Applied Machine Learning in Genomic and Nutritional Data Science

Tutorial @ D3 Spring School 2024

Jan Voges

About Us

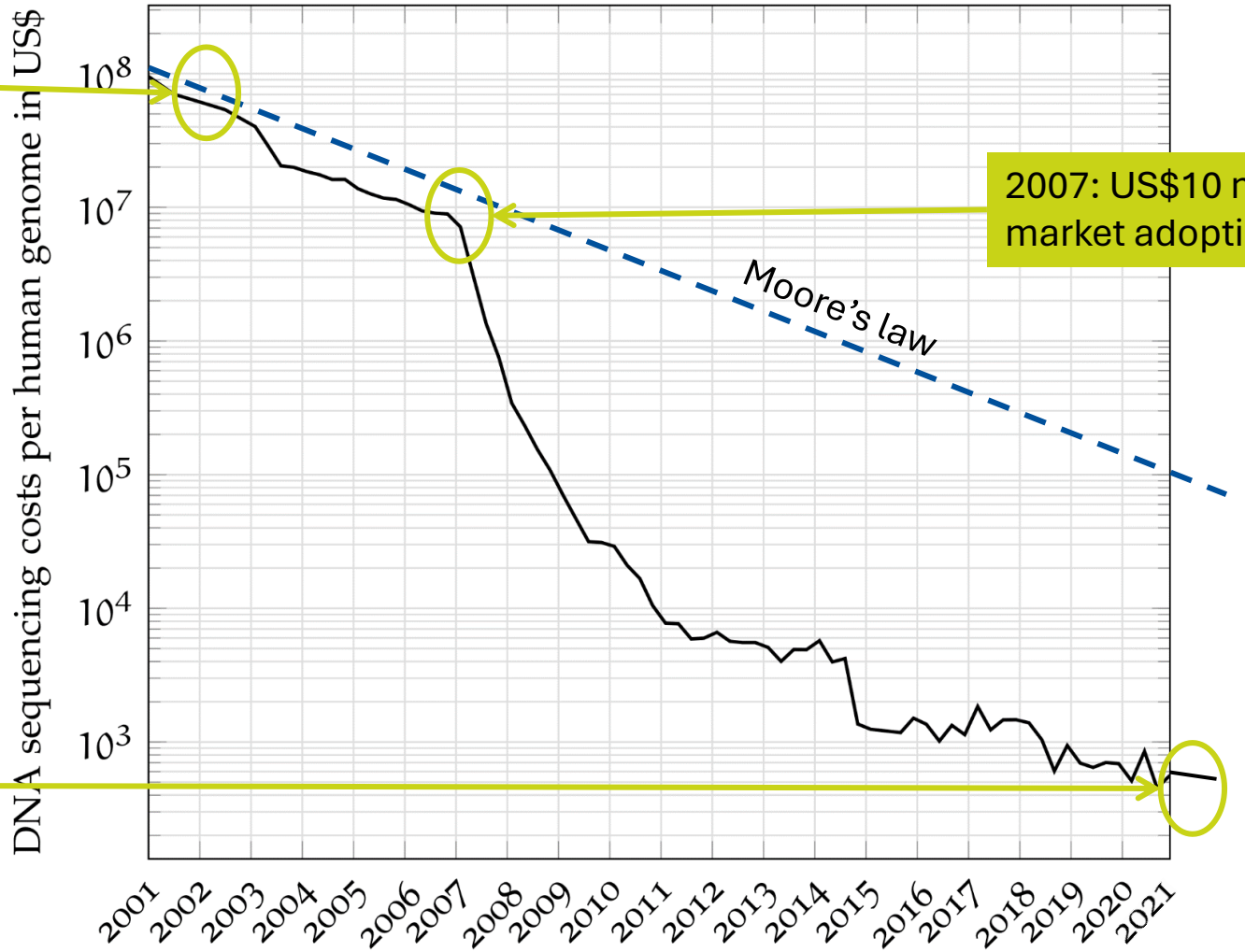
The **Genomic Data Science** Group at Leibniz University Hannover

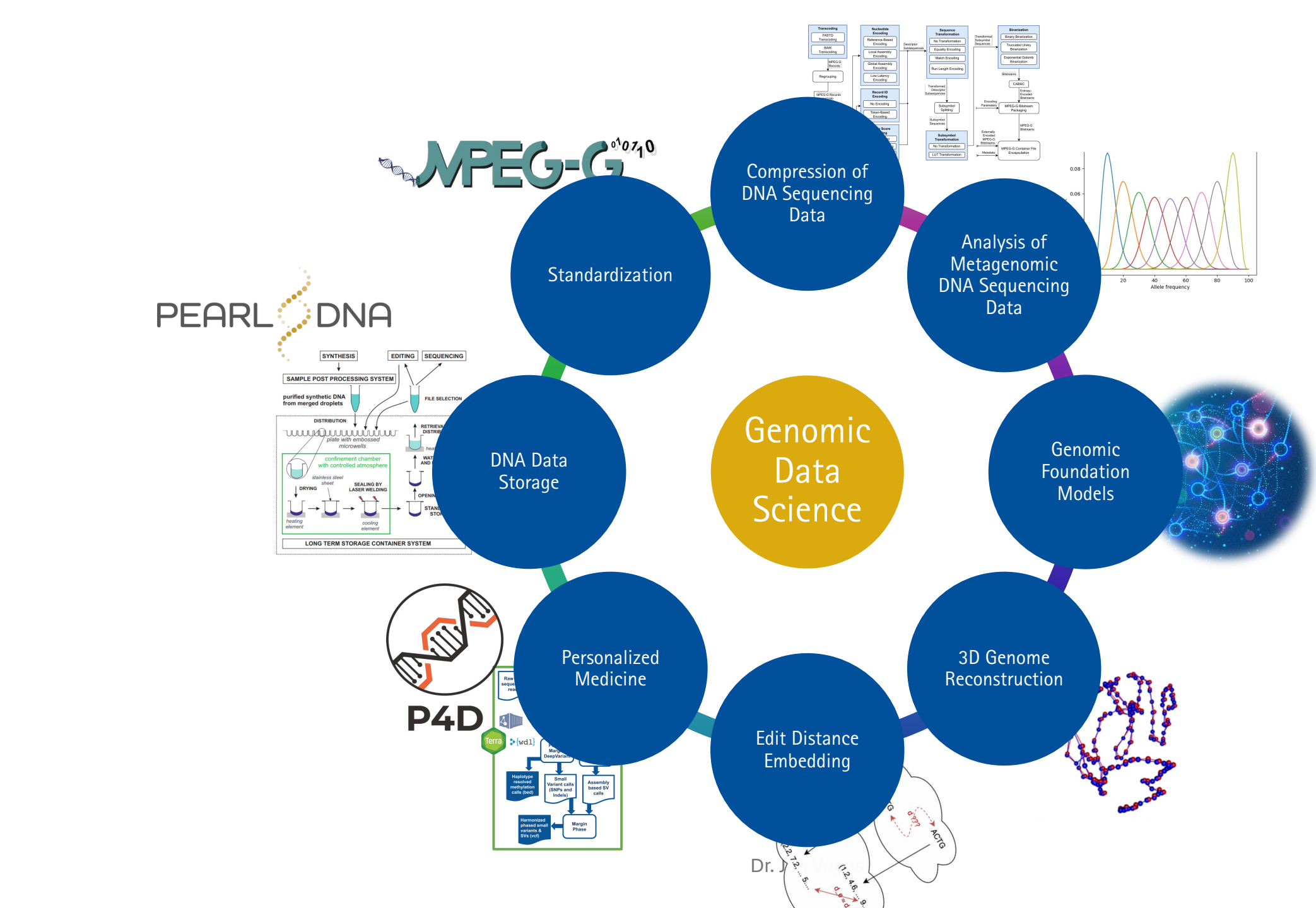
Falling DNA sequencing costs — genomics comes to life

1990–2003: US$3 billion (Human Genome Project)

2007: US$10 million (*before* widespread market adoption of HTS technologies)

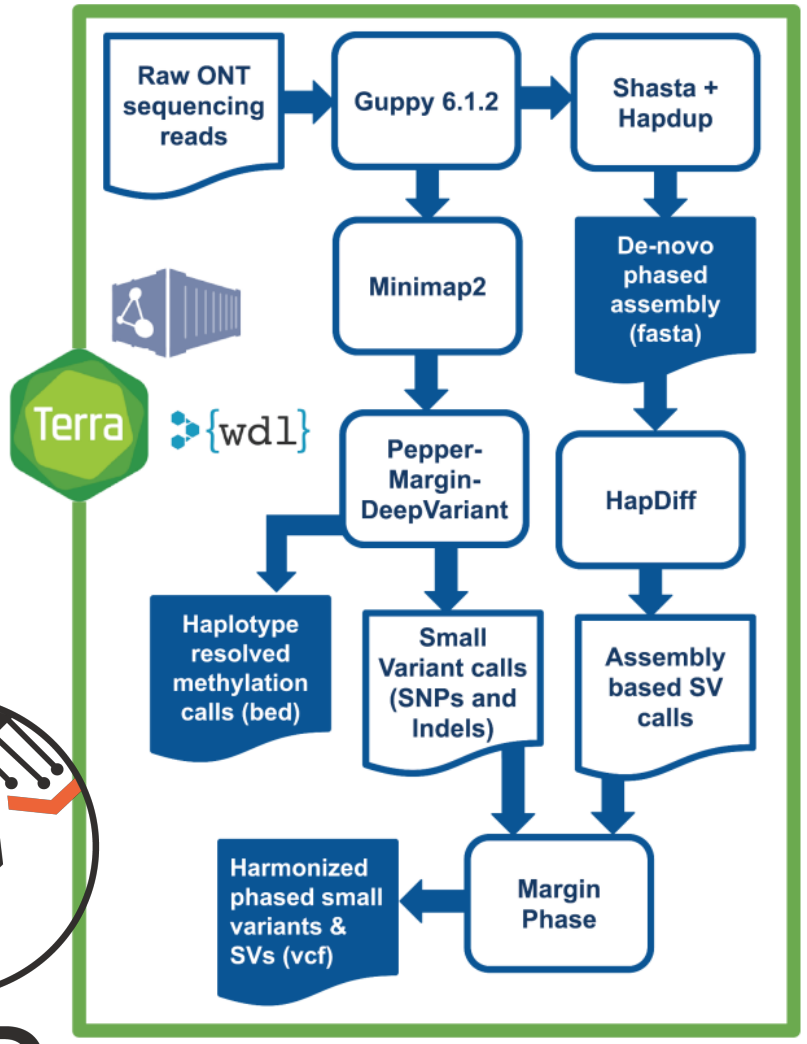
Today: < US$1,000




MPEG-G
PEARL DNA
Compression of DNA Sequencing Data
Analysis of Metagenomic DNA Sequencing Data
Genomic Foundation Models
3D Genome Reconstruction
Edit Distance Embedding
Personalized Medicine
DNA Data Storage
Standardization
Genomic Data Science
SYNTHESIS
EDITING
SEQUENCING
SAMPLE POST PROCESSING SYSTEM
purified synthetic DNA from merged droplets
FILE SELECTION
DISTRIBUTION
plate with embossed microwells
confinement chamber with controlled atmosphere
stainless steel sheet
DRYING
SEALING BY LASER WELDING
heating element
cooling element
LONG TERM STORAGE CONTAINER SYSTEM
P4D
Terra
Haplotype resolved methylation calls (bed)
Small Variant calls (SNPs and Indels)
Assembly based SV calls
Harmonized phased small variants & SVs (vcf)
Margin Phase
Allele frequency

Personalized, predictive, precise and preventive medicine

- Large national study involving **1,000 patients suffering from major depressive disorder** (MDD)
- Bioinformatics pipeline for nanopore sequencing data
- MPEG-G compression for long-read sequencing data
- Data analysis infrastructure: CPU/GPU compute cluster, storage cluster with multiple petabytes



P4D

DNA — the future of long-term data storage

- Researchers in fields as diverse as artificial intelligence, healthcare, astronomy and climate science seek to store **massive data sets for future data mining**.
- Traditional data storage technologies lack longevity, density, and cost-effectiveness.
- Solution: **use DNA as a storage medium** (density: 455 EB = 455,000,000 TB per gram)
- Block-by-block DNA synthesis is the most promising approach for DNA data storage.

PEARL DNA

European Innovation Council

Tutorial Overview

Applied Machine Learning in Genomic and Nutritional Data Science

Sessions

Part 1

Applied ML Foundations

Jan Voges

Tuesday, April 23

13:30–15:30 (2 hours)

Part 2

Nutritional Science Use Case

Fabian Müntefering

Wednesday, April 24

10:15–11:45 (1.5 hours)

These can also be good starting points for the “Code & Beer & Chips” session…

Contents

	Chapter	Notebook(s)
Part 1: Applied Machine Learning Foundations		
	Python Recap	**python-recap**
	Machine Learning Basics	kmer-analysis
	Unsupervised Learning	**population-clustering**
	Supervised Learning	decision-tree-demo
	Neural Network Fundamentals & Design	**pytorch-tutorial** gene-family-classification motif-analysis
Part 2: Nutritional Science Use Case		
	Exploring a Multimodal Nutritional Data Set	**nutritional-data-learning**

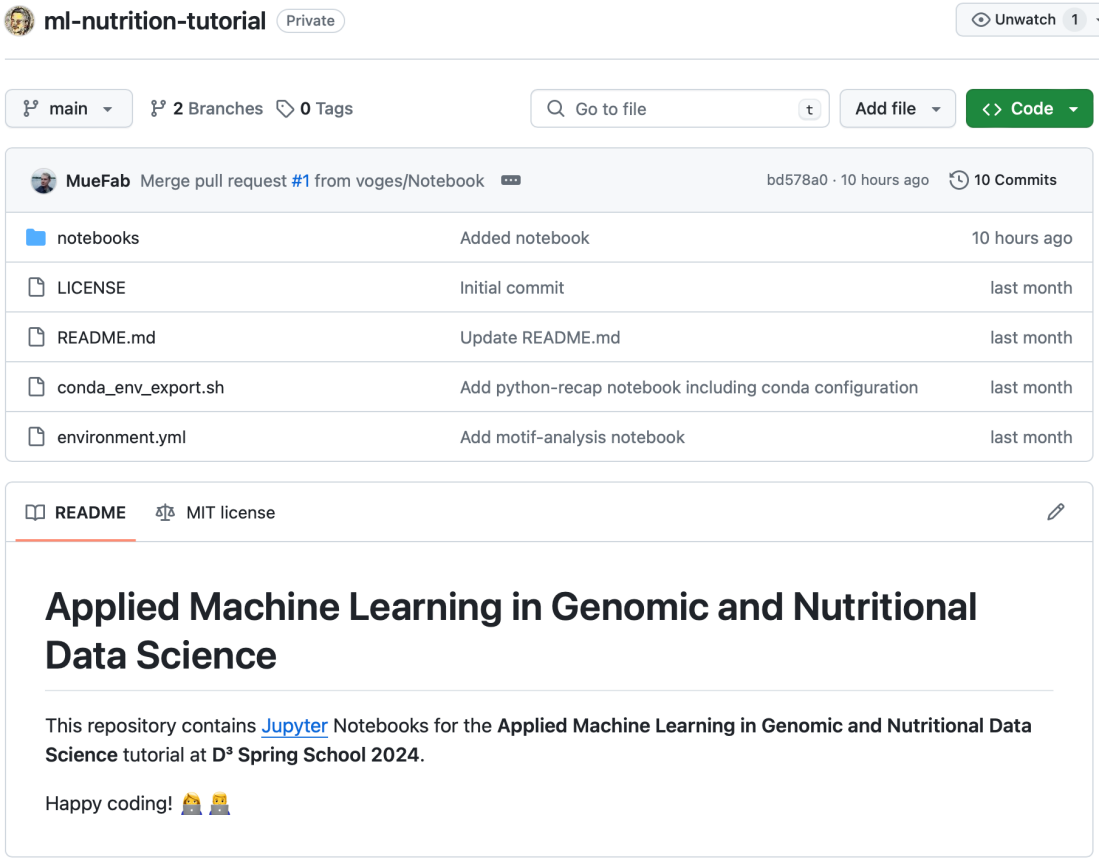
Equipment

→ Python + (ML) libraries + Jupyter

- Google Colab — https://colab.research.google.com
- Own premises — e.g., Visual Studio Code with extensions

Repository

https://github.com/voges/ml-nutrition-tutorial



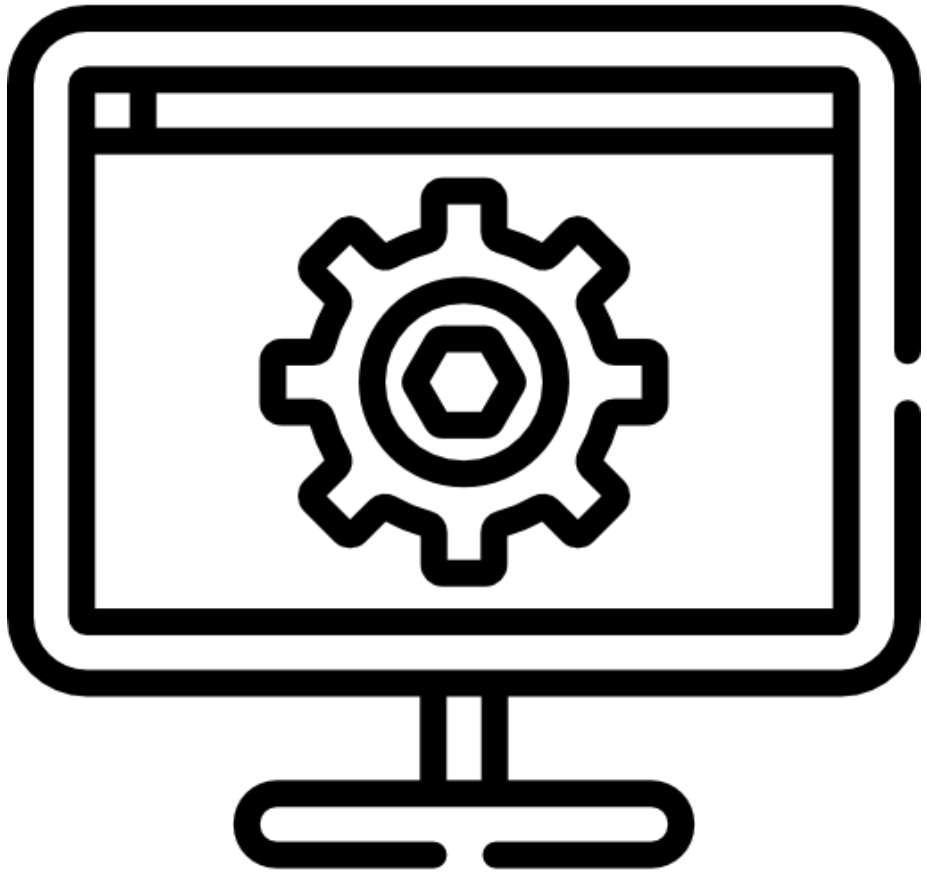
Package management

CONDA®

https://pip.pypa.io/en/stable/getting-started/

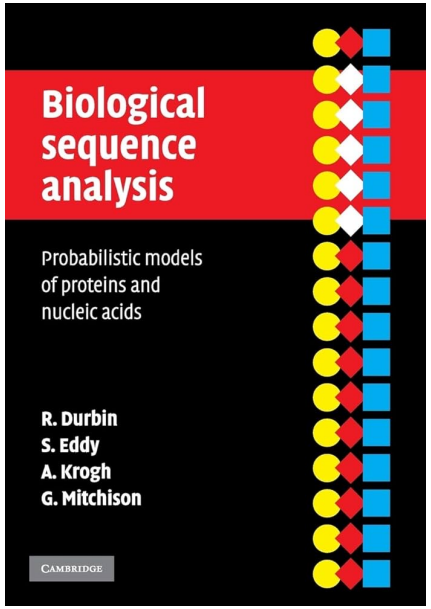
https://conda.io/projects/conda/en/latest/user-guide/install/index.html

