

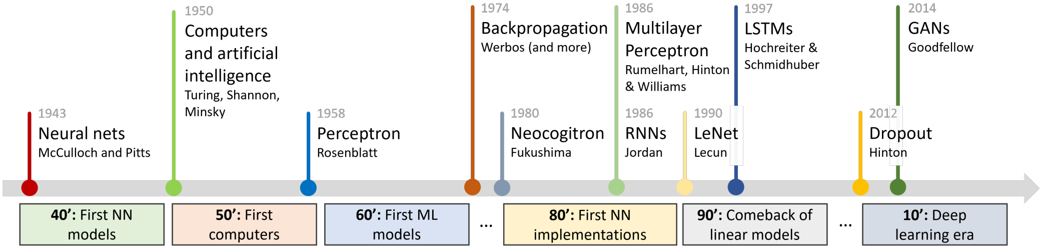
Multi-Layer Perceptron
(Rumelhart, Hinton, Williams, 75)




AlphaFold
(Jumper et.al., 2021)



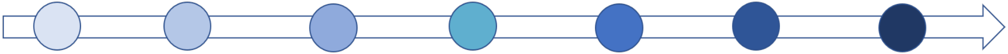
# Timeline of Deep Learning



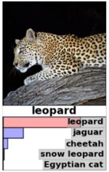

*source: adapted from Mourtzis & Angelopoulos (2020)*

# Recent deep learning applications

Image classification — Image generation — Text generation — Board games (Go, chess) — Strategy games (Starcraft, Dota2) — Code generation — Protein folding



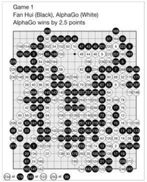
2012 ... ... ... ... ... 2021




*[Krizhevsky et.al., 2012]*

*[Goodfellow et.al., 2014]*
*[Karras et.al., 2018]*
*[news.artnet.com, 2018]*

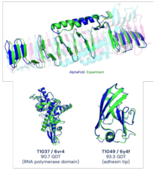
*[Vaswani et.al., 2017]*
*[Hochreiter et.al., 2018]*
*[Devlin et.al., 2019]*




*[Silver et.al., 2016]*
*[Silver et.al., 2017]*

*[openai.com/five, 2018]*
*[Oriol et.al., 2019]*

*[Brown et.al., 2020]*
*[Chen et.al., 2021]*



*[Jumper et.al., 2021]*

# Since 2021

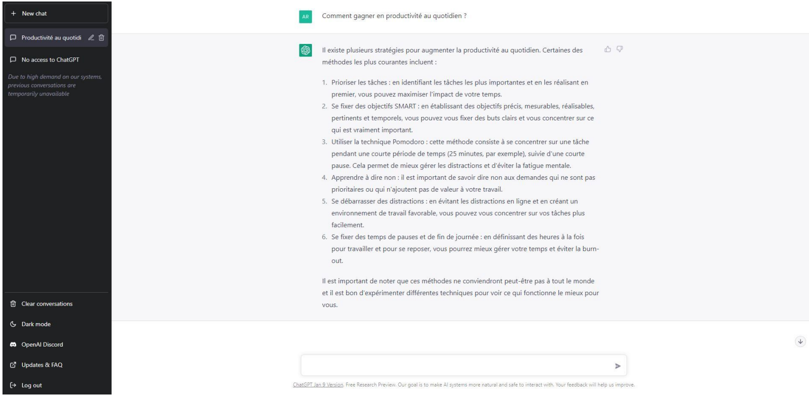
## Thousands of applications

- **Voice/audio/music generation:** MusicGen, MusicLM, MusicLDM, Jukebox, HeyGen
- **Voice to text:** Whisper
- **Image generation/deep-fakes:** Dalle-3, MidJourney, Stable Diffusion XL
- **Text generation/chatbots:** ChatGPT, GPT4, LLama, Claude, Mistral
- **Video generation:** Make-a-video, HeyGen
- **Code generation/automatic app creation:** Codex, Code LLama, phi-1.5, AutoGPT
- **Strategic games (Go, chess, Starcraft, diplomacy):** AlphaZero, LeelaChess, Cicero
- **Autonomous driving**
- ...

# Most recent breakthroughs: image generation (Dalle3, SD, MJ, ...)

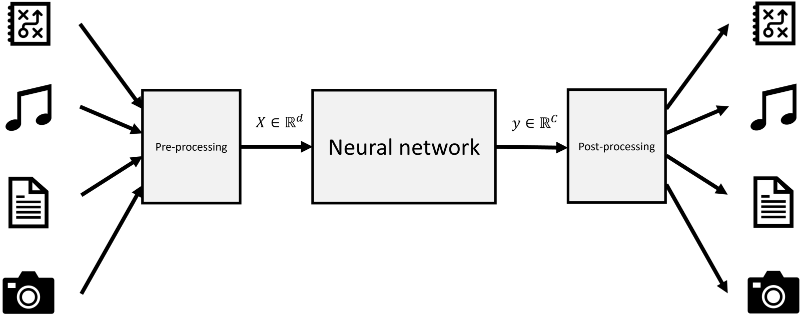
*Images generated from prompts using MidJourney (https://www.midjourney.com/)*

# Most recent breakthroughs: text generation (GPT4, LLama, Claude, ...)




*source: OpenAI's ChatGPT (https://chat.openai.com/)*

# What is Deep Learning? (usual setup)



# What is Deep Learning? (required skills)

## What do you need to create a DL architecture?

1. Know how to **encode/decode data**
    - Data loader, data augmentation, data handling during training, mini-batch, ...
    - Encoding layers, one-hot, tokenization, embeddings, ...

