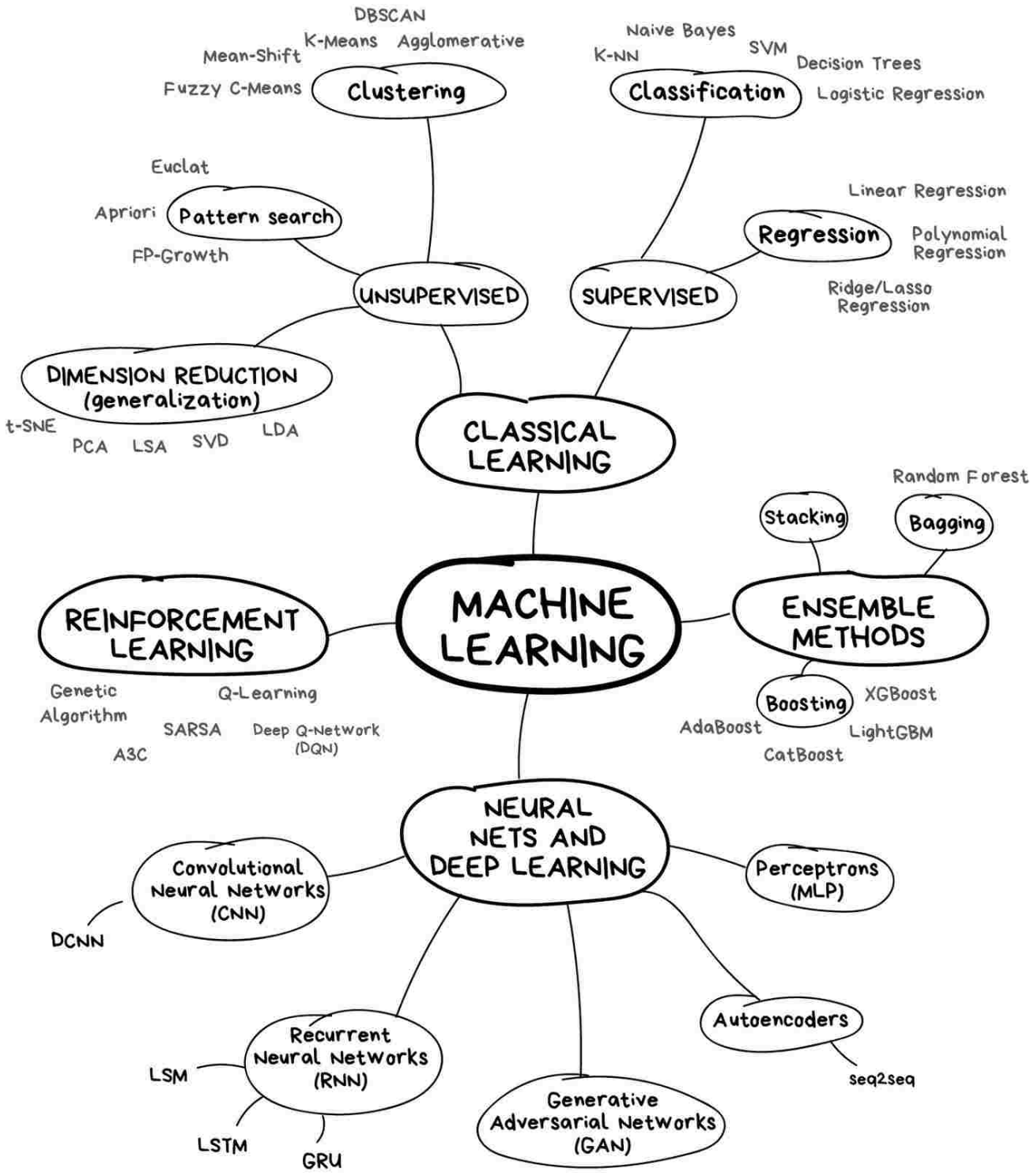
present

Deep Learning ⊂ Machine Learning ⊂ AI

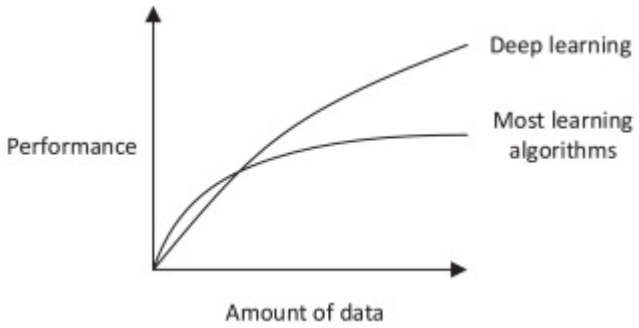


Why Did Deep Learning Become so

- Does not plateau with increase of data input.
- Better computation ressources: GPU and cloud computing.
- Packages allowing fast implementation (e.g. TensorFlow, PyTorch, Theano, Keras...). However this leads to a large community of users but a comparatively small one when it comes to understand the underlying principles!

Examples of Application:

- Data analysis.
- Image processing.
- Time series prediction.
- Control and system identification (learning the right-hand side of an ODE or even PDE).
- Anomaly detection.
- Natural language processing (NLP). Most famous example: GPT-3.



Introduction to Deep Learning

In this tutorial, you'll be walked through:

- An affine regression with a shallow network.
- The same regression on noisy data.
- A nonlinear regression with a deeper network.
- The common problems that can arise when using deep learning methods.
- Some previews of more complex architectures.

