

# Super VIP Cheatsheet: Deep Learning

Afshine AMIDI and Shervine AMIDI

November 25, 2018

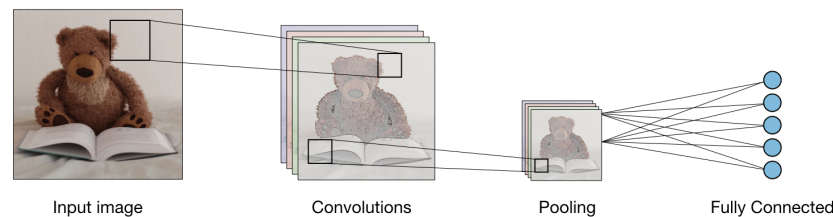
## Contents

- 1 Convolutional Neural Networks** **2**
  - 1.1 Overview 2
  - 1.2 Types of layer 2
  - 1.3 Filter hyperparameters 2
  - 1.4 Tuning hyperparameters 3
  - 1.5 Commonly used activation functions 3
  - 1.6 Object detection 4
    - 1.6.1 Face verification and recognition 5
    - 1.6.2 Neural style transfer 5
    - 1.6.3 Architectures using computational tricks 6
- 2 Recurrent Neural Networks** **7**
  - 2.1 Overview 7
  - 2.2 Handling long term dependencies 8
  - 2.3 Learning word representation 9
    - 2.3.1 Motivation and notations 9
    - 2.3.2 Word embeddings 9
  - 2.4 Comparing words 9
  - 2.5 Language model 10
  - 2.6 Machine translation 10
  - 2.7 Attention 10
- 3 Deep Learning Tips and Tricks** **11**
  - 3.1 Data processing 11
  - 3.2 Training a neural network 12
    - 3.2.1 Definitions 12
    - 3.2.2 Finding optimal weights 12
  - 3.3 Parameter tuning 12
    - 3.3.1 Weights initialization 12
    - 3.3.2 Optimizing convergence 12
  - 3.4 Regularization 13
  - 3.5 Good practices 13

## 1 Convolutional Neural Networks

### 1.1 Overview

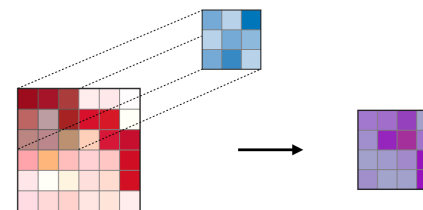
□ **Architecture of a traditional CNN** – Convolutional neural networks, also known as CNNs, are a specific type of neural networks that are generally composed of the following layers:



The convolution layer and the pooling layer can be fine-tuned with respect to hyperparameters that are described in the next sections.

### 1.2 Types of layer

□ **Convolutional layer (CONV)** – The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input  $I$  with respect to its dimensions. Its hyperparameters include the filter size  $F$  and stride  $S$ . The resulting output  $O$  is called *feature map* or *activation map*.

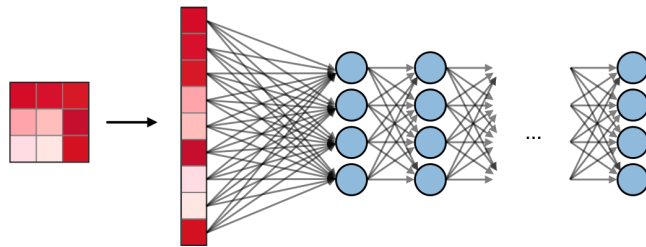


*Remark: the convolution step can be generalized to the 1D and 3D cases as well.*

□ **Pooling (POOL)** – The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.

	Max pooling	Average pooling
<b>Purpose</b>	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
<b>Illustration</b>		
<b>Comments</b>	- Preserves detected features - Most commonly used	- Downsamples feature map - Used in LeNet

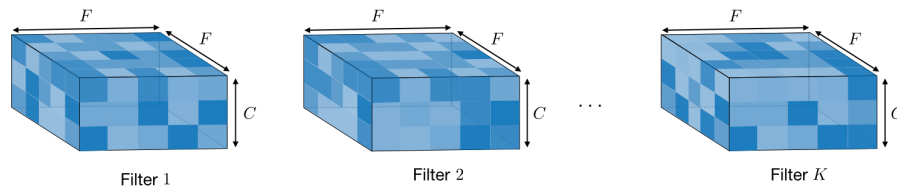
□ **Fully Connected (FC)** – The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



### 1.3 Filter hyperparameters

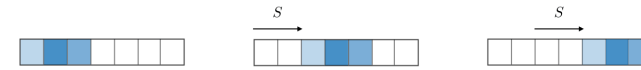
The convolution layer contains filters for which it is important to know the meaning behind its hyperparameters.

□ **Dimensions of a filter** – A filter of size  $F \times F$  applied to an input containing  $C$  channels is a  $F \times F \times C$  volume that performs convolutions on an input of size  $I \times I \times C$  and produces an output feature map (also called activation map) of size  $O \times O \times 1$ .