# What is Deep Learning? (required skills)

## What do you need to create a DL architecture?

1. Know how to **encode/decode data**
   - Data loader, data augmentation, data handling during training, mini-batch, ...
   - Encoding layers, one-hot, tokenization, embeddings, ...
2. Know how to **create a neural network**
   - Different types of layers, attention mechanism, batch normalization, ...
   - Multiple architectures: MLPs, RNNs, CNNs, GNNs, Transformers, ...

# What is Deep Learning? (required skills)

## What do you need to create a DL architecture?

1. Know how to **encode/decode data**
   - Data loader, data augmentation, data handling during training, mini-batch, ...
   - Encoding layers, one-hot, tokenization, embeddings, ...
2. Know how to **create a neural network**
   - Different types of layers, attention mechanism, batch normalization, ...
   - Multiple architectures: MLPs, RNNs, CNNs, GNNs, Transformers, ...
3. Know how to **train the neural network**
   - Optimization perspective, auto-diff, SGD, Adam, momentum, ...
   - Weight initialization, loss functions, scheduling, hyper-parameter optimization...

# What is Deep Learning? (twitter wisdom)

**Yann LeCun**
@ylecun

Some folks still seem confused about what deep learning is. Here is a definition:

DL is constructing networks of parameterized functional modules & training them from examples using gradient-based optimization....
facebook.com/722677142/post...

Traduire le Tweet

4:32 PM · 24 déc. 2019 · Facebook

# What is Deep Learning? (twitter wisdom)

**Yann LeCun**
@ylecun

Some folks still seem confused about what deep learning is. Here is a definition:

DL is constructing networks of parameterized functional modules & training them from examples using gradient-based optimization....
facebook.com/722677142/post...

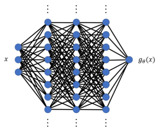
Traduire le Tweet

4:32 PM · 24 déc. 2019 · Facebook

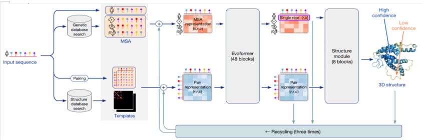
# Why Deep Learning Now?

- Five decades of research in machine learning

Multi-Layer Perceptron
(Rumelhart, Hinton, Williams, 75)

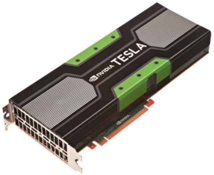


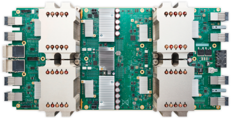

AlphaFold
(Jumper et.al., 2021)



# Why Deep Learning Now?

- Five decades of research in machine learning
- CPUs/GPUs/storage developed for other purposes



# Why Deep Learning Now?

- Five decades of research in machine learning
- CPUs/GPUs/storage developed for other purposes
- lots of data from "the internet"

# Why Deep Learning Now?

- Five decades of research in machine learning
- CPUs/GPUs/storage developed for other purposes
- lots of data from “the internet”
- tools and culture of collaborative and reproducible science

# Why Deep Learning Now?

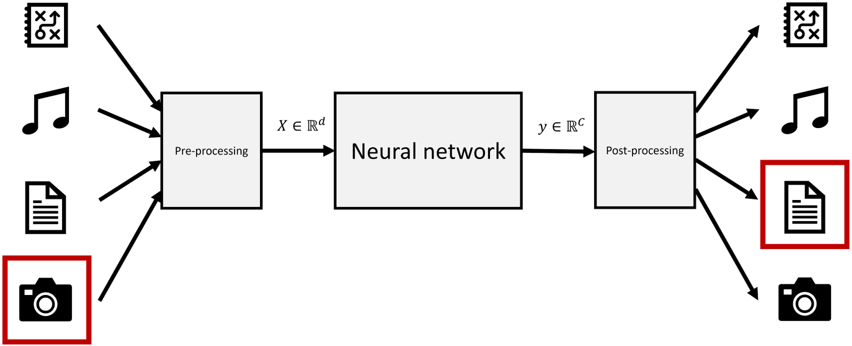
- Five decades of research in machine learning
- CPUs/GPUs/storage developed for other purposes
- lots of data from “the internet”
- tools and culture of collaborative and reproducible science
- resources and efforts from large corporations

# Machine Learning pipeline

## A short recap

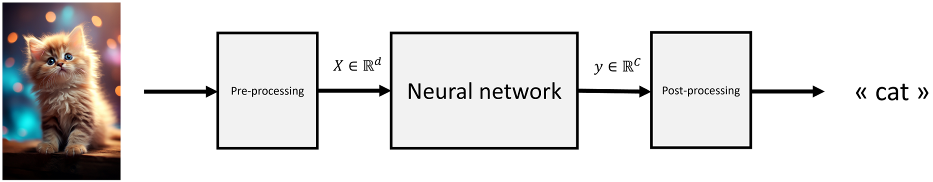
# Simple example: cats vs. dogs

Typical binary classification task. Objective is to distinguish **cat images from dog images**.