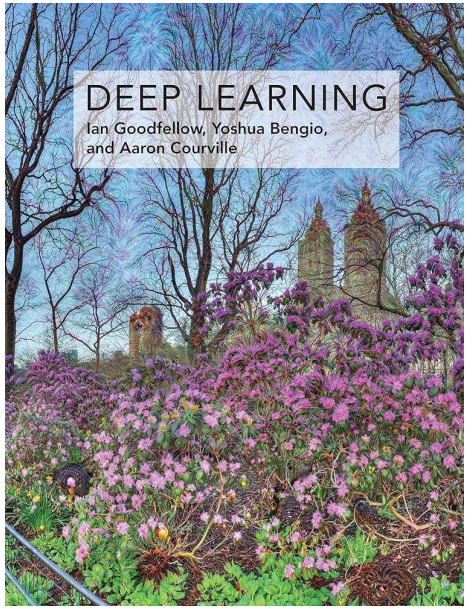
Setup time



Literature

- Durbin et al., Biological sequence analysis, Cambridge University Press
- Goodfellow et al., Deep learning, MIT Press





Python Recap

Reiterating Some Python Basics

Python recap notebook

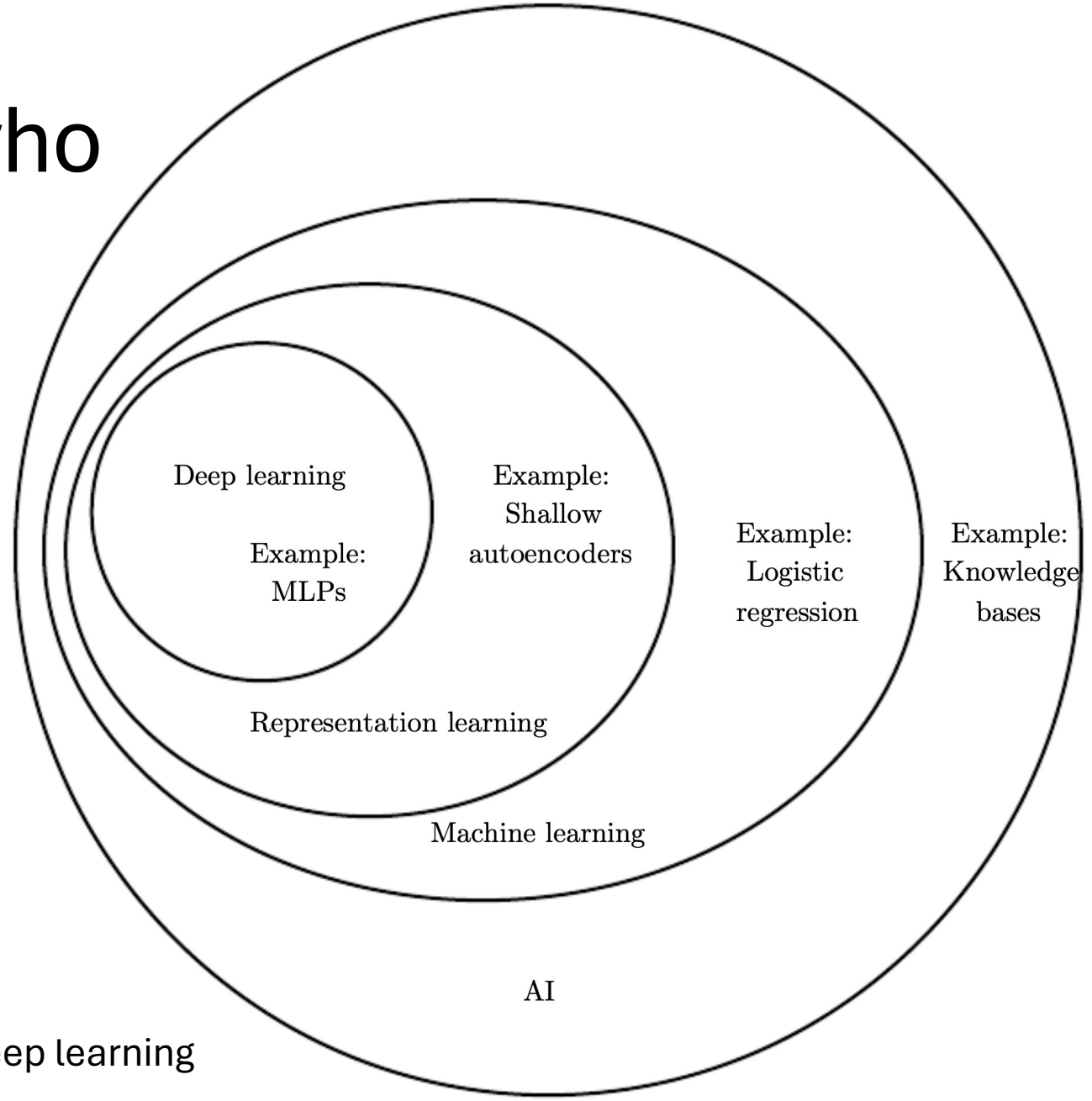
Notebook ‘python-recap’

- Python 3’s f-strings
- Plotting with matplotlib
- A first glimpse into the machine learning world

Machine Learning Basics

Unlocking the Power of Data

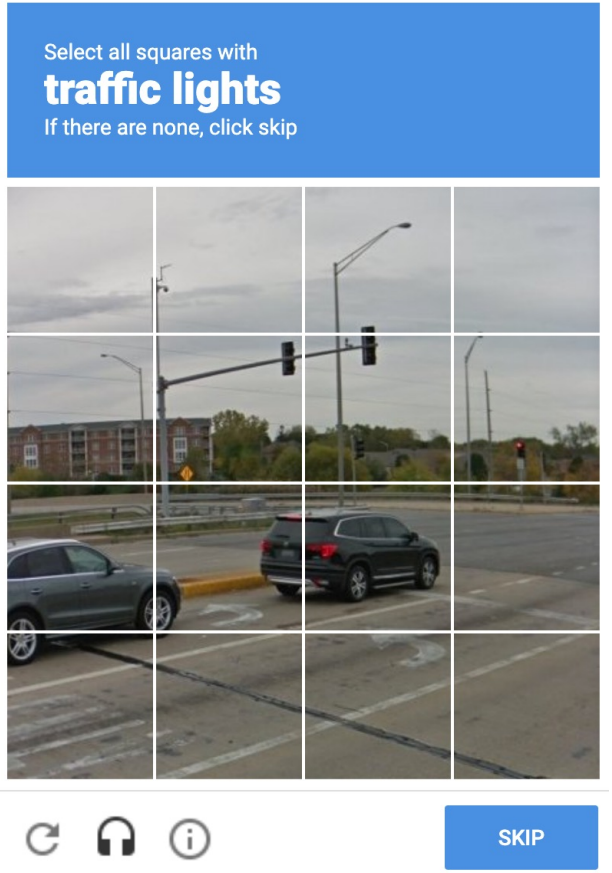
Who is who



Goodfellow et al., Deep learning

Components of ML

- Data
  - Manual collection and curation
  - Automatic collection
  - Or somewhere in between — remember reCAPTCHA, anyone?
- Features
  - Simple: car mileage
  - Obvious: word frequencies
  - Sophisticated/high-level: HOG, edges, …
- Algorithms



Learning = experience + task + performance measure

Mitchell, 1997:

"A computer program is said to learn from **experience** E with respect to some class of **tasks** T and **performance measure** P, if its performance at tasks in T, as measured by P, improves with experience E. "

Experience

Datasets… here are some prominent computer vision datasets:

ImageNet	Labeled object image database, used in the ImageNet Large Scale Visual Recognition Challenge	Labeled objects, bounding boxes, descriptive words, SIFT features
Caltech 101	Pictures of objects.	Detailed object outlines marked.
CIFAR-10	Many small, low-resolution, images of 10 classes of objects.	Classes labeled; training set splits created.
MNIST database	Database of handwritten digits.	Hand-labeled.

Tasks

- **Classification**: the learning algorithm is usually asked to produce a function $f: \mathbb{R}^n \rightarrow \{1, \dots, k\}$. When $y = f(\boldsymbol{x})$, the model assigns an input described by vector $\boldsymbol{x}$ to a category identified by numeric code y. There are other variants of the classification task, for example, where f outputs a probability distribution over classes.
- **Regression**: In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f: \mathbb{R}^n \rightarrow \mathbb{R}$.
- **More tasks**: machine translation, anomaly detection, synthesis and sampling, denoising, density estimation, …

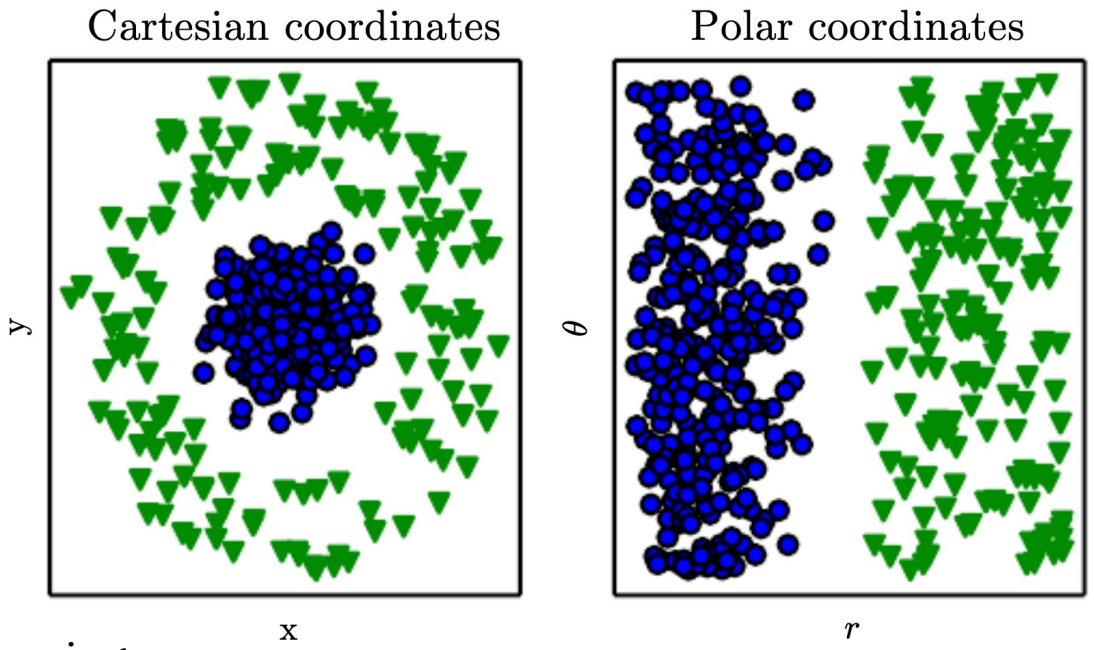
Performance measures

Metrics:

- Mean absolute error (MAE)
- Mean squared error (MSE)
- Accuracy
- Precision & recall
- F1-score
- AUROC
- …

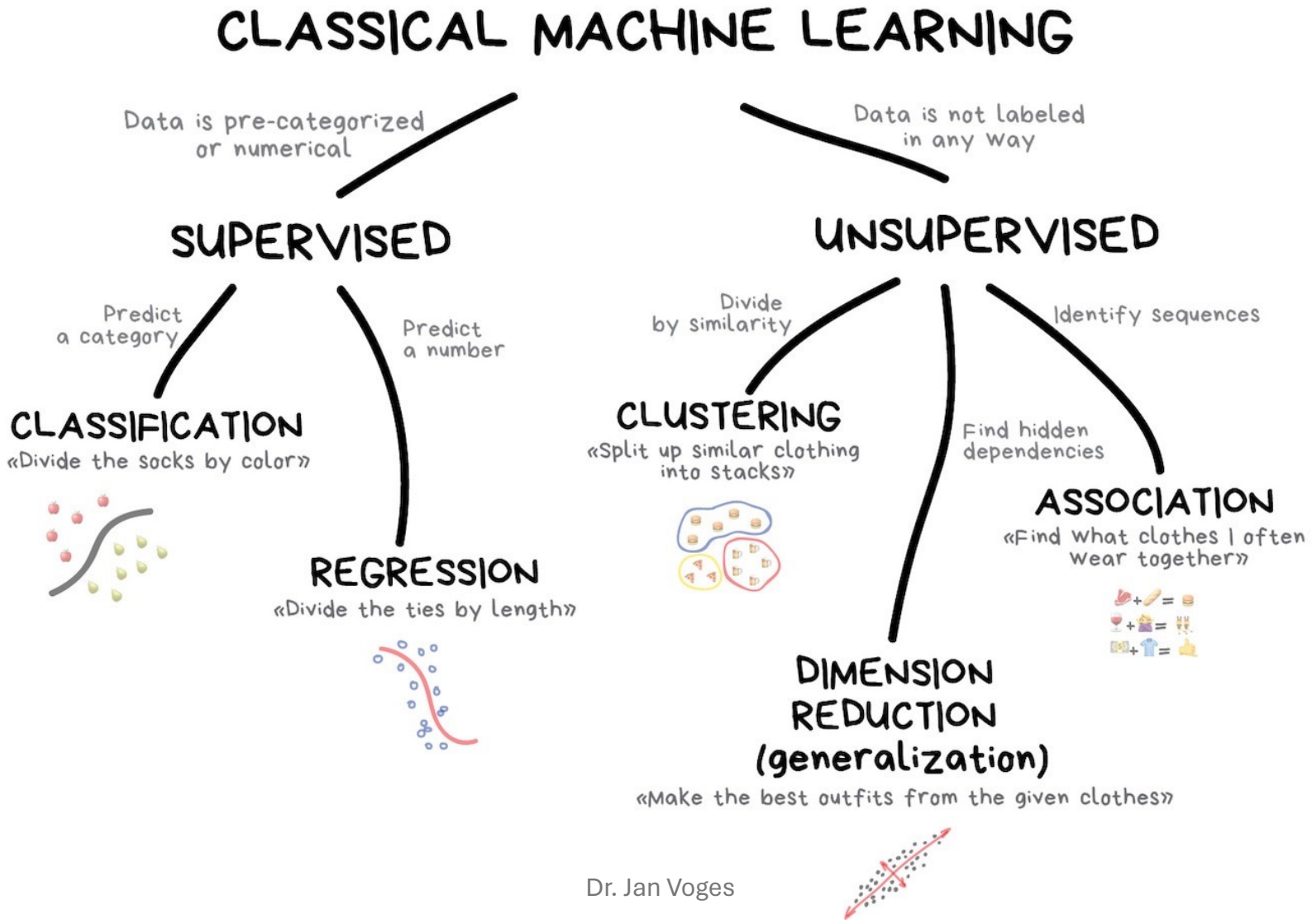
Feature representation is key

Many AI tasks can be solved by designing the **right set of features** to extract for that task, then providing these feature to a simple ML algorithm.



Goodfellow et al., Deep learning

Supervised and unsupervised learning

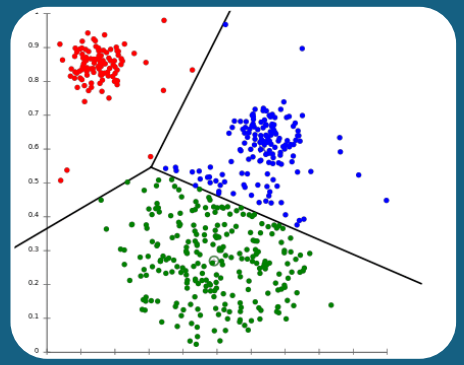


vas3k, 2023

Unsupervised Learning

Discovering Patterns Beyond Labels

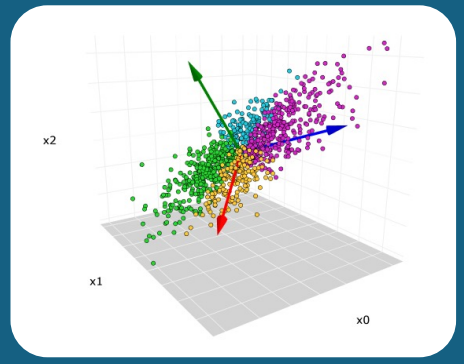
Unsupervised learning examples



Unsupervised learning

↳ Clustering

↳ *k*-means



Unsupervised learning

↳ Dimensionality reduction

↳ PCA

A very high-level view on clustering

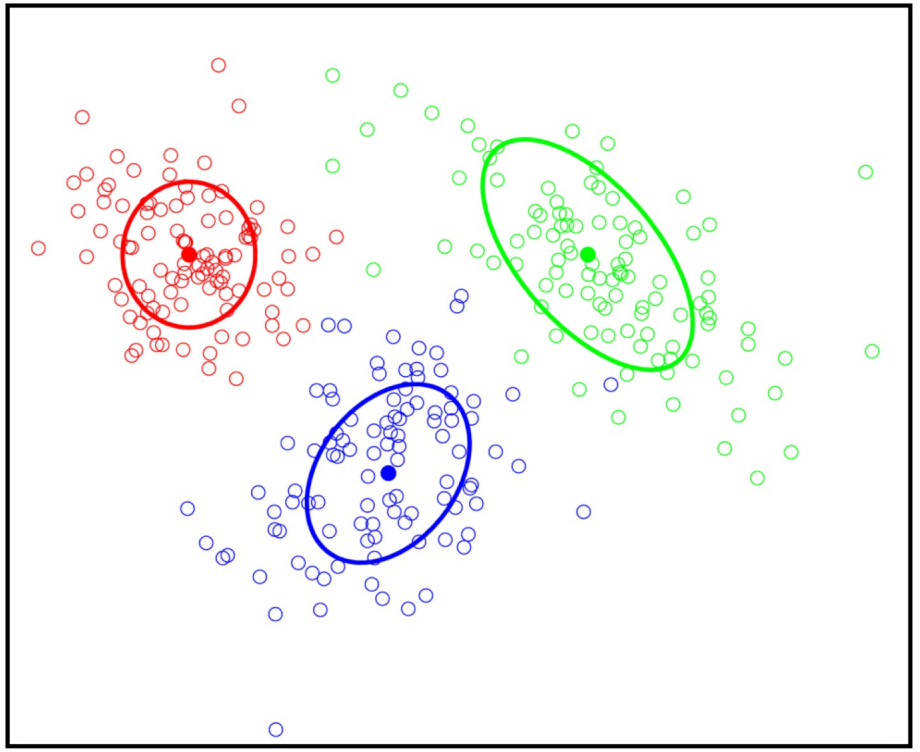
Goal: structure articles by topic, i.e., discover groups (**clusters**) of related articles

SPORTS

WORLD NEWS

Clustering — an unsupervised learning task

No labels provided … uncover cluster structure from input alone!