For this purpose we will use Julia as programming language, Pluto as notebook interface and Flux as deep learning package.

Linear Regression with Shallow

Importing Packages

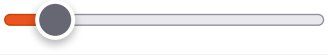
- Flux: a deep learning package that allows you to go pretty low-level, while being still user-freindly.
- Makie: a plotting package with several backend. Here the Cairo one has been chosen.
- PlutoUI: a package allowing the use of widgets.
- Random: a package for generating random outputs.
- LinearAlgebra: will provide us the matrix-norm functions.

```
using Flux , CairoMakie , PlutoUI , Random , LinearAlgebra
```

Linear Function as Ground Truth

Here we will construct some data that should be fitted by the ANN. To this end we create a random but reproduceable matrix M and bias vector p.

```
@bind nx PlutoUI.Slider(2:2:20, show_value=true, default=4)
```



```
@bind ny PlutoUI.Slider(2:2:20, show_value=true, default=4)
```

```
@bind nm PlutoUI.Slider(100:100:2000, show_value=true, default=1000)
```

```
input_range = 100
```

```
input_range = 100
```

```
4×4 Matrix{Float64}:
 -0.9    0.38   0.68  -0.6
 -0.76   0.54   0.78  -0.98
 -0.21  -0.83  -0.72   0.18
 -0.95   0.71  -0.3    0.6
```

```
begin
    Random.seed!(1)
```

```
[25.4006, 31.012, -38.7865, -6.5715]
```

```
linearmap([4.91, 11.90, 39.32, 2.40])
```

```
4×1000 Matrix{Float64}:
  25.3963  -78.5113  -53.2501    -25.3599  …  -62.5419  -36.9306  -41.281   -37.2457
  31.0071  -88.1898  -39.0641    -46.9691     -54.4798  -21.4411  -44.8547  -23.245
 -38.798   -69.1123  -95.275     -33.0573     -44.3955  -55.6806  -92.3828  -94.1014
  -6.5692   37.4928    0.125554   12.0246     -46.0414  -64.8993   22.1939  -91.3689
```

```
begin
    Random.seed!(1)
    X = input_range .* rand(nx, nm)
    Y = mapslices(linearmap, X; dims=1)
end
```

Build a Dense Layer

Dense layers have exactly the same structure as the above defined deterministic linear mapping!

$$y = Wx$$

The question now is: can we recover the entries of M by only using the data pair $\{X, Y\}$? Mathematically speaking:

$$\text{minimise} \quad \text{dist}(W, M)$$

We will now initialise such a layer with random parameters. In deep learning, *parameters* describe any value that can be optimised during the loss minimisation, e.g. W_{11}. In opposition, *hyperparameters* describe numerical values that will not be tuned at training time! For instance we can think of the dimension of the matrix W.

Constructing the Loss

The distance between model and reality is quantified by a loss w.r.t. the output of the model $\hat{Y}$. In other word, if the predicted data is similar to the ground truth, we consider the model to be suited. Our goal can now be formulated rigorously:

$$\underset{W}{\text{minimise}} \quad L(\hat{Y}, Y)$$

As an intuitive error measure, we choose the mean square error:

$$l(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^{\mathrm{T}}(\hat{y} - y)$$
$$L(\hat{Y}, Y) = \frac{1}{n_m} \Sigma_{\hat{y},y} l(\hat{y}, y).$$

```
loss (generic function with 1 method)
```

```
begin
    α = 1e-4
    ps = params(predict)
    gs(x,y) = Flux.gradient(() -> loss(x,y), ps)
    grad1 = gs(X[:,1], Y[:,1])
    dW = grad1.grads[grad1.params[1]]
    predict.layers[1].W .-= α .* dW
end
```

```
249.0228824649704
```

```
loss(X[:,1], Y[:,1])
```

Hurray! The cost decreased and if we repeat this several times, we might get a near-zero loss!

Split the Data

Before going to the full training procedure, we perform a common step called *data-splitting*. While a fraction f_1 is used for training, a fraction f_2 is used to control the generalisation error during the training. Finally a fraction f_3 is kept aside to evaluate the generalisation performance on data that was never evaluated by the ANN. Commonly, f f is chosen. Typical values of f

size n_b. This allows to update the parameters on a small number of experiments, thus avoiding some drawbacks:

- If $n_b = 1$, we get our previous update method. This is noisy and not well-suited in the vicinity of the minimum.
- If $n_b = n_m$, updating takes a long time and we miss some noisiness to leave local minima.

```
@bind batch_size PlutoUI.Slider(10:10:200, show_value=true, default=100)
```

```
train_loader =
  DataLoader((data = 4×700 Matrix{Float64}:
                      4.91718  69.1857  84.0384  19.8521    …  91.6438   76.446   19.4
                     11.9079   76.7518  89.077    0.819786      3.11955  23.4843  39.5
```

```
plot_loss (generic function with 1 method)
```

```
function plot_loss(ntrain, batch_size, trainloss, devloss)
    scale_epoch = ntrain/batch_size
    fig = Figure(resolution = (800, 400))
    ax = CairoMakie.Axis(fig[1,1], xlabel="epochs", ylabel="loss",
              xminorticks=IntervalsBetween(5), xminorgridvisible=true)
    lines!(ax, (1:length(trainloss)) ./  scale_epoch, Float32.(trainloss))
    lines!(ax, (1:length(devloss)) ./ scale_epoch , Float32.(devloss))
    fig
```

```
    Xn = input_range .* rand(nx, nn)
    Yn = mapslices(noisy_linearmap, X; dims=1)
end
```