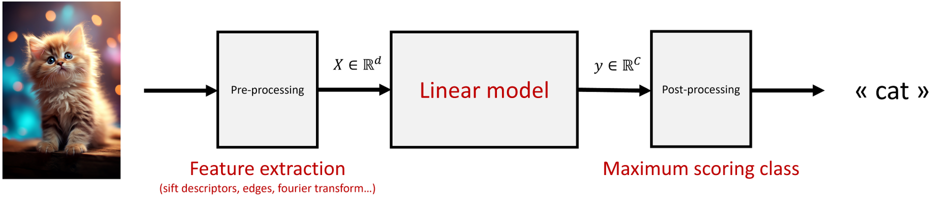
# Simple example: cats vs. dogs

Output class is represented as a **2d vector** ($(0, 1)$ for "cat" and $(1, 0)$ for "dog").



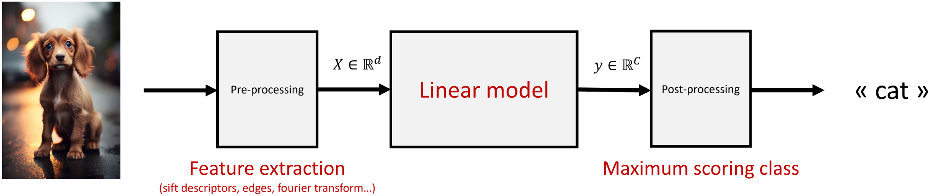
# Simple example: cats vs. dogs (linear model)

**Image features** (sift, wavelets,...) are extracted and given as input to the model.



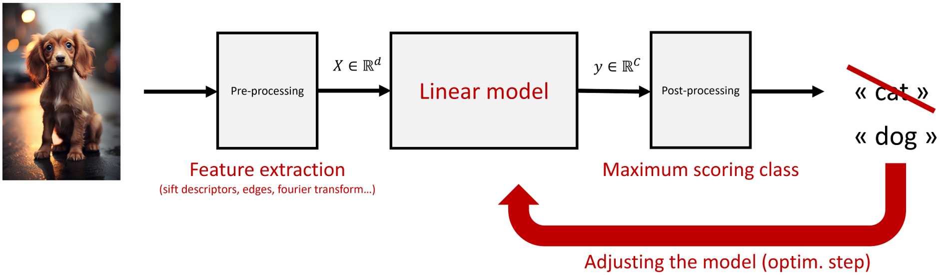
# Simple example: cats vs. dogs (inference)

The model makes a **prediction** ("cat" or "dog") for a given image.



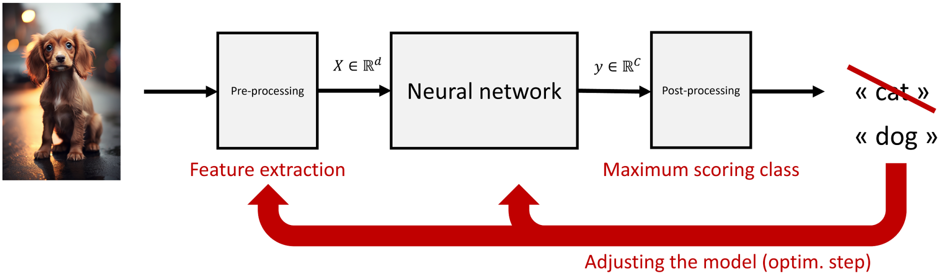
# Simple example: cats vs. dogs (training loop)

If the prediction is false, the **model updates its parameters** to improve its prediction.



# Simple example: cats vs. dogs (deep learning version)

In deep learning, we can train the **whole pipeline** using **automatic differentiation**.



# Typical Machine Learning setup

## Data distribution

Let $\mathcal{X}, \mathcal{Y}$ be an input and output space and $\mathcal{D}$ a distribution over $(\mathcal{X}, \mathcal{Y})$. Then, we denote our (test) input/output pair as

$$(X, Y) \sim \mathcal{D}$$

# Typical Machine Learning setup

## Data distribution

Let $\mathcal{X}, \mathcal{Y}$ be an input and output space and $\mathcal{D}$ a distribution over $(\mathcal{X}, \mathcal{Y})$. Then, we denote our (test) input/output pair as

$$(X, Y) \sim \mathcal{D}$$

## Risk minimization (a.k.a. supervized ML)

The objective of *risk minimization* is to find a minimizer $\theta^\star \in \mathbb{R}^p$ of the optimization problem

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}\big(\ell(g_\theta(X), Y)\big)$$

where $\ell : \mathcal{Y}^2 \to \mathbb{R}_+$ is a loss function and $g_\theta : \mathcal{X} \to \mathcal{Y}$ a model parameterized by $\theta \in \mathbb{R}^p$.

# Typical Machine Learning setup

## Data distribution

Let $\mathcal{X}, \mathcal{Y}$ be an input and output space and $\mathcal{D}$ a distribution over $(\mathcal{X}, \mathcal{Y})$. Then, we denote our (test) input/output pair as

$$(X, Y) \sim \mathcal{D}$$

## Risk minimization (a.k.a. supervized ML)

The objective of *risk minimization* is to find a minimizer $\theta^\star \in \mathbb{R}^p$ of the optimization problem

$$\min_{\theta \in \mathbb{R}^p} \mathbb{E}\big(\ell(g_\theta(X), Y)\big)$$

where $\ell : \mathcal{Y}^2 \to \mathbb{R}_+$ is a loss function and $g_\theta : \mathcal{X} \to \mathcal{Y}$ a model parameterized by $\theta \in \mathbb{R}^p$.