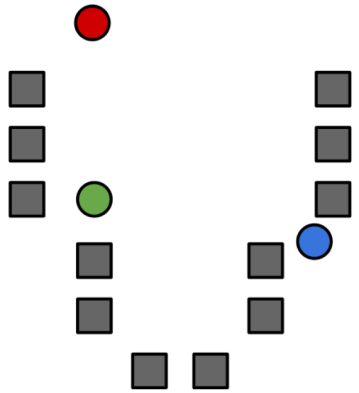


What defines a cluster?

- A cluster is defined by its center & shape/spread.
- Assign observation $\boldsymbol{x}^{(j)}$ to cluster k if score under cluster k is higher than under all other clusters.
- We often define the score as distance to the cluster center, ignoring the shape.

k-means

The *k*-means algorithm works by initializing k different centroids $\{\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(k)}\}$ to different values, then alternating between two different steps until convergence.

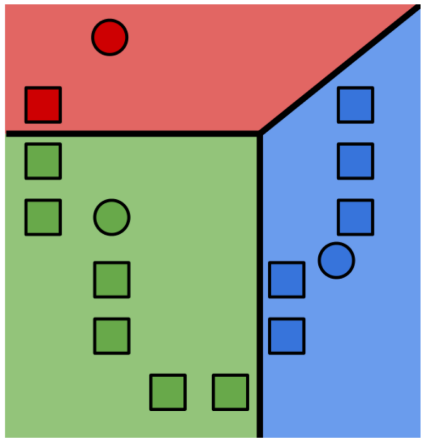


k-means

In the **assignment step**, each training example $\boldsymbol{x}^{(j)}$ is assigned to cluster i, where i is the index of the nearest centroid $\boldsymbol{\mu}^{(i)}$.

Inferred label for observation $\boldsymbol{x}^{(j)}$, where supervised learning has *given labels* $y^{(j)}$.

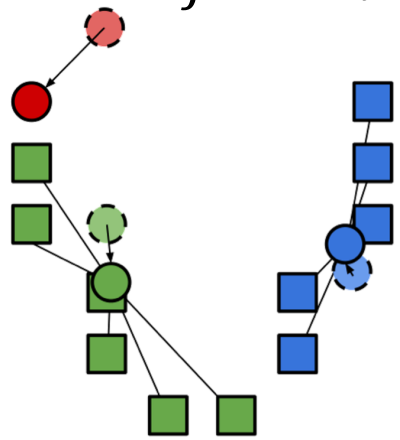
$$\longrightarrow z^{(j)} \leftarrow \underset{i}{\operatorname{argmin}} \left\| \boldsymbol{\mu}^{(i)} - \boldsymbol{x}^{(j)} \right\|_2^2$$



k-means

In the **update step**, each centroid $\boldsymbol{\mu}^{(i)}$ is updated to the mean of all training examples $\boldsymbol{x}^{(j)}$ assigned to cluster i.

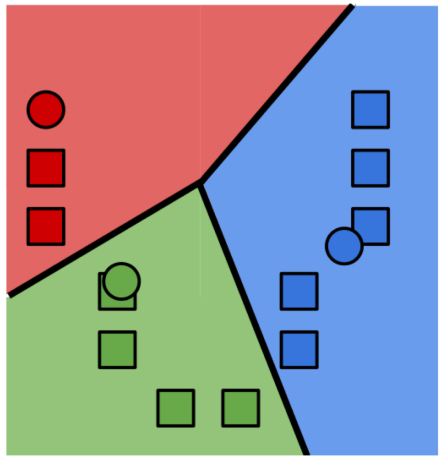
$$\boldsymbol{\mu}^{(i)} = \frac{1}{n_i} \sum_{j:z^{(j)}=i} x^{(j)}$$



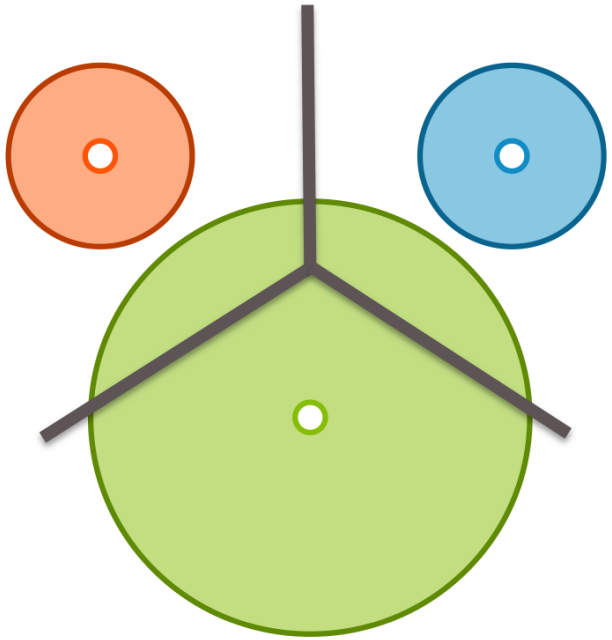
k-means

Initialize cluster centers

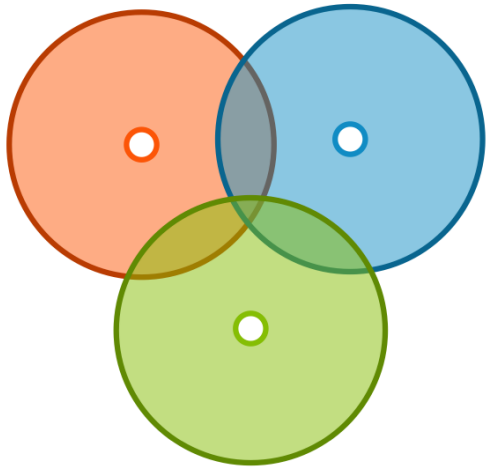
1. Assignment **step**
2. Update **step**
3. Iterate 1.+2. until convergence



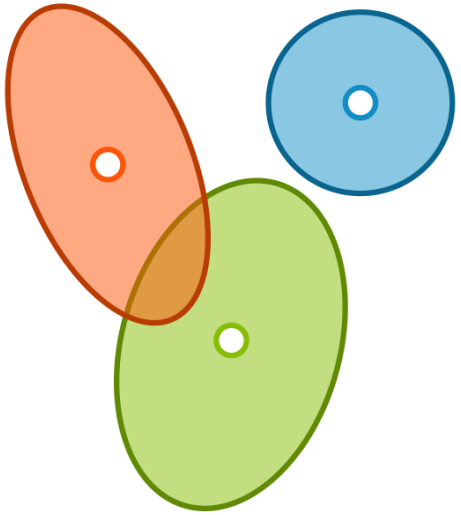
Failure modes of *k*-means



disparate cluster sizes

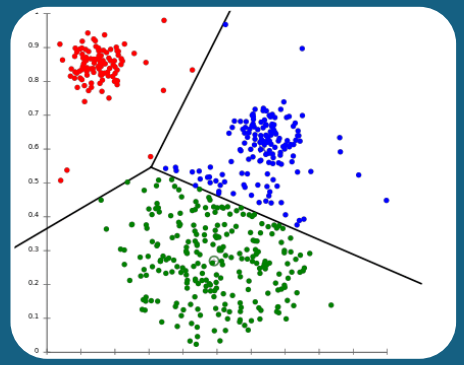


overlapping clusters



different shaped/oriented clusters

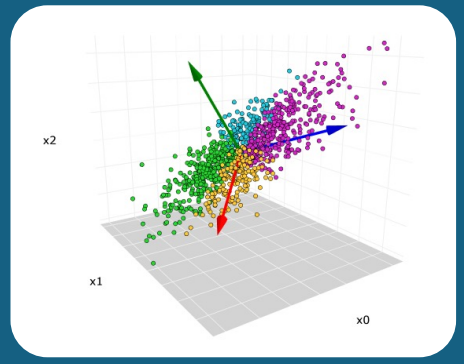
Unsupervised learning examples



Unsupervised learning

↳ Clustering

↳ *k*-means

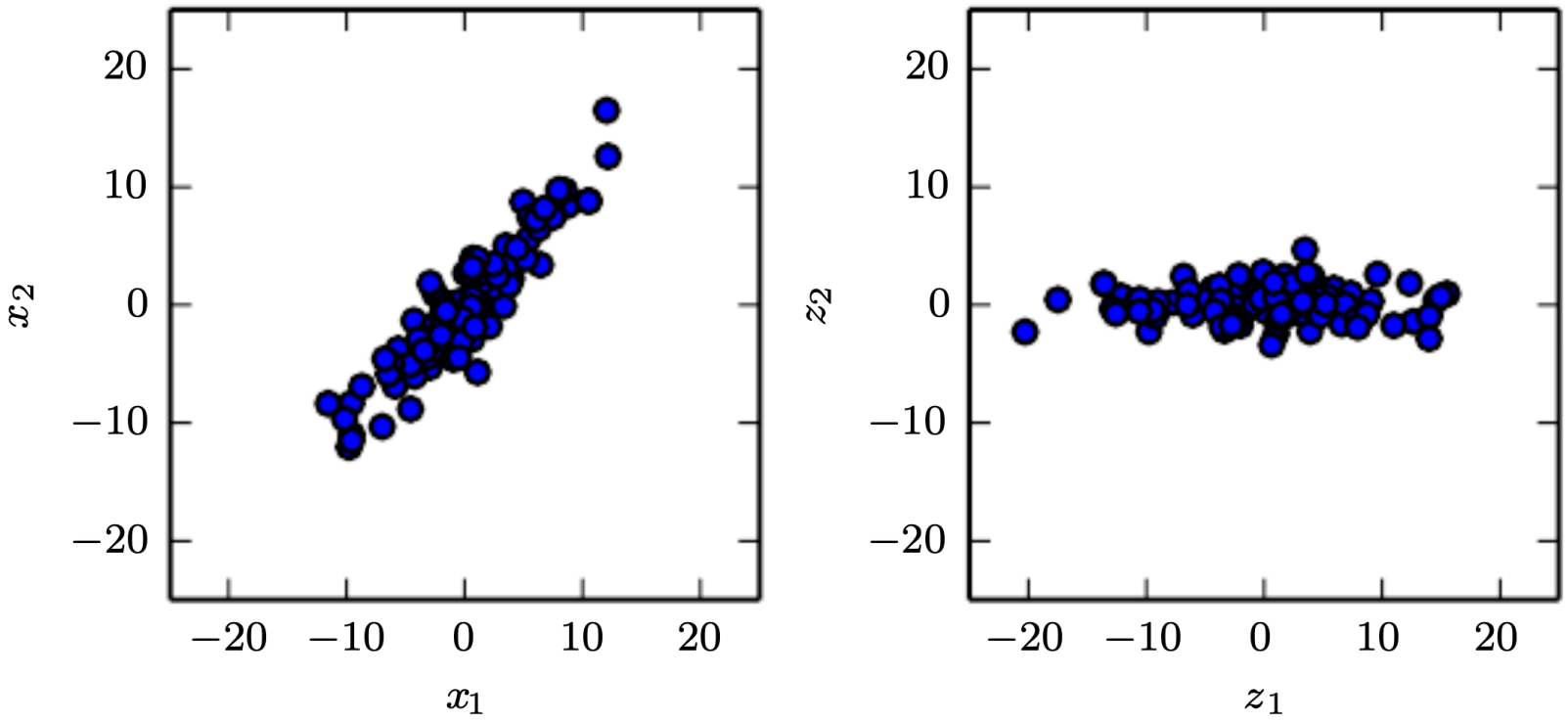


Unsupervised learning

↳ Dimensionality reduction

↳ PCA

Principal component analysis (PCA)



Goodfellow et al., Deep learning

PCA

- Unsupervised learning algorithm that learns a representation of data
- PCA learns an orthogonal, linear transformation of the data that projects an input $\boldsymbol{x}$ to a representation $\boldsymbol{z}$.
- PCA preserves as much of the information as possible as measured by least-squares reconstruction error.

PCA step by step

1. **Standardizing**: standardize the data (zero mean, unit variance)
2. **Eigendecomposition**: obtain the eigenvectors and eigenvalues from the covariance matrix or perform singular value decomposition of the data matrix
3. **Selection of principal components**: sort eigenvalues in descending order and choose the k eigenvectors that correspond to the k largest eigenvalues where k is the number of dimensions of the new feature subspace.
4. **Construction of the projection matrix**: construct the projection matrix from the selected k eigenvectors.
5. **Transformation into new feature subspace**: transform the original data using the projection matrix to a k-dimensional feature subspace.

Eigendecomposition

An eigenvector of a *square* matrix $\boldsymbol{A}$ is a non-zero vector $\boldsymbol{v}$ such that multiplication by $\boldsymbol{A}$ alters only the scale of $\boldsymbol{v}$:

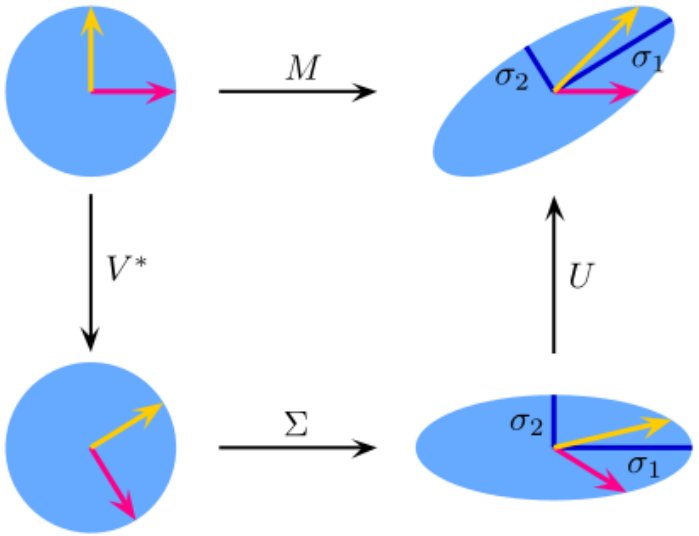
$$\boldsymbol{A}\boldsymbol{v} = \lambda\boldsymbol{v}$$

For example:

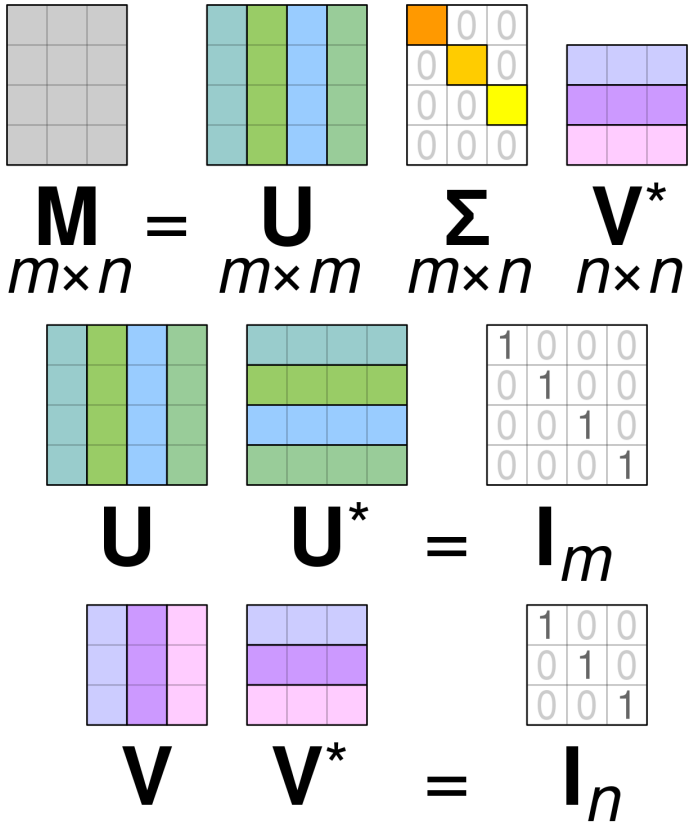
$$\begin{pmatrix} 2 & 3 \\ 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 2 \end{pmatrix} = \begin{pmatrix} 12 \\ 8 \end{pmatrix} = 4 \cdot \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

Singular value decomposition (SVD)

The SVD of $\boldsymbol{X}$ is defined as the decomposition $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$.



$$M = U \cdot \Sigma \cdot V^*$$



Eigendecomposition vs. SVD

Using the SVD of $\boldsymbol{X}$, we can express the covariance matrix of $\boldsymbol{X}$ as:

$$\mathbf{K} = \frac{1}{m-1}\boldsymbol{X}^T\boldsymbol{X} = \frac{1}{m-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T)^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$$

$$= \frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^T\boldsymbol{U}^T\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T = \frac{1}{m-1}\boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^T$$

This is an eigendecomposition of $\boldsymbol{X}^T\boldsymbol{X}$!

We use the fact that $\boldsymbol{U}^T\boldsymbol{U} = \boldsymbol{I}$, because $\boldsymbol{U}$ is defined to be orthogonal.

So, we can conclude that obtaining an eigendecomposition of the covariance matrix $\boldsymbol{K}$ can also be achieved by performing an SVD on the data matrix $\boldsymbol{X}$.