The target loss (e.g. accuracy) may be hard to train, and can thus be different from the one used as objective during training!

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)
- **Output data:** $Y \in \mathbb{R}^2$ are classes, **one-hot** encoded (i.e. $Y_i = 1$ iff $i$ is the true class).

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)
- **Output data:** $Y \in \mathbb{R}^2$ are classes, **one-hot** encoded (i.e. $Y_i = 1$ iff $i$ is the true class).
- **Training data:** Image-label pairs $(x_i, y_i)_{i \in [\![1,n]\!]}$ ($n$ number of data points).

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)
- **Output data:** $Y \in \mathbb{R}^2$ are classes, **one-hot** encoded (i.e. $Y_i = 1$ iff $i$ is the true class).
- **Training data:** Image-label pairs $(x_i, y_i)_{i \in [\![1,n]\!]}$ ($n$ number of data points).
- **Model:** $g_\theta : X \mapsto \langle \theta, f(X) \rangle$, where $f(X) \in \mathbb{R}^F$ are pre-computed features and $\theta \in \mathbb{R}^F$.

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)
- **Output data:** $Y \in \mathbb{R}^2$ are classes, **one-hot** encoded (i.e. $Y_i = 1$ iff $i$ is the true class).
- **Training data:** Image-label pairs $(x_i, y_i)_{i \in [\![1,n]\!]}$ ($n$ number of data points).
- **Model:** $g_\theta : X \mapsto \langle \theta, f(X) \rangle$, where $f(X) \in \mathbb{R}^F$ are pre-computed features and $\theta \in \mathbb{R}^F$.
- **Loss function (test):** $\ell(y, y') = \mathbb{1}\{\text{argmax}_i\, y'_i = \text{argmax}_i\, y_i\}$ (accuracy)

# Typical Machine Learning setup (back to cats and dogs)

- **Input data:** $X \in [0, 255]^{w \times h \times 3}$ are images encoded as **tensors** (i.e. high-dim. matrices)
- **Output data:** $Y \in \mathbb{R}^2$ are classes, **one-hot** encoded (i.e. $Y_i = 1$ iff $i$ is the true class).
- **Training data:** Image-label pairs $(x_i, y_i)_{i \in [\![1,n]\!]}$ ($n$ number of data points).
- **Model:** $g_\theta : X \mapsto \langle \theta, f(X) \rangle$, where $f(X) \in \mathbb{R}^F$ are pre-computed features and $\theta \in \mathbb{R}^F$.
- **Loss function (test):** $\ell(y, y') = \mathbb{1}\{\text{argmax}_i\, y'_i = \text{argmax}_i\, y_i\}$ (accuracy)
- **Loss function (train):** $\ell(y, y') = -\sum_i y'_i \ln\left(\exp(y_i) / \sum_j \exp(y_j)\right)$ (cross entropy)

# Training objective

## Empirical risk minimization

Let $(x_i, y_i)_{i \in [\![1,n]\!]}$ be a collection of $n$ observations drawn independently according to $\mathcal{D}$.

Then, the objective of *empirical risk minimization* (ERM) is to find a minimizer $\hat{\theta}_n \in \mathbb{R}^p$ of

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$$

# Training objective

## Empirical risk minimization

Let $(x_i, y_i)_{i \in [\![1,n]\!]}$ be a collection of $n$ observations drawn independently according to $\mathcal{D}$. Then, the objective of *empirical risk minimization* (ERM) is to find a minimizer $\hat{\theta}_n \in \mathbb{R}^p$ of

$$\min_{\theta \in \mathbb{R}^p} \frac{1}{n} \sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$$

## Optimization by gradient descent

We can minimize this loss by iterating

$$\theta_{t+1} = \theta_t - \eta \nabla \hat{\mathcal{L}}_n(\theta_t)$$

where $\eta > 0$ is a fixed step-size and $\hat{\mathcal{L}}_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(g_\theta(x_i), y_i)$ is our objective.

# Typical loss functions

- In its simplest form, the **accuracy** is $\ell(y, y') = \mathbb{1}\{y \neq y'\}$.

# Typical loss functions

- In its simplest form, the **accuracy** is $\ell(y, y') = \mathbb{1}\{y \neq y'\}$.
- For **classification** tasks, we usually use $\mathcal{Y} = \mathbb{R}^C$ where $C$ is the number of classes, and
  - $\ell(y, y') = \mathbb{1}\{\operatorname{argmax}_i y'_i = \operatorname{argmax}_i y_i\}$ (top-1 accuracy) or,
  - $\ell(y, y') = -\sum_i y'_i \ln\left(\exp(y_i)/\sum_j \exp(y_j)\right)$ (cross entropy).

# Typical loss functions

- In its simplest form, the **accuracy** is $\ell(y, y') = \mathbb{1}\{y \neq y'\}$.
- For **classification** tasks, we usually use $\mathcal{Y} = \mathbb{R}^C$ where $C$ is the number of classes, and
  - $\ell(y, y') = \mathbb{1}\{\mathrm{argmax}_i\, y'_i = \mathrm{argmax}_i\, y_i\}$ (top-1 accuracy) or,
  - $\ell(y, y') = -\sum_i y'_i \ln\left(\exp(y_i)/\sum_j \exp(y_j)\right)$ (cross entropy).
- For **regression** tasks, we usually use $\mathcal{Y} = \mathbb{R}^d$ and
  - $\ell(y, y') = \|y - y'\|_2^2 = \sum_i (y_i - y'_i)^2$ (mean square error) or,
  - $\ell(y, y') = \|y - y'\|_1 = \sum_i |y_i - y'_i|$ (mean absolute error).

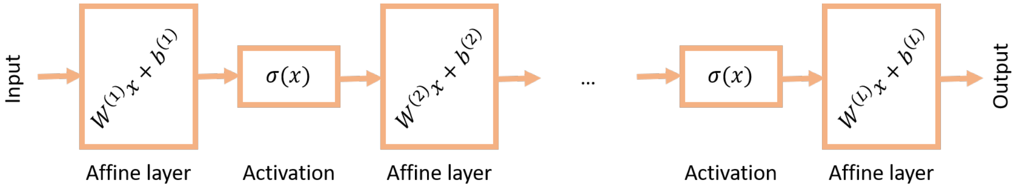
# Recap

- Learning is rephrased as minimizing a **loss function** over the **training dataset**.
- Loss is typically **cross entropy** for classification and **MSE** for regression.
- Training achieved by (stochastic) **gradient descent** (or its variants).
- The whole pipeline is trained (i.e. its parameters are optimized) using **autodiff**.

# Multi-Layer Perceptron

## Definition and Pytorch implementation

# Multi-Layer Perceptron (Rumelhart, Hinton, Williams, 75)




## Details

- **Idea:** Composition of **affine** (also called linear) and **activation** (simple non-linear coordinate-wise) functions. Simple extension of linear models.
- **Activations:** Coordinate-wise functions. (usually ReLU i.e. $\sigma(x)_i = \max\{0, x_i\}$).
- **Update rule:** $x^{(l+1)} = \sigma(W^{(l)}x^{(l)} + b^{(l)})$ (except for the last layer!).
- **Brain analogy:** A *"neuron"* is a coordinate of an activation layer.

# Structure of MLPs with ReLU activations

## ReLU networks create affine regions

- Case of two layers and $d^{(2)} = 1$: $g_\theta(x) = \sum_i w_i^{(2)} \sigma(\langle w_i^{(1)}, x \rangle + b_i) + c$

# Structure of MLPs with ReLU activations

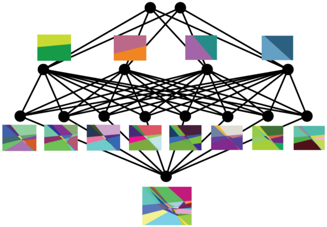
## ReLU networks create affine regions

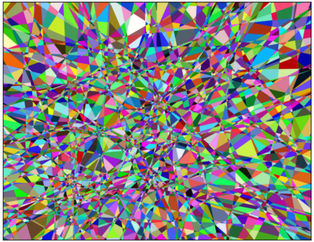
- Case of two layers and $d^{(2)} = 1$: $g_\theta(x) = \sum_i w_i^{(2)} \sigma(\langle w_i^{(1)}, x \rangle + b_i) + c$
- Each ReLU activation can create a new affine region.

# Structure of MLPs with ReLU activations

## ReLU networks create affine regions

- Case of two layers and $d^{(2)} = 1$: $g_\theta(x) = \sum_i w_i^{(2)} \sigma(\langle w_i^{(1)}, x \rangle + b_i) + c$
- Each ReLU activation can create a new affine region.
- Example of affine regions of a ReLU network trained on MNIST:





*(image credits: Hanin & Rolnik, 2019)*

# Pytorch implementation

- Simple implementation as a **sequence of base operations**
- **Affine layers** in Pytorch:

```
layer = torch.nn.Linear(n_in, n_out)
```

- **ReLU activation layers** in Pytorch:

```
layer = torch.nn.ReLU()
```

- Each layer contains its parameters, that can be accessed with layer.parameters().
- We can thus create an **MLP** with the code:

```
model = torch.nn.Sequential(torch.nn.Linear(n_in, n_internal),
                            torch.nn.ReLU(),
                            ...,
                            torch.nn.Linear(n_internal, n_out))
```

# Automatic differentiation

## Differentiating composite functions

# Existing approaches to compute gradients

- **Finite differences:** small perturbations $g'(x) \approx \frac{g(x+\varepsilon)-g(x)}{\varepsilon}$. Leads to **round-off** errors.

# Existing approaches to compute gradients

- **Finite differences:** small perturbations $g'(x) \approx \frac{g(x+\varepsilon)-g(x)}{\varepsilon}$. Leads to **round-off** errors.
- **Symbolic differentiation:** keeps **symbolic expressions** at each step of the process.

# Existing approaches to compute gradients

- **Finite differences:** small perturbations $g'(x) \approx \frac{g(x+\varepsilon)-g(x)}{\varepsilon}$. Leads to **round-off** errors.
- **Symbolic differentiation:** keeps **symbolic expressions** at each step of the process.
- **Automatic differentiation:** clever use of the **chain rule**.

# Existing approaches to compute gradients

- **Finite differences:** small perturbations $g'(x) \approx \frac{g(x+\varepsilon)-g(x)}{\varepsilon}$. Leads to **round-off** errors.
- **Symbolic differentiation:** keeps **symbolic expressions** at each step of the process.
- **Automatic differentiation:** clever use of the **chain rule**.