PCA decorrelates features

If we now transform the data taking $\boldsymbol{z} = \boldsymbol{x}^T \boldsymbol{W}$, we see that the covariance matrix $\boldsymbol{Z}^T \boldsymbol{Z}$ is diagonal:

$$\boldsymbol{Z}^T \boldsymbol{Z} = \boldsymbol{W}^T \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{W} = \boldsymbol{W}^T \boldsymbol{W} \boldsymbol{\Sigma}^2 \boldsymbol{W}^T \boldsymbol{W} = \boldsymbol{\Sigma}^2$$

We use the fact that $\boldsymbol{W}^T \boldsymbol{W} = \boldsymbol{I}$, because $\boldsymbol{W}$ is defined to be orthogonal.

PCA decorrelates features

- If we project the data matrix $\boldsymbol{X}$ to $\boldsymbol{Z}$ via the linear transformation $\boldsymbol{W}$, the resulting representation has a diagonal covariance matrix. Hence the individual elements of $\boldsymbol{Z}$ are mutually uncorrelated.
- We achieved a representation that attempts to **disentangle the unknown factors of variation** underlying the data.
- In the case of PCA, this disentangling takes the form of finding a rotation of the input space (described by $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{Z}$.

Unsupervised learning notebook

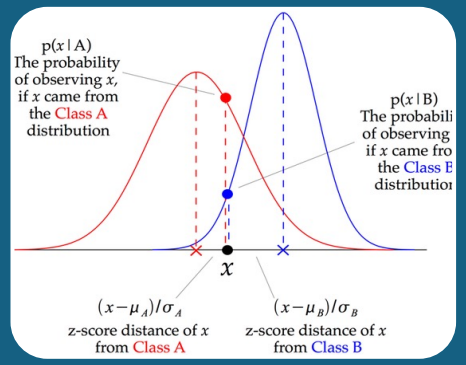
Notebook ‘population-clustering’

- Data preprocessing
- PCA step by step

Supervised Learning

Unveiling the Mentorship of Data

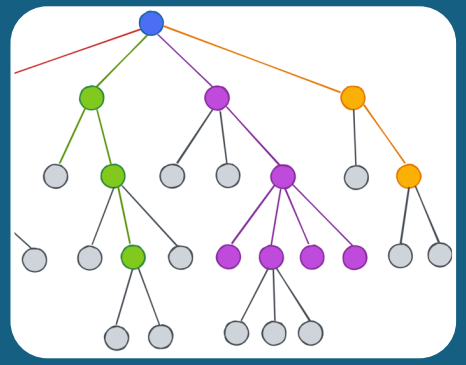
Supervised learning examples



Supervised learning

↳ Classification

↳ Naïve Bayes classifier

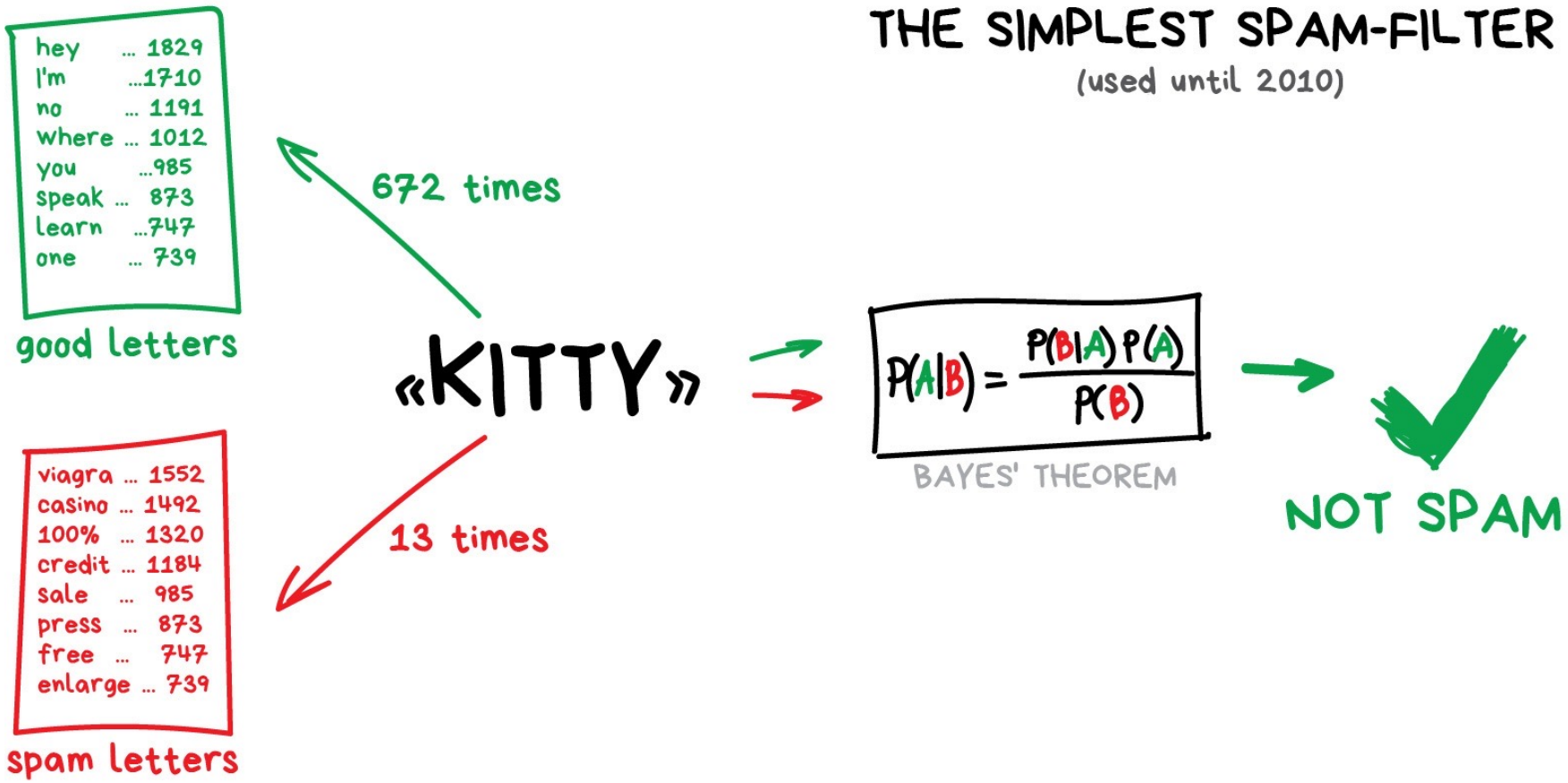


Supervised learning

↳ Classification

↳ Decision trees

Naïve Bayes classifier



vas3k, 2023

Naïve Bayes classifier

Bayes' theorem is often applied in a scenario where we want to infer a **class label** c_k based on observing some **features** $f_1, \dots, f_n$:

$$P(c_k|f_1, \dots, f_n) = \frac{P(f_1, \dots, f_n|c_k) \cdot P(c_k)}{P(f_1, \dots, f_n)}$$

In plain English, the theorem can be rephrased as:

$$\text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

Naïve Bayes classifier

$$P(c_k|f_1, \dots, f_n) = \frac{P(f_1, \dots, f_n|c_k) \cdot P(c_k)}{P(f_1, \dots, f_n)}$$

The **evidence** $P(f_1, \dots, f_n)$ can be computed using the **law of total probability**:

$$P(f_1, \dots, f_n) = \sum_i P(f_1, \dots, f_n|c_i) \cdot P(c_i)$$

It does not depend on the classes c_i, and as the values of the features are given, it is effectively constant.

Naïve Bayes classifier

$$P(c_k|f_1, \dots, f_n) = \frac{P(f_1, \dots, f_n|c_k) \cdot P(c_k)}{P(f_1, \dots, f_n)}$$

It is therefore sufficient to disregard the denominator and to work out the *relative* **posterior**:

$$P(c_k|f_1, \dots, f_n) \propto P(f_1, \dots, f_n|c_k) \cdot P(c_k)$$

Naïve Bayes classifier

$$P(c_k|f_1, \ldots, f_n) \propto P(f_1, \ldots, f_n|c_k) \cdot P(c_k)$$

Note: the relative posterior distribution can easily be normalized using, e.g., the softmax function:

$$\boldsymbol{\sigma}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Naïve Bayes classifier

$$P(c_k|f_1, \dots, f_n) \propto P(f_1, \dots, f_n|c_k) \cdot P(c_k)$$

Now the *naïve* conditional independence assumption comes into play: assume that the features $f_1, \dots, f_n$ are *mutually independent*, conditional on the class c_k.

We can write the right side of the equation also as joint probability:

$$P(f_1, \dots, f_n|c_k) \cdot P(c_k) = P(f_1, \dots, f_n, c_k)$$

Naïve Bayes classifier

$$P(f_1, \dots, f_n | c_k) \cdot P(c_k) = P(f_1, \dots, f_n, c_k)$$

We can then apply the chain rule for repeated application of the definition of conditional probability:

$$P(f_1, \dots, f_n, c_k) = P(f_1 | f_2, \dots, f_n, c_k) \cdot P(f_2, \dots, f_n, c_k)$$

$$= P(f_1 | f_2, \dots, f_n, c_k) \cdot P(f_2 | f_3, \dots, f_n, c_k) \cdots P(f_n | c_k) \cdot P(c_k)$$

Naïve Bayes classifier

$$P(f_1|f_2, \dots, f_n, c_k) \cdot P(f_2|f_3, \dots, f_n, c_k) \cdots P(f_n|c_k) \cdot P(c_k)$$

Now we assume that the features $f_1, \dots, f_n$ are *mutually independent*, conditional on the class c_k:

$$P(f_i|f_{i+1}, \dots, f_n, c_k) = P(f_i|c_k)$$

Naïve Bayes classifier

Then the distribution of the relative posterior simplifies to:

$$P(c_k|f_1, \ldots, f_n) \propto P(c_k) \cdot \prod_j P\left(f_j|c_k\right)$$

This gives the following classification rule:

$$\hat{c} = \underset{c_k}{\operatorname{argmax}}\, P(c_k) \cdot \prod_j P\left(f_j|c_k\right)$$

This is known as maximum a posteriori (MAP) estimation.

Naïve Bayes classifier

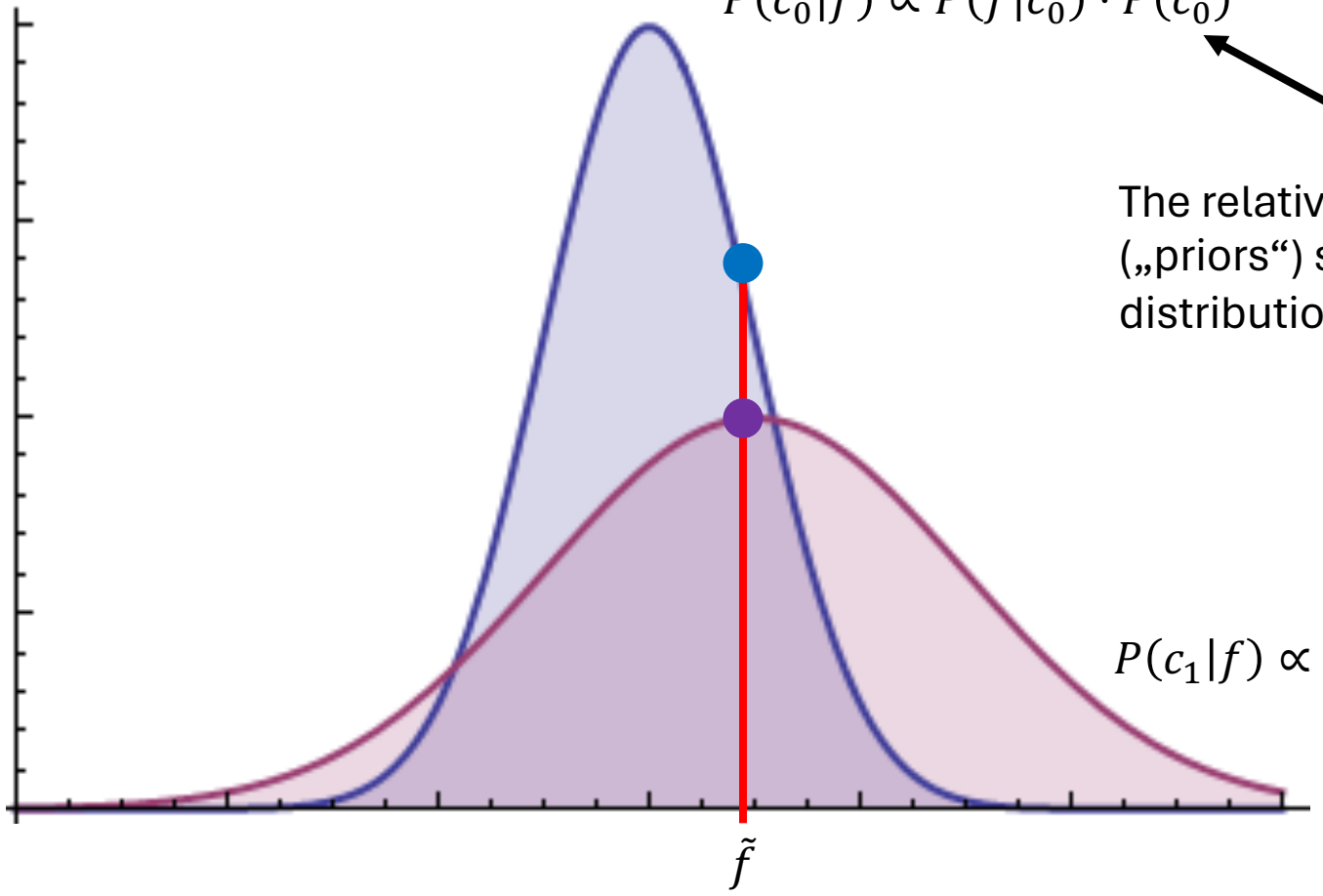
Several simplifications give the following classification rule:

$$\hat{c} = \underset{c_k}{\operatorname{argmax}} P(c_k) \cdot \prod_j P\left(f_j \middle| c_k\right)$$

This is known as maximum a posteriori (MAP) estimation.

Naïve Bayes classifier

$$P(c_0|f) \propto P(f|c_0) \cdot P(c_0)$$



The relative class frequencies („priors“) scale the posterior distributions vertically.

$$P(c_1|f) \propto P(f|c_1) \cdot P(c_1)$$

$$\hat{c} = \underset{\{c_0,c_1\}}{\operatorname{argmax}}\{P(f = \tilde{f}|c_0) \cdot P(c_0), P(f = \tilde{f}|c_1) \cdot P(c_1)\}$$

Naïve Bayes classifier

We could in principle further simplify and assume the relative frequencies of all classes in the training set are equal, i.e.:

$$P(c_k) = \frac{1}{K} \forall c_k$$

Then, we can further simplify to:

$$P(c_k | f_1, \ldots, f_n) \propto \prod_j P\left(f_j | c_k\right)$$

Naïve Bayes classifier

$$P(c_k|f_1, \ldots, f_n) \propto \prod_j P(f_j|c_k)$$

And the following classification rule is known as maximum likelihood (ML) estimation:

$$\hat{c} = \underset{c_k}{\operatorname{argmax}} \prod_j P(f_j|c_k)$$

Naïve Bayes classifier

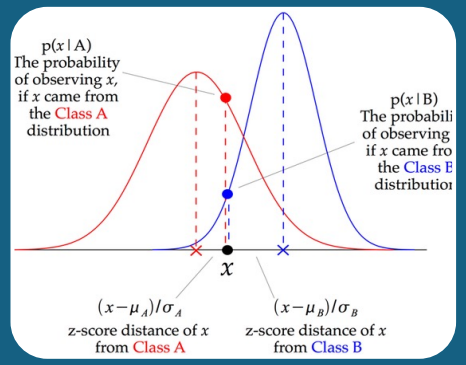
Maximum a posteriori (MAP) estimation:

$$\hat{c} = \underset{c_k}{\operatorname{argmax}} P(c_k) \cdot \prod_j P\left(f_j \middle| c_k\right)$$

Maximum likelihood (ML) estimation:

$$\hat{c} = \underset{c_k}{\operatorname{argmax}} \prod_j P\left(f_j \middle| c_k\right)$$

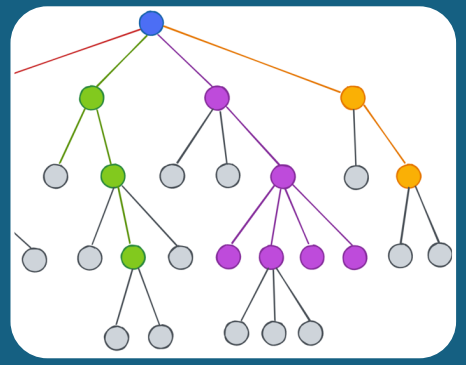
Supervised learning examples



Supervised learning

↳ Classification

↳ Naïve Bayes classifier

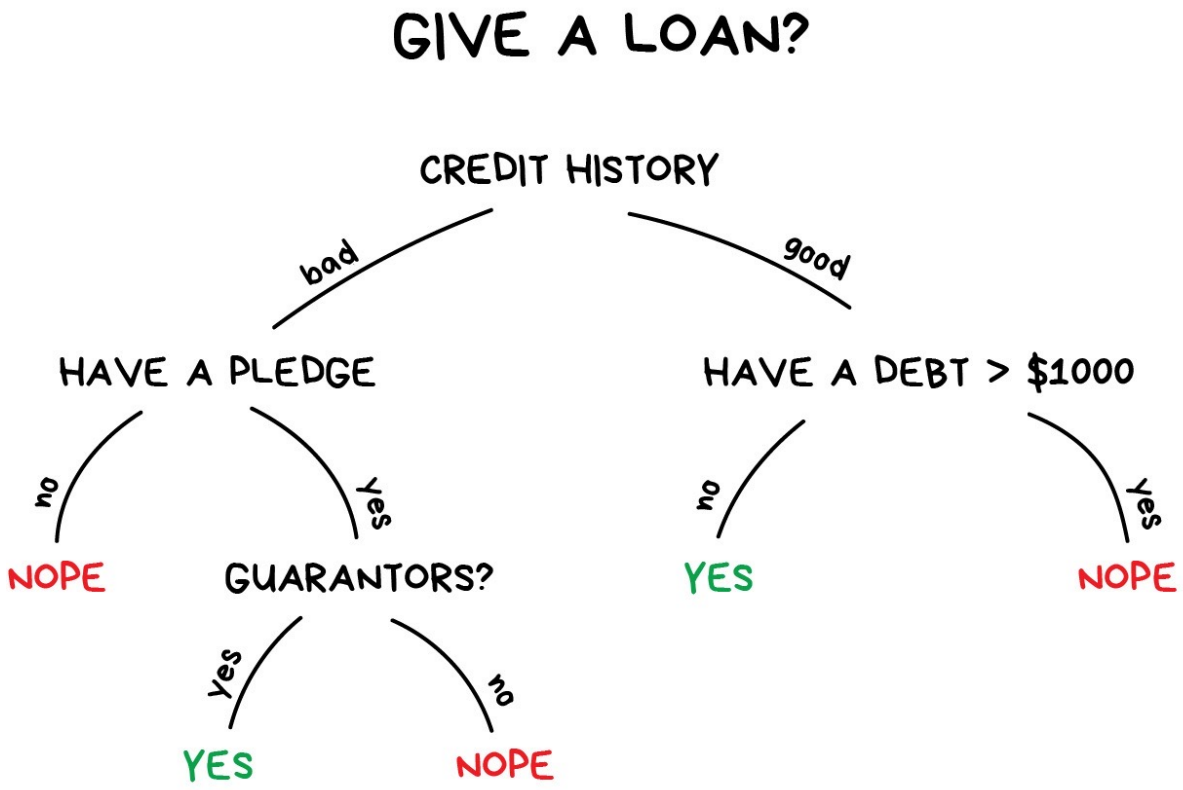


Supervised learning

↳ Classification

↳ Decision trees

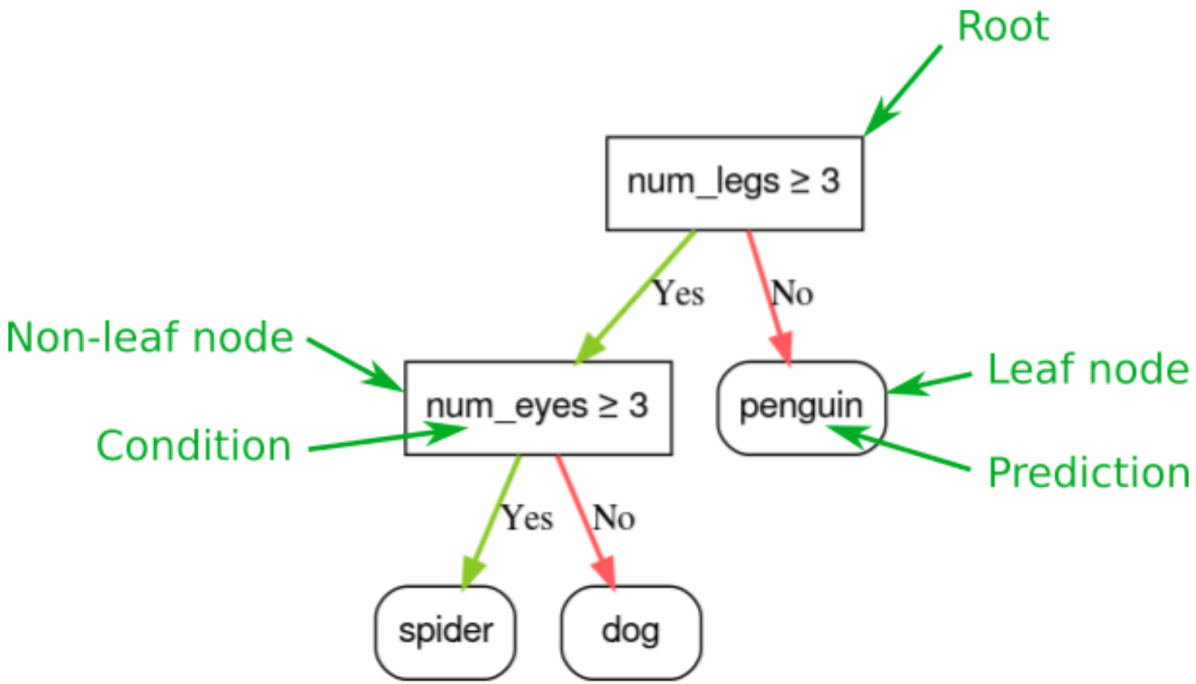
A very simple decision tree



vas3k, 2023

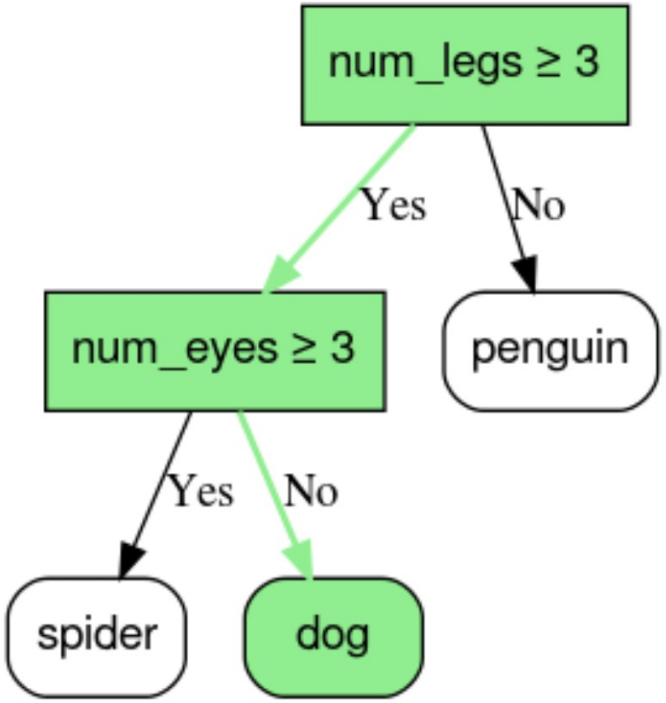
A tree of questions

A decision tree is a model composed of a collection of “questions” organized hierarchically in the shape of a tree. The questions are usually called a **condition**, a split, or a test.

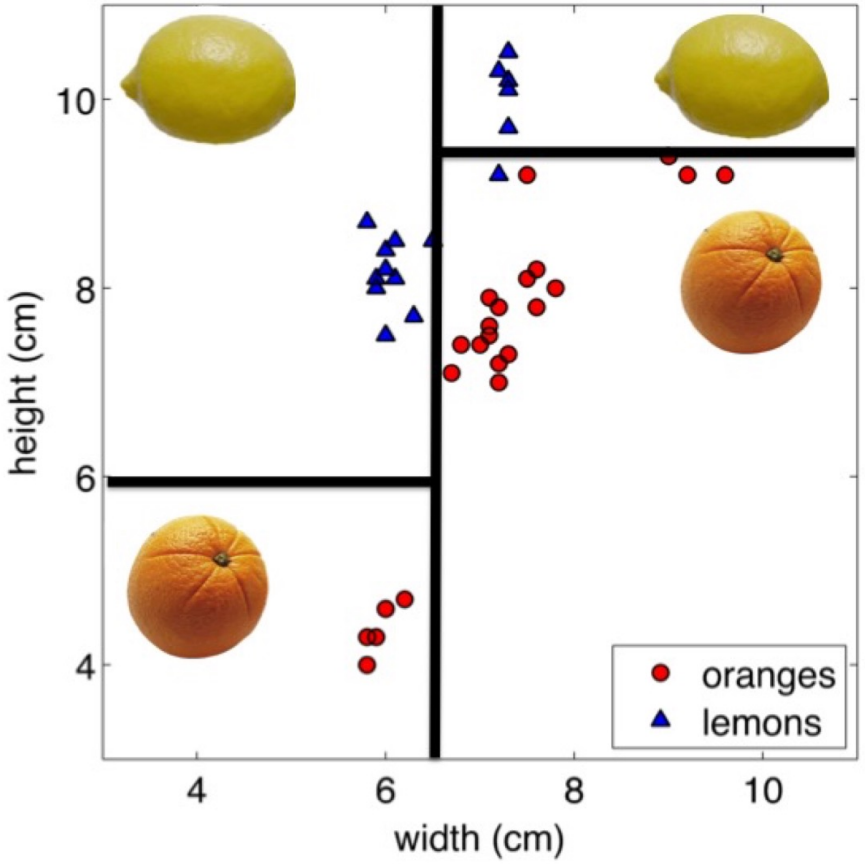


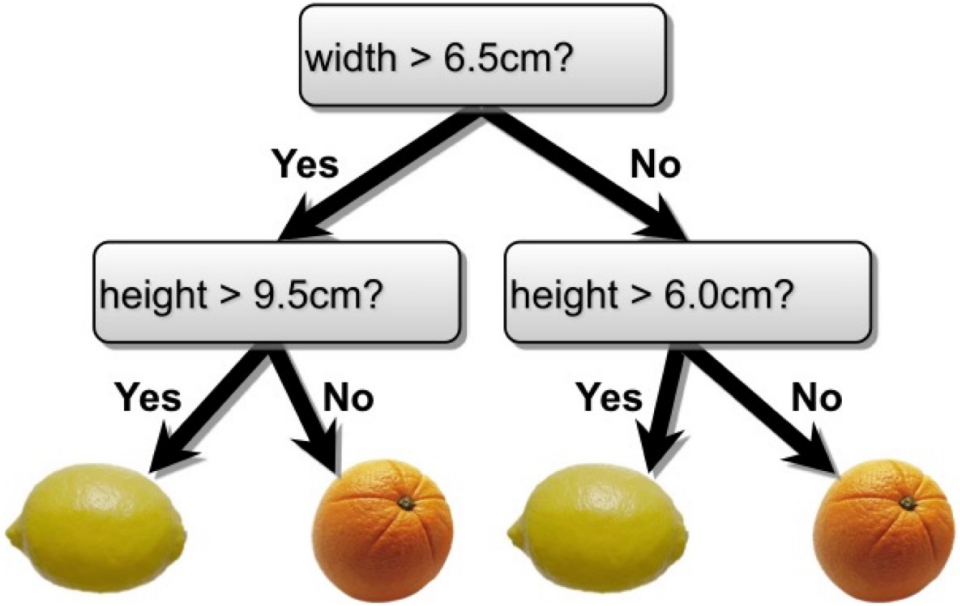
Inference

1. `num_legs ≥ 3 → Yes`
2. `num_eyes ≥ 3 → No`

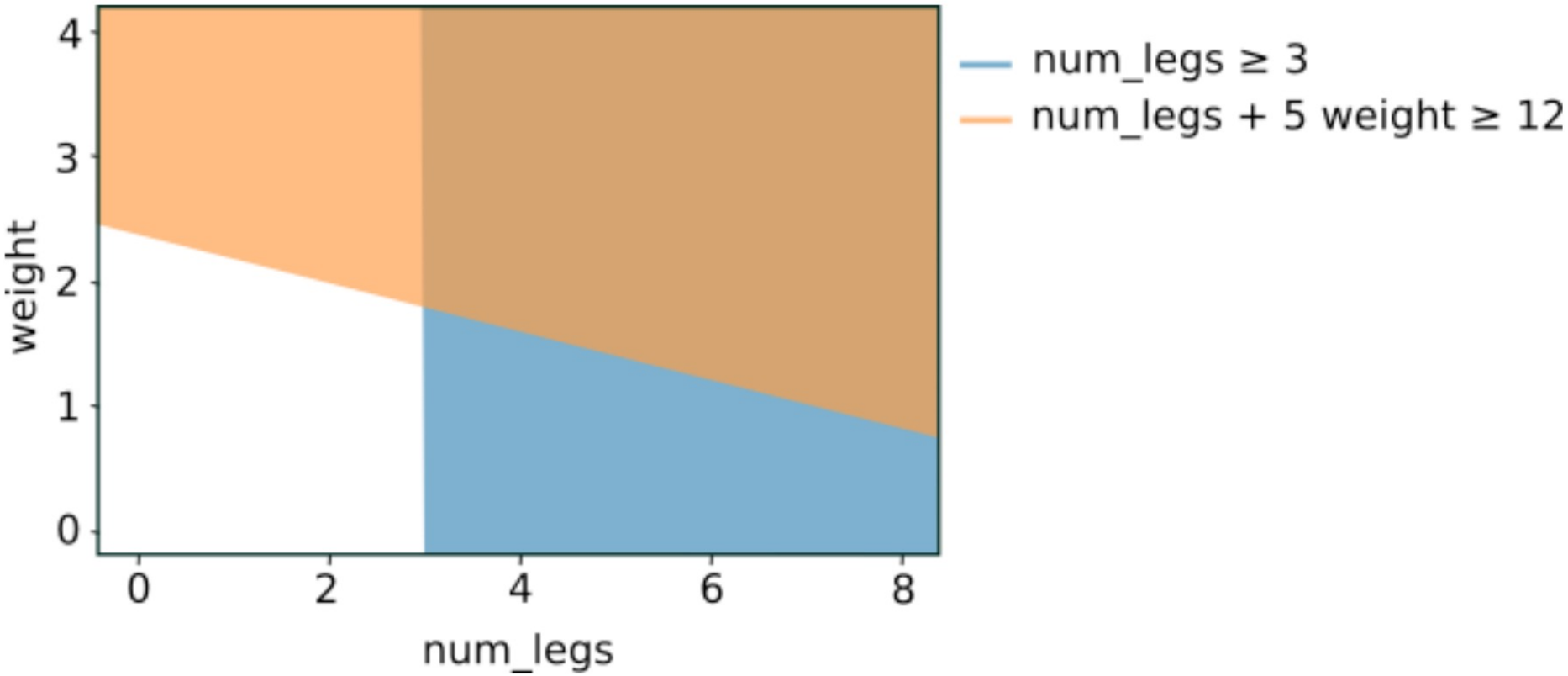


Axis-aligned conditions



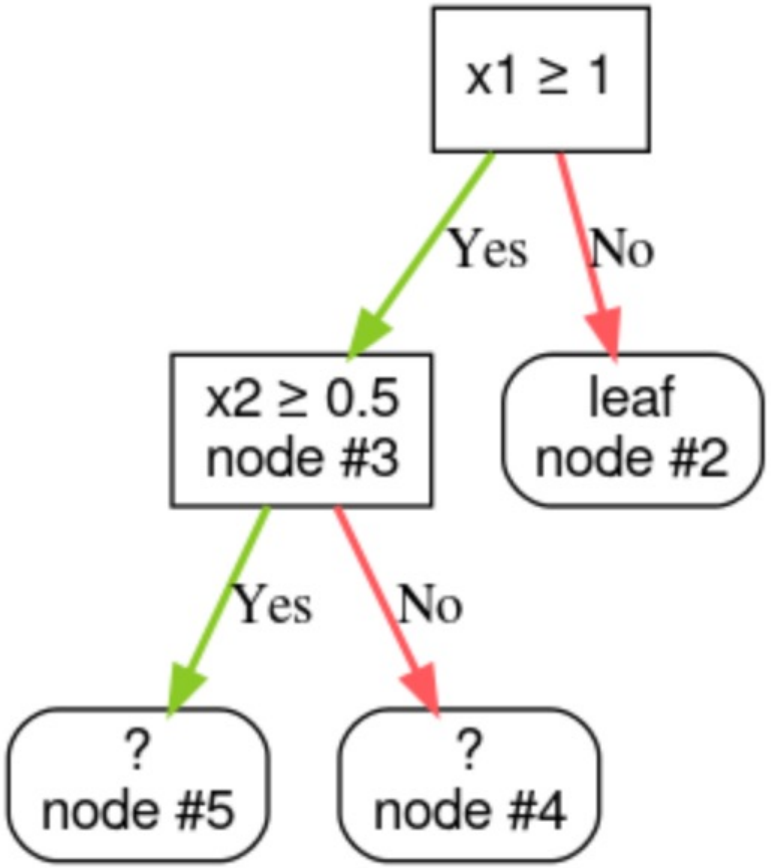


Oblique conditions



Training decision trees

1. Create a root.
2. Grow node #1. The condition “x1 ≥ 1” was found. Two child nodes are created.
3. Grow node #2. No satisfying conditions were found. So, make the node into a leaf.
4. Grow node #3. The condition “x2 ≥ 0.5” was found. Two child nodes are created.



The splitter

- The routine responsible for finding the best condition is called the splitter. Because it needs to test a lot of possible conditions, splitters are the bottleneck when training a decision tree.
- The score maximized by the splitter depends on the task. For example:
  - Information gain and Gini index are commonly used for classification.
  - Mean squared error is commonly used for regression.
- We will look more closely at **information gain**…

Information gain

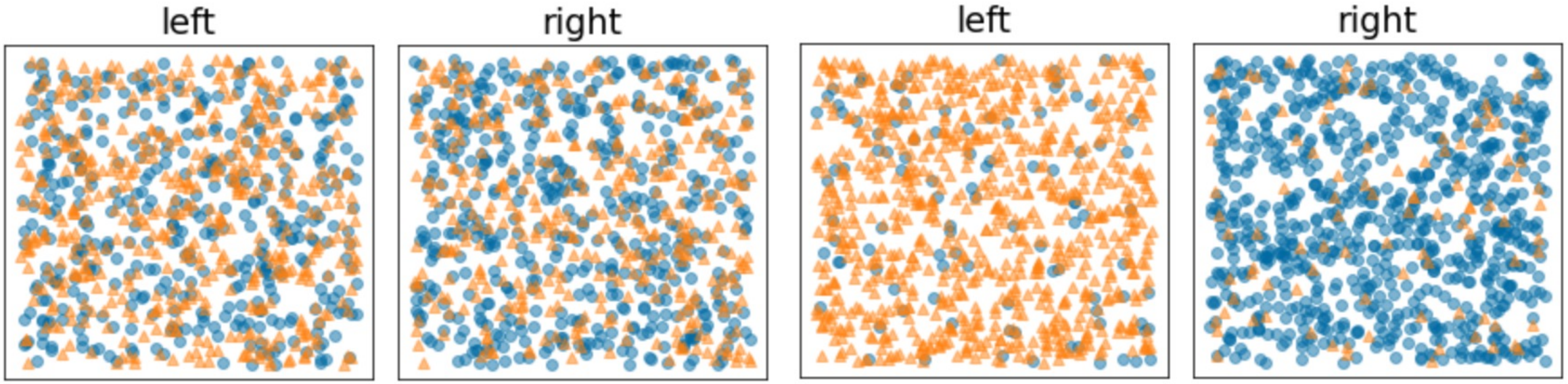
The information gain of a random variable x obtained from an observation of a random variable a taking value $\mathrm{a} = a$ is defined as:

$$IG(\mathrm{x}, \mathrm{a} = a) = H(\mathrm{x}) - H(\mathrm{x}|\mathrm{a} = a)$$

By learning $\mathrm{a} = a$ about x, our uncertainty about x can be reduced, i.e.:

$$\mathrm{IG}(\mathrm{x}|\mathrm{a} = a) \geq 0$$

Information gain



Bad split

Good split

Overfitting & pruning

To limit overfitting a decision tree, apply one or both of the following regularization criteria while training the decision tree:

- Set a maximum depth: prevent decision trees from growing past a maximum depth, such as 10.
- Set a minimum number of examples in leaf: a leaf with less than a certain number of examples will not be considered for splitting.