## Chain rule (simple version)

Let $f, g : \mathbb{R} \to \mathbb{R}$ differentiable, then

$$(f \circ g)' = (f' \circ g) \cdot g'$$

# Recap: derivatives of multi-dimensional functions

## Definition (Jacobian matrix)

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ a differentiable function. Its Jacobian $J_f(x) \in \mathbb{R}^{m \times n}$ is the matrix whose coordinates are the partial derivatives:

$$J_f(x) = \begin{bmatrix} \nabla f_1(x)^\top \\ \cdots \\ \nabla f_m(x)^\top \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \dots & \frac{\partial f_1(x)}{\partial x_n} \\ \dots & \dots & \dots \\ \frac{\partial f_m(x)}{\partial x_1} & \dots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}$$

# Recap: derivatives of multi-dimensional functions

## Definition (Jacobian matrix)

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ a differentiable function. Its Jacobian $J_f(x) \in \mathbb{R}^{m \times n}$ is the matrix whose coordinates are the partial derivatives:

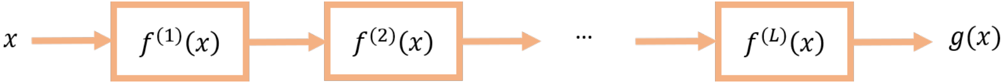
$$J_f(x) = \begin{bmatrix} \nabla f_1(x)^\top \\ \cdots \\ \nabla f_m(x)^\top \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_m(x)}{\partial x_1} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}$$

## Chain rule (multi-dimensional version)

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^p \to \mathbb{R}^n$ differentiable, then

$$J_{f \circ g} = (J_f \circ g) \times J_g$$

# Derivative of a composition of functions

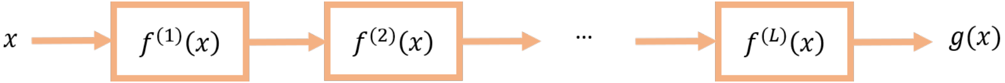



## Composite function

- Let $f^{(l)} : \mathbb{R}^{d^{(l-1)}} \rightarrow \mathbb{R}^{d^{(l)}}$ and $g(x) = g^{(L)}(x)$ where

$$g^{(l)}(x) = f^{(l)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(x)$$

# Derivative of a composition of functions




## Composite function

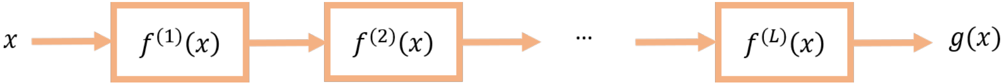
- Let $f^{(l)} : \mathbb{R}^{d(l-1)} \to \mathbb{R}^{d(l)}$ and $g(x) = g^{(L)}(x)$ where

$$g^{(l)}(x) = f^{(l)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(x)$$

- Then, the Jacobian matrix (i.e. matrix of derivatives) of $g$ is

$$J_g(x) = J_{f^{(L)}}\left(g^{(L-1)}(x)\right) \times \cdots \times J_{f^{(2)}}\left(g^{(1)}(x)\right) \times J_{f^{(1)}}(x)$$

# Derivative of a composition of functions




## Composite function

- Let $f^{(l)} : \mathbb{R}^{d(l-1)} \to \mathbb{R}^{d(l)}$ and $g(x) = g^{(L)}(x)$ where

$$g^{(l)}(x) = f^{(l)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(x)$$

- Then, the Jacobian matrix (i.e. matrix of derivatives) of $g$ is

$$J_g(x) = J_{f^{(L)}}\left(g^{(L-1)}(x)\right) \times \cdots \times J_{f^{(2)}}\left(g^{(1)}(x)\right) \times J_{f^{(1)}}(x)$$

- What is the **computational complexity** to compute the Jacobian matrix?

# Computational complexity

## Finite differences

- The gradient of $g$ can be approximated by **finite differences**: $\nabla g(x)_i \approx \frac{g(x+\varepsilon e_i)-g(x)}{\varepsilon}$
- **Computational complexity:** proportional to **input dimension**.

# Computational complexity

## Finite differences

- The gradient of $g$ can be approximated by **finite differences**: $\nabla g(x)_i \approx \frac{g(x+\varepsilon e_i)-g(x)}{\varepsilon}$
- **Computational complexity:** proportional to **input dimension**.

## Matrix product

- We have $\nabla g(x)^\top = J_L \times \cdots \times J_2 \times J_1$ where $J_l = J_{f^{(l)}}\left(g^{(l-1)}(x)\right)$.

# Computational complexity

## Finite differences

- The gradient of $g$ can be approximated by **finite differences**: $\nabla g(x)_i \approx \frac{g(x+\varepsilon e_i)-g(x)}{\varepsilon}$
- **Computational complexity:** proportional to **input dimension**.

## Matrix product

- We have $\nabla g(x)^\top = J_L \times \cdots \times J_2 \times J_1$ where $J_l = J_{f^{(l)}}\left(g^{(l-1)}(x)\right)$.
- There are $(L-1)!$ ways to compute this product of $L$ matrices.

# Computational complexity

## Finite differences

- The gradient of $g$ can be approximated by **finite differences**: $\nabla g(x)_i \approx \frac{g(x+\varepsilon e_i)-g(x)}{\varepsilon}$
- **Computational complexity:** proportional to **input dimension**.