Neural Network Fundamentals & Design

Basics, Architectures, Loss, and More

What we will cover

- Deep learning basics
- Defining a neural network architecture
- Defining a loss function
- Optimizing the loss function

Machine learning framework

ML framework = data-driven learning of a mapping from in- to output

“Traditional” ML approaches:

Input → **Feature extractor** → **Machine learning model** → **Output**

(e.g., ) (e.g., color and texture) (e.g., decision tree) (e.g., benign or malignant)

Machine learning framework

"Traditional" ML approaches:

Input → **Feature extractor** → **Machine learning model** → **Output**

(e.g., ) (e.g., color and texture) (e.g., decision tree) (e.g., benign or malignant)

Deep learning:

Directly learns what are useful (and better) features from the training data

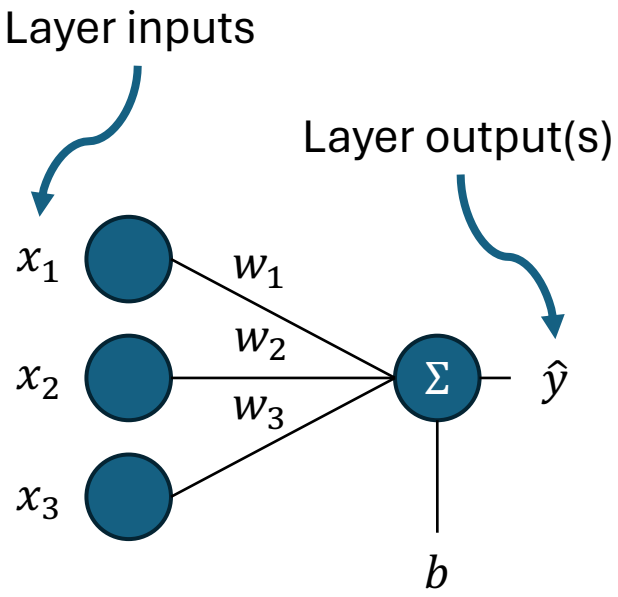
Input → **Deep learning model** → **Output**

(e.g., ) (e.g., convolutional neural network) (e.g., benign or malignant)

Defining a neural network architecture

- Our first architecture: a single-layer, fully connected neural network
- Fully connected = all inputs of a layer are connected to all outputs of a layer
- For simplicity, we will use a 3-dimensional input

Defining a neural network architecture



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \boldsymbol{w}^T \boldsymbol{x} + b$

Neural network parameters: $W = \{\boldsymbol{w}, b\}$

Often refer to all parameters together as just "weights". Bias is implicitly assumed.

Layer weights

Layer bias

Bias term (allows constant shift)

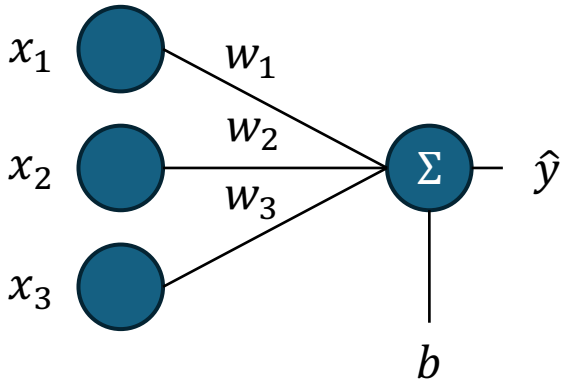
Caveats:

- Single layer still "shallow", not yet a "deep" neural network
- Equivalent to a linear regression model (but a useful base case)

Defining a loss function

Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \boldsymbol{w}^T \boldsymbol{x} + b$

Neural network parameters: $W = \{\boldsymbol{w}, b\}$



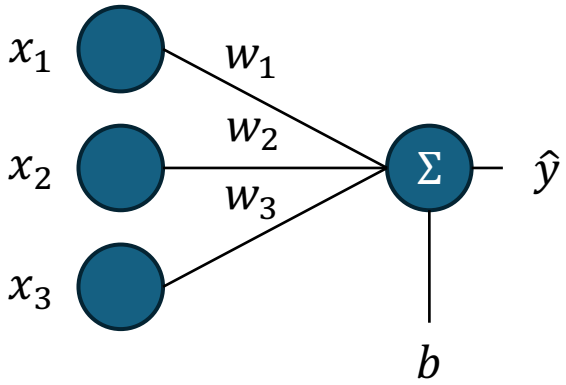
Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how "good" the model parameters are).

Here, we will use the mean squared error (MSE) loss, targeting a regression use case.

Defining a loss function

Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \boldsymbol{w}^T \boldsymbol{x} + b$

Neural network parameters: $W = \{\boldsymbol{w}, b\}$



MSE loss for a single example $\boldsymbol{x}^{(i)}$ when the prediction is $\hat{y}^{(i)}$ and the correct (ground truth) output is $y^{(i)}$:

$$L^{(i)}(W) = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

The loss is small when the prediction is close to the ground truth.

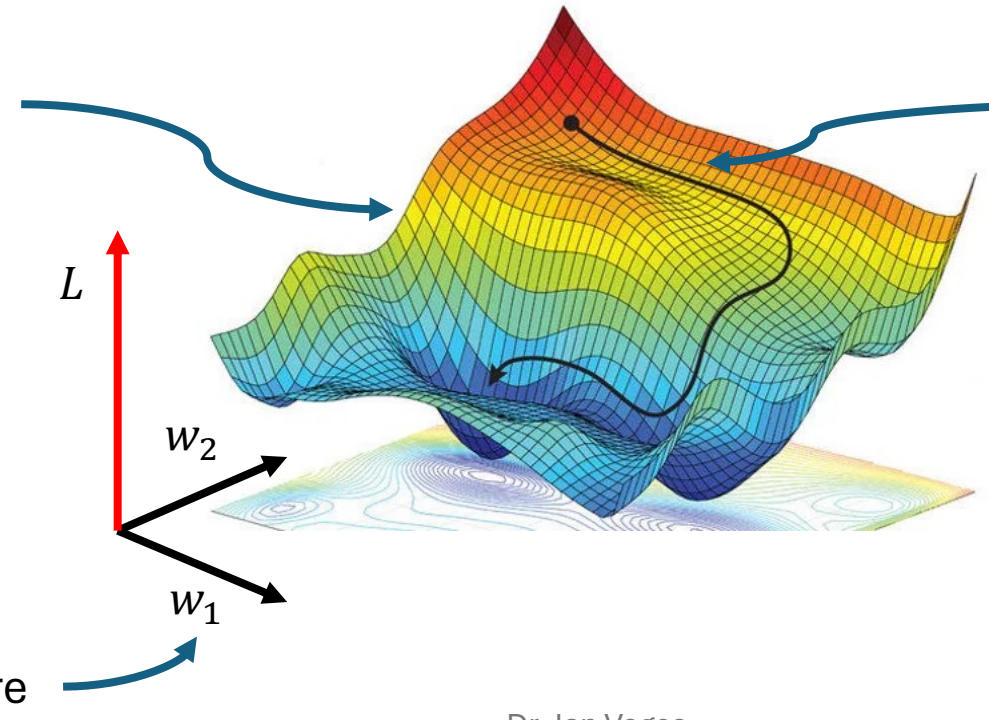
MSE loss over a set of examples $i = \{1, \dots, M\}$:

$$L = \frac{1}{M} \sum_i L^{(i)}(W)$$

Optimizing the loss function: gradient descent

Goal: find the "best" values of the neural network (i.e., model) parameters that minimize the loss function.

"Loss landscape": the value of the loss function at every values of the model parameters



Main idea: to iteratively update the model parameters, take steps in the local direction of steepest (negative) slope, i.e., the negative gradient

Review: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Example: $f(x) = x^2$ $\quad \frac{\partial f}{\partial x} = 2x$

The **gradient** of a function of multiple variables is the vector of partial derivatives of the function w.r.t. each variable.

Example: $f(x_1, x_2) = 3x_1^2 + x_2^2$ $\quad \nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = [6x_1 \quad 2x_2]^T$

Review: derivatives and gradients

$$f(x_1, x_2) = 3x_1^2 + x_2^2 \qquad \nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = [6x_1 \quad 2x_2]^T$$

The gradient evaluated at a particular point is the directions of steepest *ascent* of the function at that point:

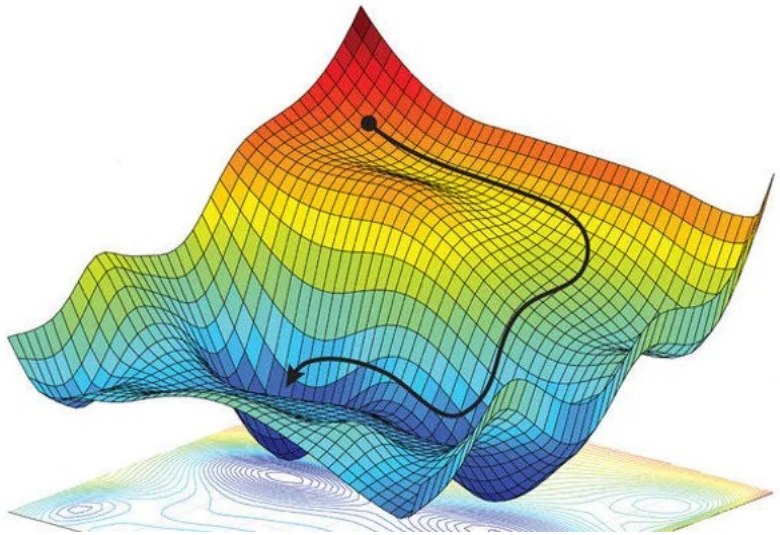
$$\nabla f|_{x_1=1, x_2=1} = [6 \quad 2]^T$$

Gradient descent algorithm

Let the gradient of the loss function w.r.t. the model parameters $\boldsymbol{w}$ be:

$$\nabla L = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K}\right]^T$$

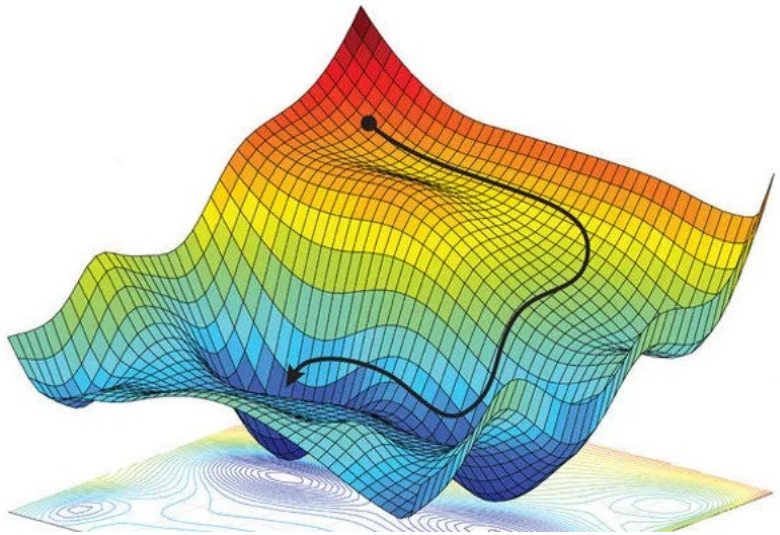
For ease of notation, we rewrite the bias parameter b as a w_k (corresponding to a $x_k = 1$).



Gradient descent algorithm

Then we can minimize the loss function by iteratively updating the model parameters ("taking steps") in the direction of the negative gradient, until convergence:

$$\boldsymbol{w} := \boldsymbol{w} - \eta \nabla L$$

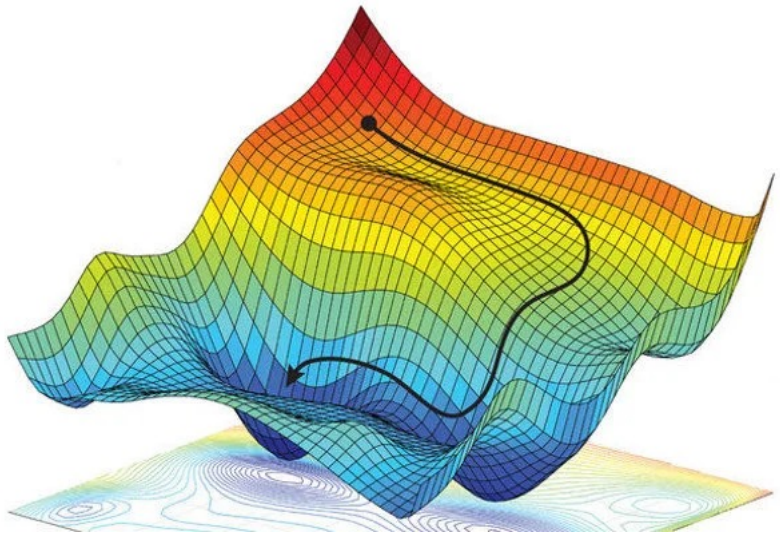


"Step size" hyperparameter (design choice, also referred to as **learning rate**), indicating how big of a step in the negative gradient direction we want to take at each update. Too big: may overshoot minima. Too small: optimization takes (too) long.

Gradient descent algorithm

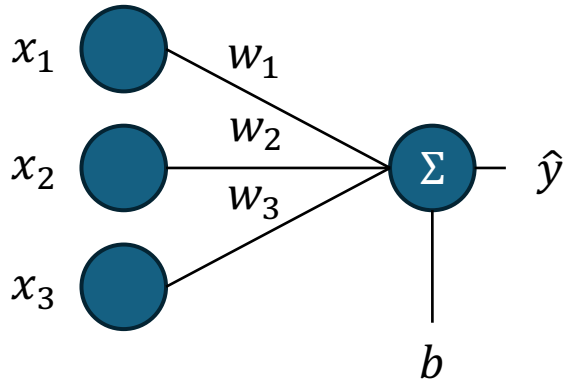
Evaluating the gradient involves iterating over all data examples, which can be slow!

In practice, we usually use stochastic gradient descent (or more sophisticated methods), where we estimate the gradient over a sample of data examples.



Optimizing the loss function: our example

Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b = \boldsymbol{w}^T \boldsymbol{x} + b$
Neural network parameters: $W = \{\boldsymbol{w}, b\} \widehat{=} \boldsymbol{w}$



Loss function

Per example $\boldsymbol{x}^{(i)}$: $L^{(i)}(\boldsymbol{w}) = \left(\hat{y}^{(i)} - y^{(i)}\right)^2$

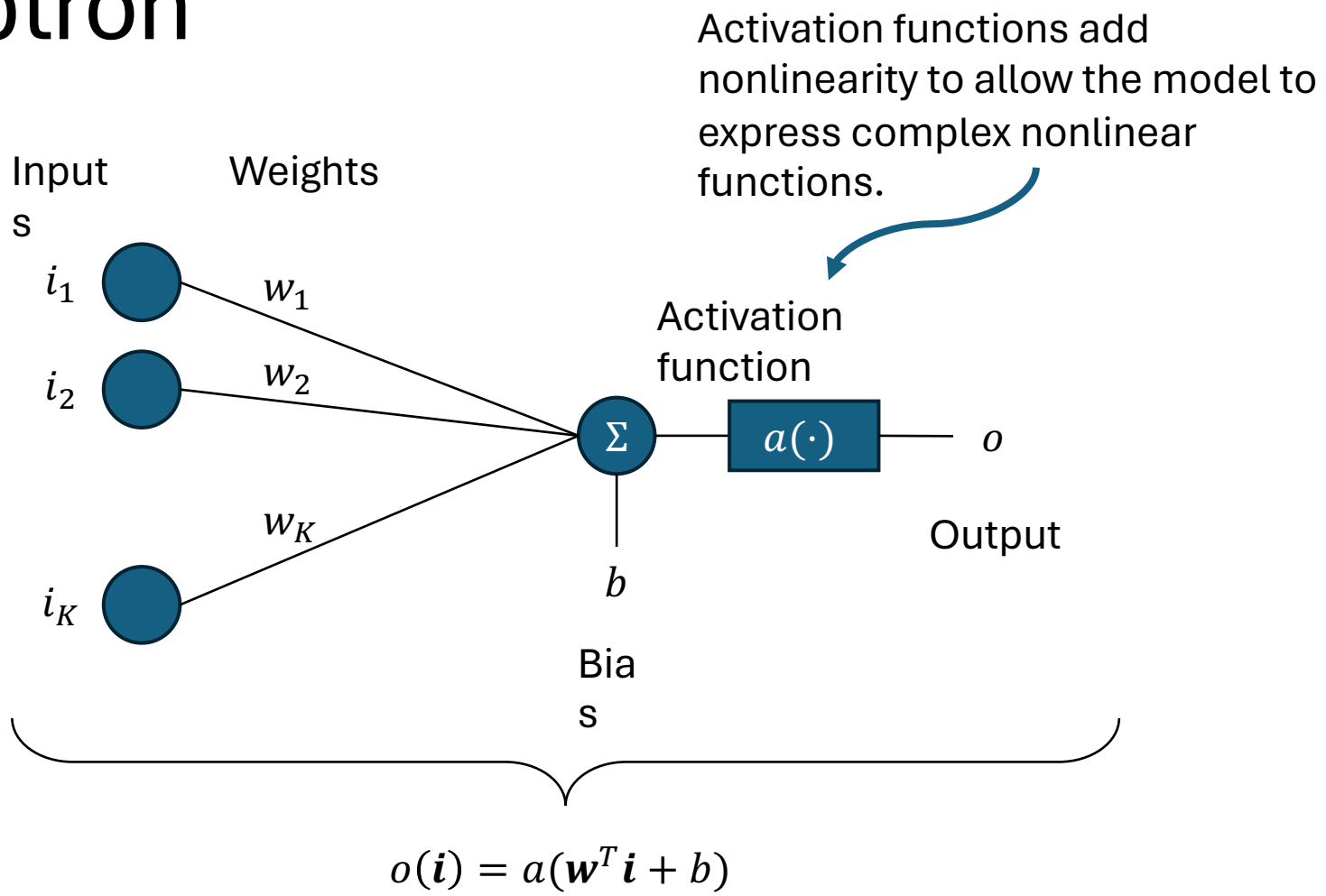
Over M examples: $L = \frac{1}{M}\sum_i L^{(i)}(\boldsymbol{w})$

Gradient of loss function w.r.t. the weights

Partial derivative w.r.t. the k-th weight: $\frac{\partial L^{(i)}}{\partial w_k} = \frac{\partial L^{(i)}}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial w_k} = 2 \cdot \left(\hat{y}^{(i)} - y^{(i)}\right) \cdot x_k^{(i)}$

Over M examples: $\frac{\partial L}{\partial w_k} = \frac{1}{M}\sum_i \frac{\partial L^{(i)}}{\partial w_k} = \frac{1}{M}\sum_i 2\left(\hat{y}^{(i)} - y^{(i)}\right) x_k^{(i)}$

The perceptron

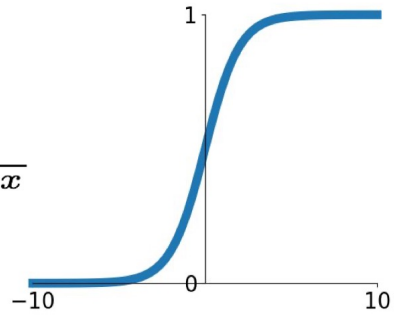


$$o(\boldsymbol{i}) = a(\boldsymbol{w}^T \boldsymbol{i} + b)$$

Activation functions

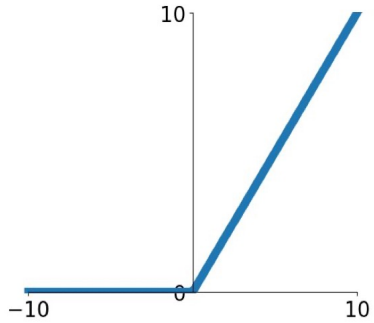
Sigmoid

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



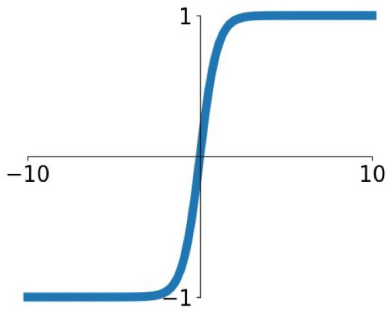
ReLU

$$\max(0, x)$$



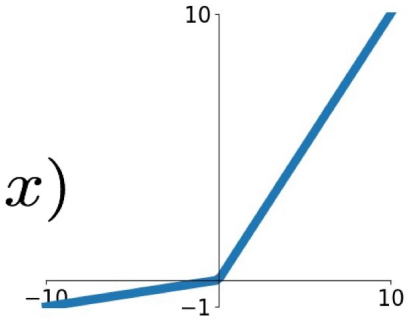
Tanh

$$\tanh(x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Activation functions

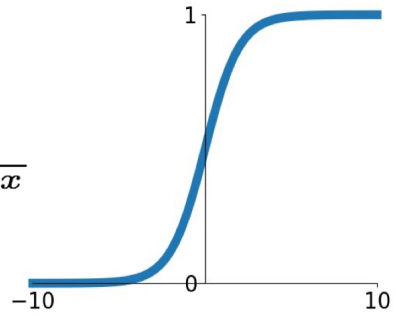
Problem: tiny gradients for large and small x ("vanishing gradient")

Piecewise linear, but nonlinear as a whole

Easily differentiable: $\text{sig}'(x) = \text{sig}(x)\left(1 - \text{sig}(x)\right)$

Sigmoid

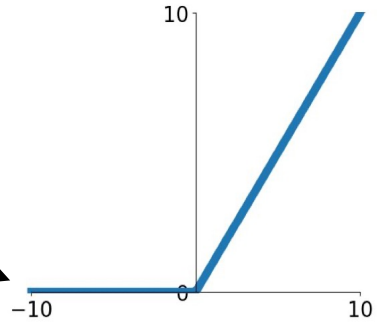
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



ReLU

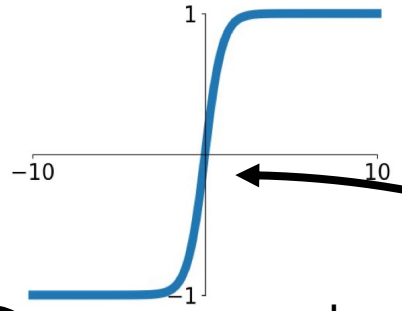
$$\max(0, x)$$

Output zero can lead to "dying" neurons.
→ Leaky ReLU



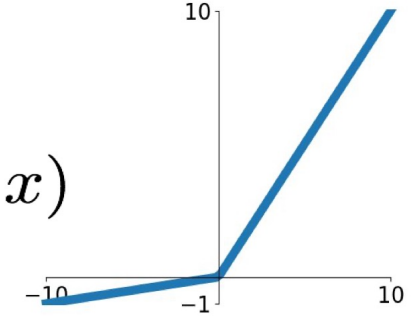
Tanh

$$\tanh(x)$$



Leaky ReLU

$$\max(0.1x, x)$$



Larger slope around $x = 0$

Scaled and shifted sigmoid function: $\tanh(x) = 2 \cdot \text{sig}(2x) - 1$

Computing gradients with backpropagation

Motivation

Deriving a full expression for multi-layer fully-connected neural networks (i.e., multi-layer perceptrons — MLPs) and other more complex architectures is (too) complex.

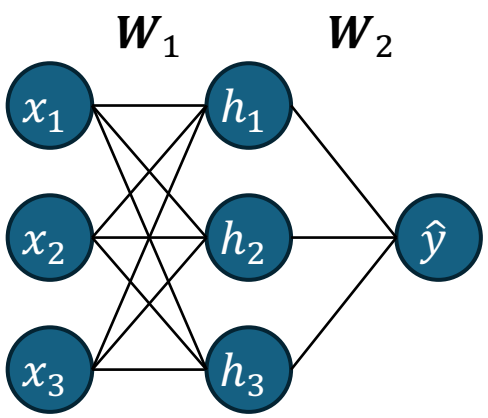
Key idea

Don't mathematically derive entire math expression for e.g., $\frac{\partial L}{\partial w_k}$.

Construct a **computational graph**.

Then, by writing $\frac{\partial L}{\partial w_k}$ as nested applications of the **chain rule**, only must derive simple "local" gradients representing relationships between connected nodes of the graph.

Backprop in a two-layer MLP

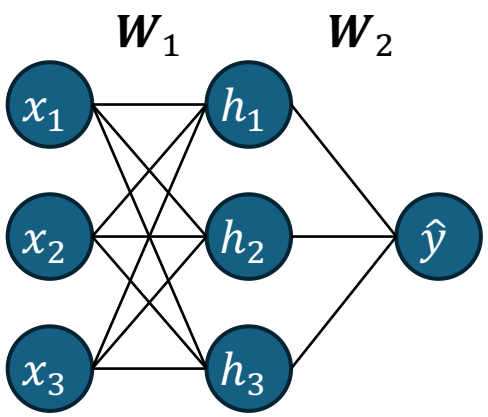


Network output: $\hat{y} = \boldsymbol{W}_2\big(\sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\big) + \boldsymbol{b}_2$

Loss: $L = (\hat{y} - y)^2$

We only consider a single training example; in the case of a minibatch, the gradients $\frac{\partial L}{\partial w_k}$ are averaged over all examples in the minibatch.

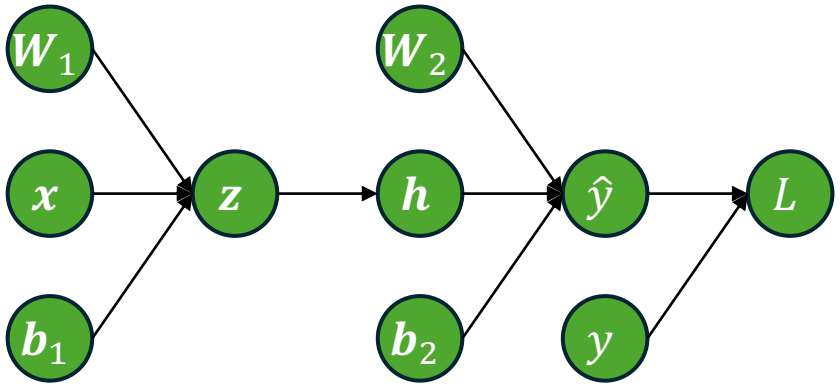
Backprop in a two-layer MLP



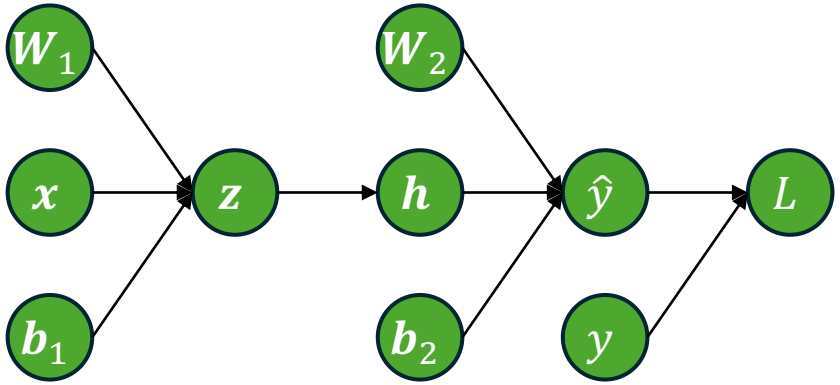
Network output: $\hat{y} = \boldsymbol{W}_2\big(\sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\big) + \boldsymbol{b}_2$

Loss: $L = (\hat{y} - y)^2$

We now think of computing the loss function as a staged computation of intermediate variables:



Backprop in a two-layer MLP



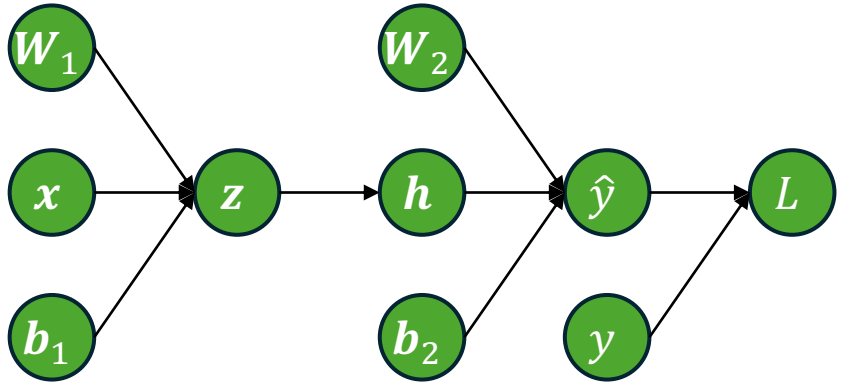
Network output: $\hat{y} = \boldsymbol{W}_2\big(\sigma(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)\big) + \boldsymbol{b}_2$

Loss: $L = (\hat{y} - y)^2$

Forward pass:

$$\boldsymbol{z} = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1$$
$$\boldsymbol{h} = \sigma(\boldsymbol{z})$$
$$\hat{y} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$$
$$L = (\hat{y} - y)^2$$

Backprop in a two-layer MLP



Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

The goal is to obtain all $\frac{\partial L}{\partial w_k}$.

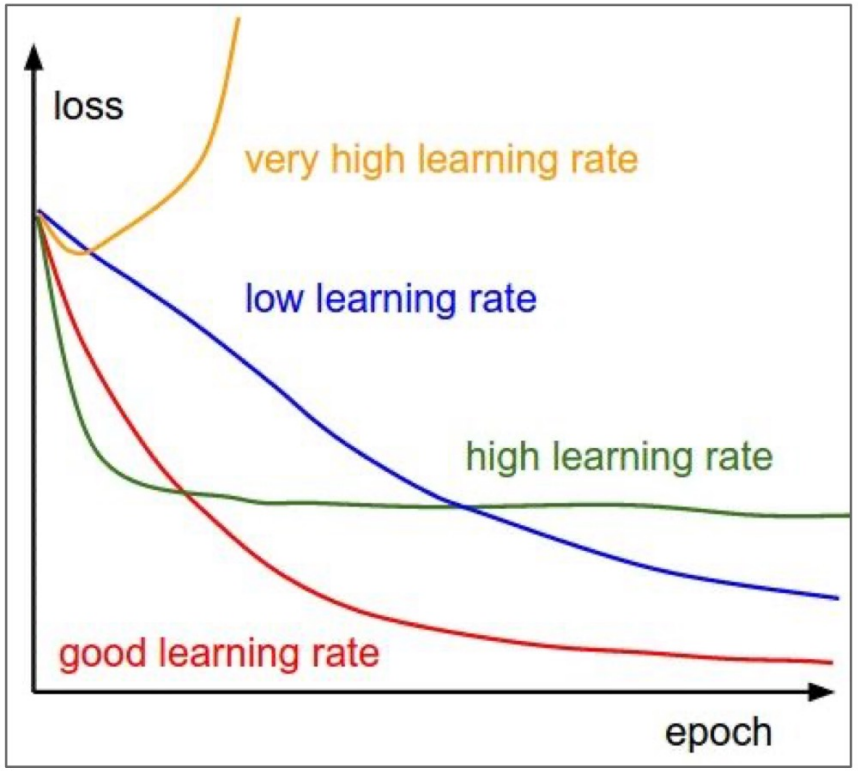
Backward pass:

Plug in from earlier computations via chain rule

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y), \qquad \frac{\partial L}{\partial \boldsymbol{W}_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{W}_2}, \qquad \frac{\partial L}{\partial \boldsymbol{h}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{h}}, \qquad \frac{\partial L}{\partial \boldsymbol{z}} = \frac{\partial L}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}, \qquad \frac{\partial L}{\partial \boldsymbol{W}_1} = \frac{\partial L}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}_1}$$

Local gradients to derive

Learning rates



Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time.

Empirical rule of thumb: If you increase the **batch size** by N, also scale the initial learning rate by N.

Batch size

Batch size vs. training data:

- In a typical ML dataset, you have many training examples, each consisting of input data (features) and corresponding target labels.
- Instead of feeding the entire dataset into the neural network at once, you divide it into smaller subsets or batches.

Batch size and training iterations:

- During each training iteration (or epoch), the model processes one batch of data. The batch size determines how many examples are included in this batch.
- After processing one batch, the model computes the gradient of the loss function with respect to its parameters (weights and biases) using the examples in that batch.

Impact on training:

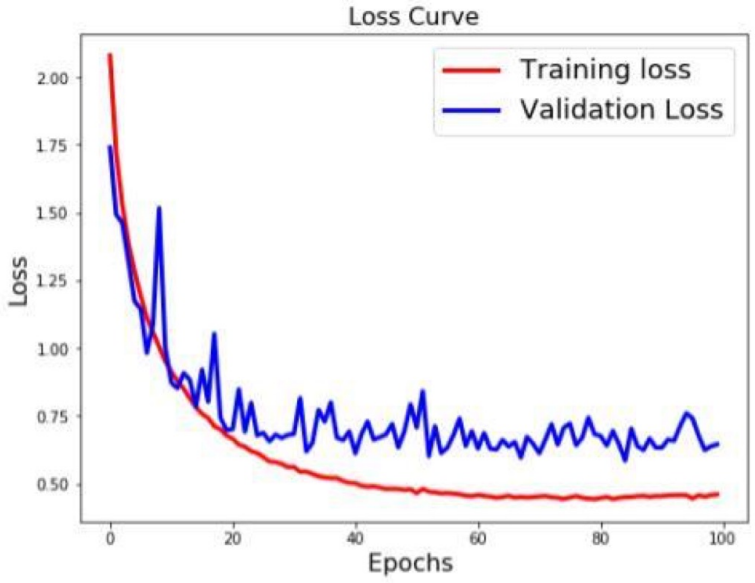
- Smaller batch size: more frequent model parameter updates. Can lead to noisy gradients but can help the model converge faster.
- Larger batch size: more stable gradient estimates. May slow down training convergence, especially on large datasets.

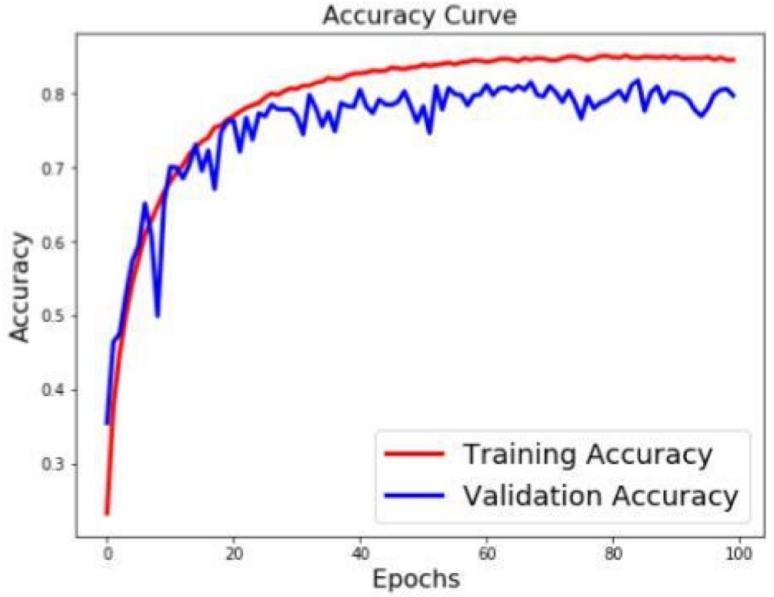
Trade-offs:

- Training speed vs. model stability.
- Also, larger batch sizes can be computationally efficient but might not fit in GPU memory for very large datasets.

Monitor learning curves

- Periodically evaluate validation loss
- Also, useful to plot performance on final metric



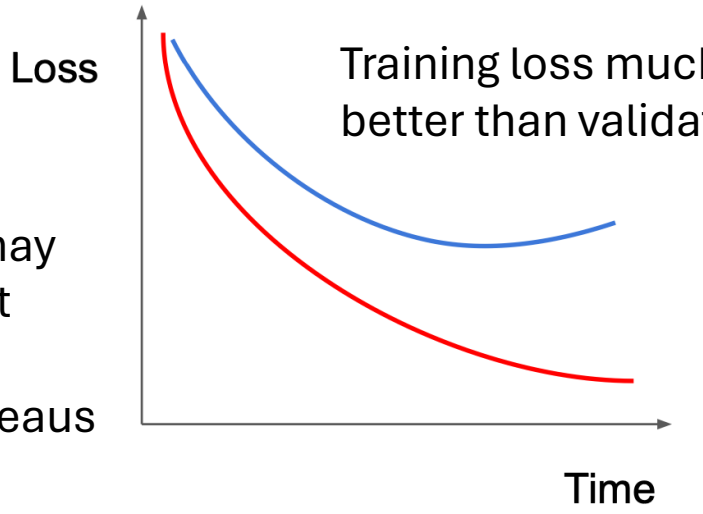


Training, validation, and test set

- Training set
  - Largest subset, typically around 70–80% of the entire dataset
- Validation set
  - Used to fine-tune the model's hyperparameters (e.g., learning rate, batch size, number of layers) and assess its performance during training
  - Used to regularly monitor the model's generalization performance
  - Typically, around 10–15% of the entire dataset
- Test set
  - Used to assess the final performance of the trained model
  - Serves as a proxy for how well the model is expected to perform on new, unseen data
  - Do *not* use during model development or hyperparameter tuning!