## Matrix product

- We have $\nabla g(x)^\top = J_L \times \cdots \times J_2 \times J_1$ where $J_l = J_{f^{(l)}}\left(g^{(l-1)}(x)\right)$.
- There are $(L-1)!$ ways to compute this product of $L$ matrices.
- **Forward propagation:** Compute $\nabla g(x)^\top = (J_L \times (J_{L-1} \times \cdots \times (J_2 \times J_1)))$. Requires computation intensive **matrix-matrix products**.

# Computational complexity

## Finite differences

- The gradient of $g$ can be approximated by **finite differences**: $\nabla g(x)_i \approx \frac{g(x+\varepsilon e_i)-g(x)}{\varepsilon}$
- **Computational complexity:** proportional to **input dimension**.

## Matrix product

- We have $\nabla g(x)^\top = J_L \times \cdots \times J_2 \times J_1$ where $J_l = J_{f^{(l)}}\left(g^{(l-1)}(x)\right)$.
- There are $(L-1)!$ ways to compute this product of $L$ matrices.
- **Forward propagation:** Compute $\nabla g(x)^\top = (J_L \times (J_{L-1} \times \cdots \times (J_2 \times J_1)))$. Requires computation intensive **matrix-matrix products**.
- **Backward propagation:** Compute $\nabla g(x)^\top = (((J_L \times J_{L-1}) \times \cdots \times J_2) \times J_1)$. If output is 1-dimensional, only needs **matrix-vector products**!

# Which algorithm is faster?

## Complexity for gradients of MLPs

- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:**
- **Finite differences:**
- **Forward propagation:**
- **Backward propagation:**

# Which algorithm is faster?

## Complexity for gradients of MLPs

- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:** $O(w^2 L)$ operations.
- **Finite differences:**
- **Forward propagation:**
- **Backward propagation:**

# Which algorithm is faster?

## Complexity for gradients of MLPs

- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:** $O(w^2 L)$ operations.
- **Finite differences:** $O(dw^2 L)$ operations.
- **Forward propagation:**
- **Backward propagation:**

# Which algorithm is faster?

## Complexity for gradients of MLPs

- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:** $O(w^2 L)$ operations.
- **Finite differences:** $O(dw^2 L)$ operations.
- **Forward propagation:** $O(dw^2 L)$ operations.
- **Backward propagation:**

# Which algorithm is faster?

## Complexity for gradients of MLPs

- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:** $O(w^2 L)$ operations.
- **Finite differences:** $O(dw^2 L)$ operations.
- **Forward propagation:** $O(dw^2 L)$ operations.
- **Backward propagation:** $O(w^2 L)$ operations.

# Which algorithm is faster?

## Complexity for gradients of MLPs

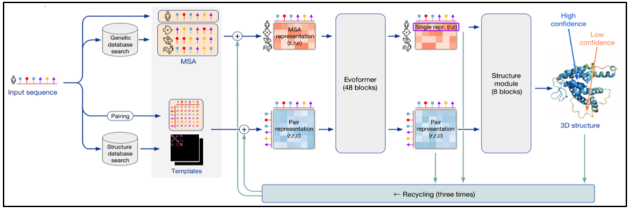
- Let $g_\theta : \mathbb{R}^d \to \mathbb{R}$ an MLP of width $w \geqslant d$ and depth $L \geqslant 1$.
- **Function value:** $O(w^2 L)$ operations.
- **Finite differences:** $O(dw^2 L)$ operations.
- **Forward propagation:** $O(dw^2 L)$ operations.
- **Backward propagation:** $O(w^2 L)$ operations.

## Intuition for gradients w.r.t. parameters

- Finite differences requires **two function calls per parameter**.
- Backprop requires **O(1) function calls for the whole gradient**.
- Interpretation as parameter testing:
  - Each partial derivative w.r.t. a parameter indicates if this parameter can describe the data.
  - With backprop, we can test **all parameters at once**.

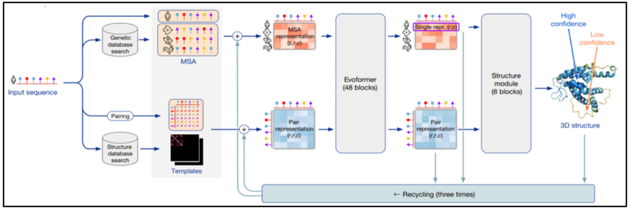
# Computation graphs: intuition

Complex neural network architecture (e.g. AlphaFold)



# Computation graphs: intuition

Complex neural network architecture (e.g. AlphaFold)

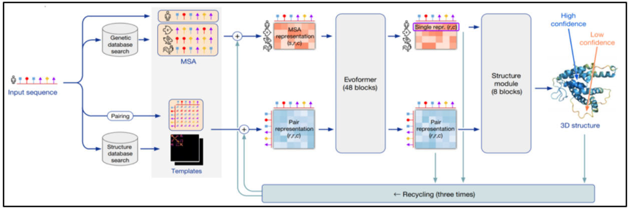



Code (e.g. Python)

```
z1 = x * y
z2 = x ** 2
z3 = exp(z1)
z4 = 2 * z2
z5 = z3 + z4
out = sin(z5)
```

# Computation graphs: intuition

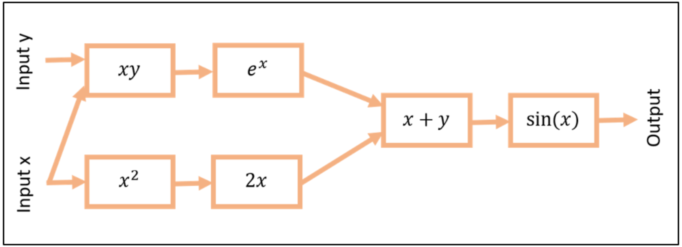
Complex neural network architecture (e.g. AlphaFold)




Code (e.g. Python)

```
z1 = x * y
z2 = x ** 2
z3 = exp(z1)
z4 = 2 * z2
z5 = z3 + z4
out = sin(z5)
```

Computation graph (DAG of mathematical operations)



# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.

# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.
- **Parameters:** For any root $r \in R$, let $x^{(r)} = \theta^{(r)}$ be an input or parameter.

# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.
- **Parameters:** For any root $r \in R$, let $x^{(r)} = \theta^{(r)}$ be an input or parameter.
- **Layers:** For any other node $v \in V/R$, let $x^{(v)} = f^{(v)}\left((x^{(w)})_{w \in \text{Parents}(v)}\right)$.

# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.
- **Parameters:** For any root $r \in R$, let $x^{(r)} = \theta^{(r)}$ be an input or parameter.
- **Layers:** For any other node $v \in V/R$, let $x^{(v)} = f^{(v)}\left((x^{(w)})_{w \in \mathsf{Parents}(v)}\right)$.
- **Output:** The output of the leaf node $x^{(f)} = \mathcal{L}(\theta) \in \mathbb{R}$ where $\theta = (\theta^{(r)})_{r \in R}$.

# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.
- **Parameters:** For any root $r \in R$, let $x^{(r)} = \theta^{(r)}$ be an input or parameter.
- **Layers:** For any other node $v \in V/R$, let $x^{(v)} = f^{(v)}\left((x^{(w)})_{w \in \mathrm{Parents}(v)}\right)$.
- **Output:** The output of the leaf node $x^{(f)} = \mathcal{L}(\theta) \in \mathbb{R}$ where $\theta = (\theta^{(r)})_{r \in R}$.

## Properties

- Essentially **all programmable functions** can be decomposed this way.

# Computation graphs: formal definition

## Definition (computation graph)

- Let $G = (V, E)$ be a *directed acyclic graph* (DAG) encoding a function $\mathcal{L} : \mathbb{R}^d \to \mathbb{R}$.
- **Parameters:** For any root $r \in R$, let $x^{(r)} = \theta^{(r)}$ be an input or parameter.
- **Layers:** For any other node $v \in V/R$, let $x^{(v)} = f^{(v)}\left((x^{(w)})_{w \in \text{Parents}(v)}\right)$.
- **Output:** The output of the leaf node $x^{(f)} = \mathcal{L}(\theta) \in \mathbb{R}$ where $\theta = (\theta^{(r)})_{r \in R}$.

## Properties

- Essentially **all programmable functions** can be decomposed this way.
- **Chain rule:** partial gradient $\frac{\partial x^{(f)}}{\partial x^{(v)}}$ for a node $v \in V$ from that of its children.

$$\frac{\partial x^{(f)}}{\partial x^{(v)}} = \sum_{w \in \text{Children}(v)} \frac{\partial f^{(w)}\left((x^{(w')})_{w' \in \text{Parents}(w)}\right)}{\partial x^{(v)}}^{\top} \frac{\partial x^{(f)}}{\partial x^{(w)}}$$

# The backpropagation algorithm (Rumelhart et al., 1986)

- Composed of 2 steps: a **forward pass** (FP) and a **backward pass** (BP).

# The backpropagation algorithm (Rumelhart et al., 1986)

- Composed of 2 steps: a **forward pass** (FP) and a **backward pass** (BP).
- **FP:** For all $r \in R$, let $y^{(r)} = x_r$ the inputs (or parameters), and, for all $v \in V/R$, we compute iteratively **from roots to leaf**,

$$y^{(v)} = f^{(v)}\left(\left(y^{(w)}\right)_{w \in \text{Parents}(v)}\right)$$

# The backpropagation algorithm (Rumelhart et al., 1986)

- Composed of 2 steps: a **forward pass** (FP) and a **backward pass** (BP).
- **FP:** For all $r \in R$, let $y^{(r)} = x_r$ the inputs (or parameters), and, for all $v \in V/R$, we compute iteratively **from roots to leaf**,

$$y^{(v)} = f^{(v)}\left((y^{(w)})_{w \in \text{Parents}(v)}\right)$$

- **BP:** Let $z^{(f)} = 1$ and, for $v \in V/F$, we compute iteratively **from leaf to roots**,

$$z^{(v)} = \sum_{w \in \text{Children}(v)} \frac{\partial f^{(w)}\left((y^{(w')})_{w' \in \text{Parents}(w)}\right)}{\partial x^{(v)}}^{\top} z^{(w)}$$

- Then, for all $r \in R$, we have $\frac{\partial \mathcal{L}(\theta)}{\partial \theta^{(r)}} = z^{(r)}$.

# Class overview

## Lessons

1. **Introduction, simple architectures (MLPs) and autodiff** 09/02
2. Training pipeline, optimization and image analysis (CNNs) 16/02
3. Sequence regression (RNNs), stability and robustness 08/03
4. Generative models in vision and text (Transformers, GANs) 15/03

## Practicals

- **TP1:** MLPs and CNNs in Pytorch 01/03
- **TP2:** RNNs and generative models 22/03