Overfitting vs. underfitting

Overfitting

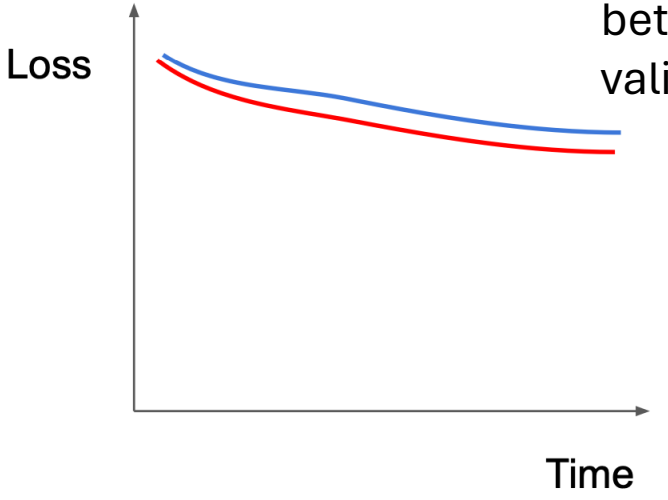


Training loss much better than validation

Training loss may continue to get better while validation plateaus or gets worse

Model is "overfitting" to the training data. Best strategy: increase data or regularize model. Second strategy: decrease model capacity (make simpler).

Underfitting

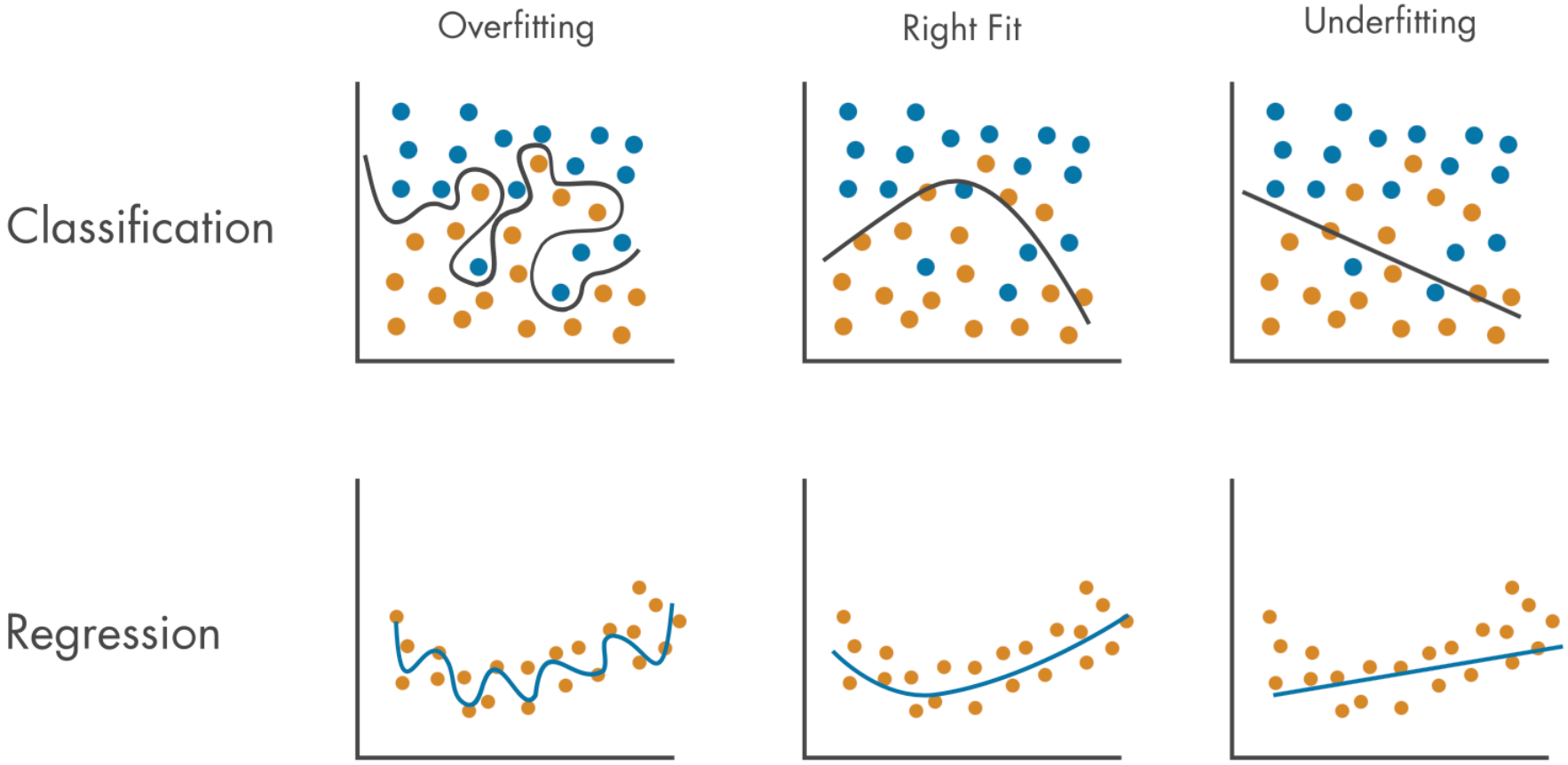


Small or no gap between training and validation loss

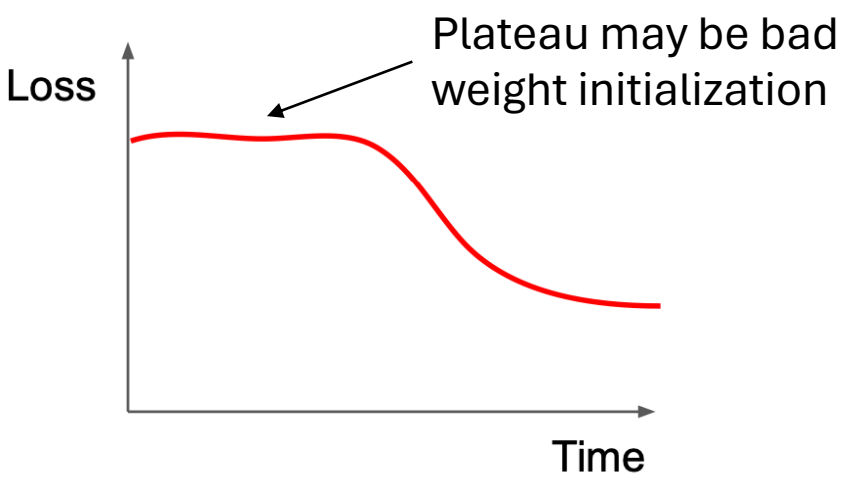
May have relatively higher loss overall (model not learning sufficiently)

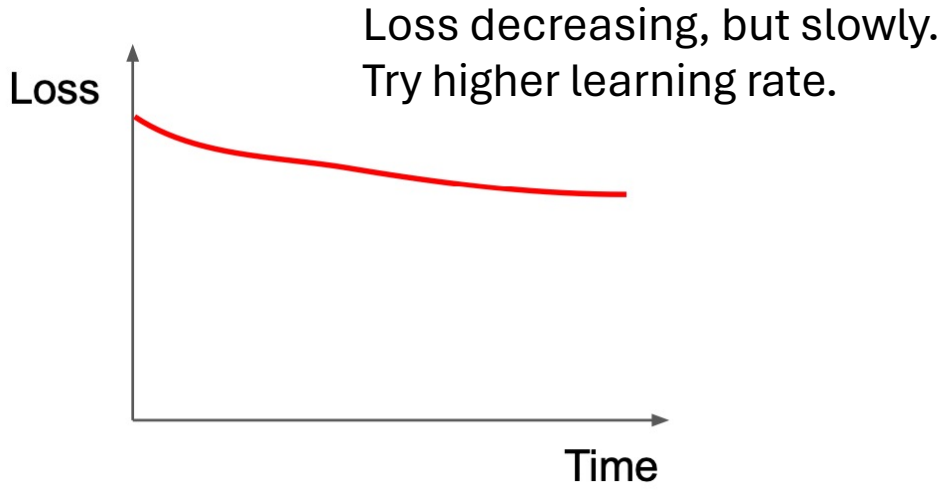
Model is not able to sufficiently learn to fit the data well. Best strategy: increase complexity (e.g., size) of the model. Second strategy: make problem simpler (easier task, cleaner data).

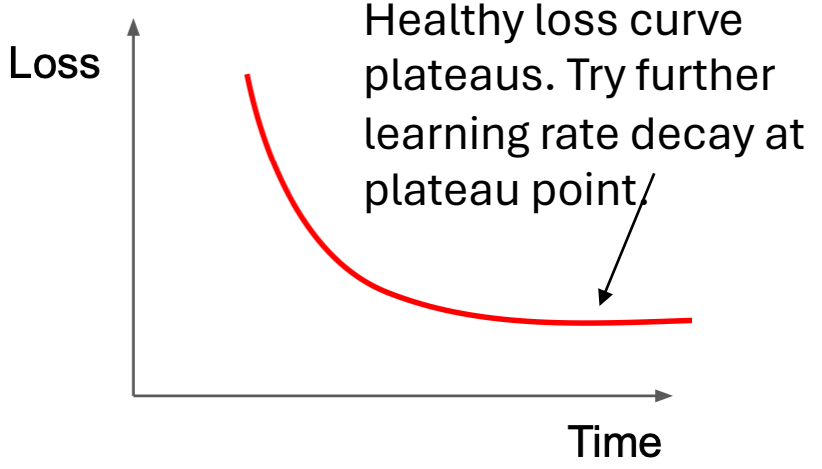
Over- vs. underfitting — more intuition

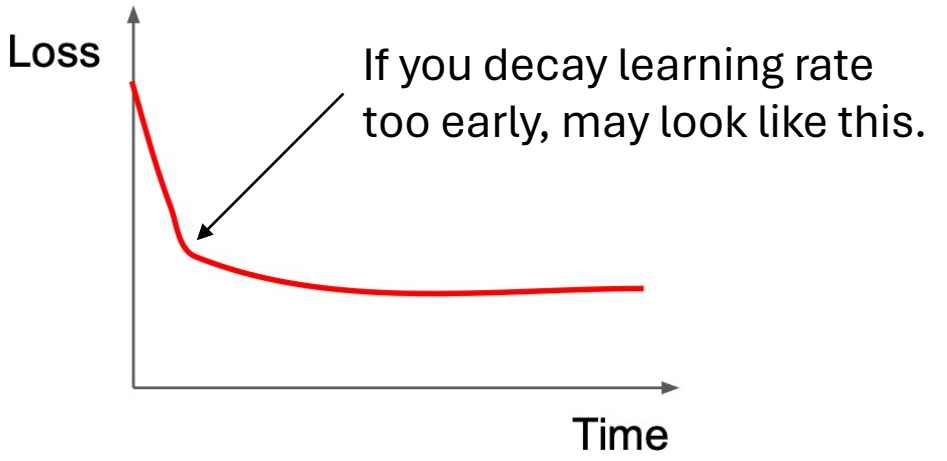


Debugging using learning curves









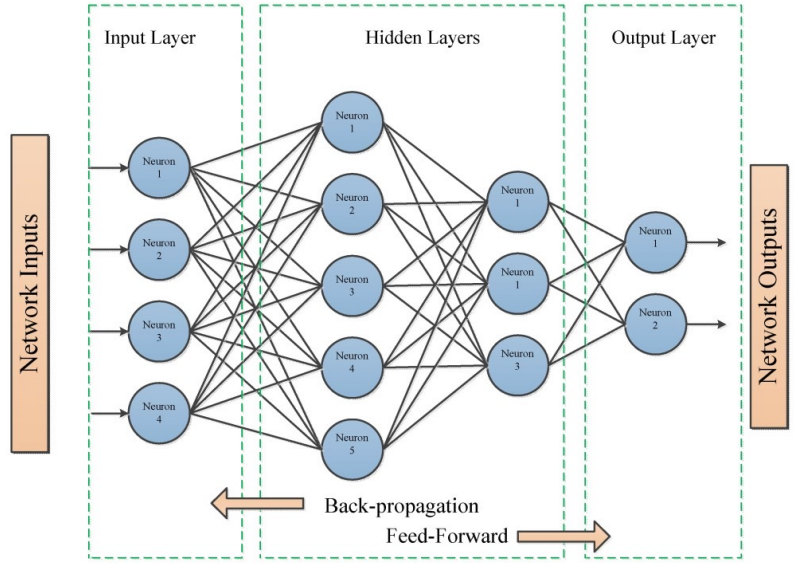
Major design choices

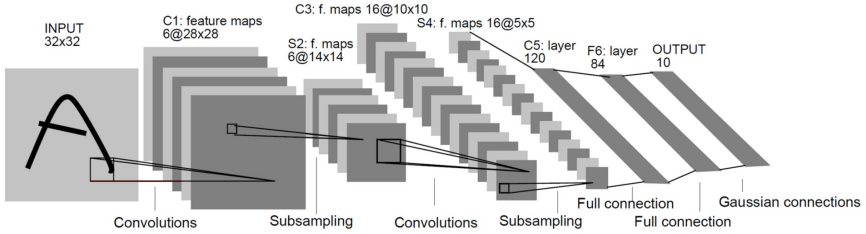
- Architecture type (FNN, CNN, RNN, autoencoder, …)
- Depth (i.e., no. of layers)
- For MLPs, no. of neurons in each layer (i.e., hidden layer size)
- Many more

→If trying to make a network bigger (when underfitting) or smaller (when overfitting), it is recommended to adjust network depth and hidden layer size first. Don’t waste too much time early on fiddling with choices that only minorly change the architecture.

Architecture types

- Feedforward neural network (**FNN**)
  - Basic neural network where information flows in one direction, from input to output layers
  - Composed of input, hidden, and output layers
  - Used for tasks like classification and regression
- Convolutional neural network (**CNN**)
  - Specialized for processing grid-like data, such as images and video
  - Employs convolutional layers to automatically learn and extract hierarchical features

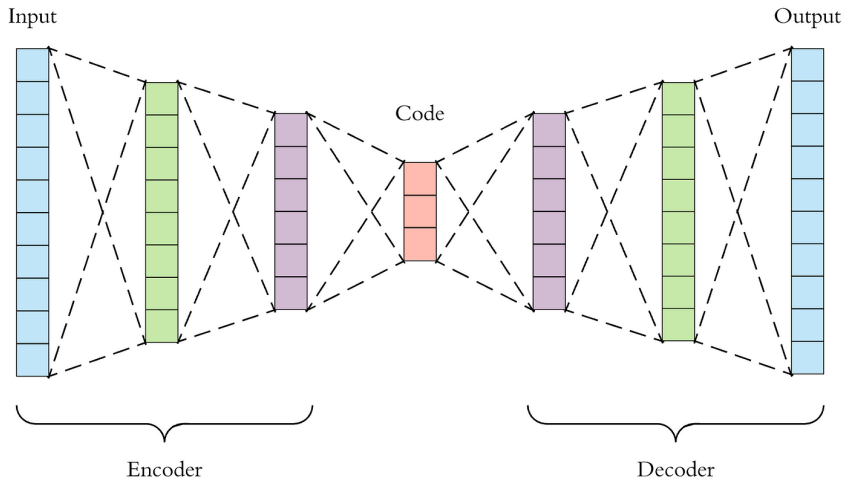




Architecture types

- Recurrent neural network (**RNN**)
  - Designed to handle sequential data, information flows in loops or recurrent connections
  - Suitable for tasks like natural language processing, speech recognition, and time series prediction
  - Has a hidden state that maintains memory of previous inputs
  - See also: LSTM, GRU
- **Autoencoder**
  - Neural network designed for unsupervised learning and feature extraction
  - Consists of an encoder to compress data and a decoder to reconstruct it
  - Used in dimensionality reduction, denoising, and anomaly detection
- Many more…

```python
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size

        # RNN layer
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)

        # Fully connected layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        # Forward pass through RNN layer
        out, hidden = self.rnn(x, hidden)

        # Reshape the output for the fully connected layer
        out = out.contiguous().view(-1, self.hidden_size)

        # Forward pass through fully connected layer
        out = self.fc(out)

        return out, hidden
```



Regularization

Remember optimizing loss function, which express how well a model fits the training data, e.g.:

$$L = \frac{1}{M}\sum_{i}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

Regularization adds a term to this, to express preferences on the weights (that prevent the model from fitting too well to the training data):

$$L = \frac{1}{M}\sum_{i}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 + \lambda R(W)$$

Regularization loss

Data loss

Importance of regularization term (a new hyperparameter)

Regularization

L1 regularization

$$R_{L1}(W) = \sum_k |w_k|$$

Encourages the model to reduce the magnitude of some weights to exactly zero, effectively performing feature selection. L1 regularization is useful when you suspect that only a subset of the input features is relevant for the task, as it can lead to sparse weight vectors.

L2 regularization

$$R_{L2}(W) = \sum_k w_k^2$$

Encourages the model to have smaller weight values across all features. L2 regularization helps to prevent extreme weight values and smoothens the optimization landscape, making it easier to train and reducing the risk of overfitting.

Elastic Net regularization

$$R_{EN}(W) = \alpha \cdot R_{L1}(W) + (1 - \alpha) \cdot R_{L2}(W)$$

Combines both L1 and L2 regularization by adding a penalty term that is a linear combination of the L1 and L2 penalties. It provides a balance between feature selection (L1) and weight smoothing (L2).

More design choices

- *Implicit* regularization
  - E.g., dropout: during training, at each iteration of forward pass randomly set some neurons to zero. Prevents the network from relying too heavily on any individual neuron, which encourages a more robust and generalizable model.
  - E.g., batch normalization: during training, at each iteration of forward pass normalize neuron activations by mean and variance of (mini-)batch. Allows keeping the weights in a healthy range.
- Data augmentation: augment effective training data size by simulating more diversity from existing data. Attention: augmentation must be realistic!
- Weight initialization
- Hyperparameter search

Neural network notebook

Notebook ‘pytorch-tutorial’

- Data generation
- Linear regression in NumPy
- Linear regression with PyTorch
  - Tensors
  - Computation of gradients
  - Optimizer
  - Loss
  - Model
  - Evaluation