*Remark: the application of  $K$  filters of size  $F \times F$  results in an output feature map of size  $O \times O \times K$ .*

□ **Stride** – For a convolutional or a pooling operation, the stride  $S$  denotes the number of pixels by which the window moves after each operation.



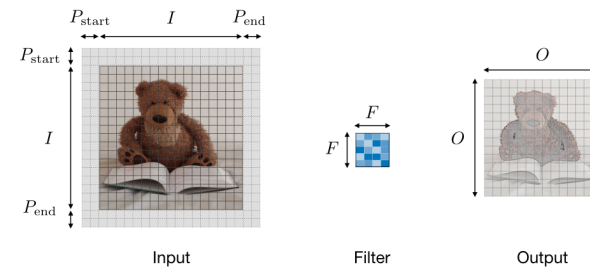
□ **Zero-padding** – Zero-padding denotes the process of adding  $P$  zeroes to each side of the boundaries of the input. This value can either be manually specified or automatically set through one of the three modes detailed below:

	Valid	Same	Full
<b>Value</b>	$P = 0$	$P_{start} = \left\lfloor \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rfloor$ $P_{end} = \left\lceil \frac{S \lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rceil$	$P_{start} \in \llbracket 0, F - 1 \rrbracket$ $P_{end} = F - 1$
<b>Illustration</b>			
<b>Purpose</b>	- No padding - Drops last convolution if dimensions do not match	- Padding such that feature map size has size $\lceil \frac{I}{S} \rceil$ - Output size is mathematically convenient - Also called 'half' padding	- Maximum padding such that end convolutions are applied on the limits of the input - Filter 'sees' the input end-to-end

### 1.4 Tuning hyperparameters

□ **Parameter compatibility in convolution layer** – By noting  $I$  the length of the input volume size,  $F$  the length of the filter,  $P$  the amount of zero padding,  $S$  the stride, then the output size  $O$  of the feature map along that dimension is given by:

$$O = \frac{I - F + P_{start} + P_{end}}{S} + 1$$



*Remark: often times,  $P_{start} = P_{end} \triangleq P$ , in which case we can replace  $P_{start} + P_{end}$  by  $2P$  in the formula above.*

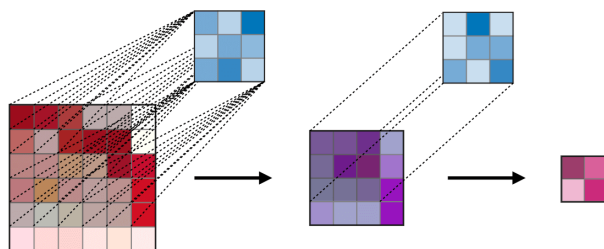
□ **Understanding the complexity of the model** – In order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

	CONV	POOL	FC
Illustration			
Input size	$I \times I \times C$	$I \times I \times C$	$N_{in}$
Output size	$O \times O \times K$	$O \times O \times C$	$N_{out}$
Number of parameters	$(F \times F \times C + 1) \cdot K$	0	$(N_{in} + 1) \times N_{out}$
Remarks	<ul style="list-style-type: none"> <li>- One bias parameter per filter</li> <li>- In most cases, <math>S &lt; F</math></li> <li>- A common choice for <math>K</math> is 2C</li> </ul>	<ul style="list-style-type: none"> <li>- Pooling operation done channel-wise</li> <li>- In most cases, <math>S = F</math></li> </ul>	<ul style="list-style-type: none"> <li>- Input is flattened</li> <li>- One bias parameter per neuron</li> <li>- The number of FC neurons is free of structural constraints</li> </ul>

□ **Receptive field** – The receptive field at layer  $k$  is the area denoted  $R_k \times R_k$  of the input that each pixel of the  $k$ -th activation map can 'see'. By calling  $F_j$  the filter size of layer  $j$  and  $S_i$  the stride value of layer  $i$  and with the convention  $S_0 = 1$ , the receptive field at layer  $k$  can be computed with the formula:

$$R_k = 1 + \sum_{j=1}^k (F_j - 1) \prod_{i=0}^{j-1} S_i$$

In the example below, we have  $F_1 = F_2 = 3$  and  $S_1 = S_2 = 1$ , which gives  $R_2 = 1 + 2 \cdot 1 + 2 \cdot 1 = 5$ .



### 1.5 Commonly used activation functions

□ **Rectified Linear Unit** – The rectified linear unit layer (ReLU) is an activation function  $g$  that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

ReLU	Leaky ReLU	ELU
$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
Non-linearity complexities biologically interpretable	Addresses dying ReLU issue for negative values	Differentiable everywhere

□ **Softmax** – The softmax step can be seen as a generalized logistic function that takes as input a vector of scores  $x \in \mathbb{R}^n$  and outputs a vector of output probability  $p \in \mathbb{R}^n$  through a softmax function at the end of the architecture. It is defined as follows:

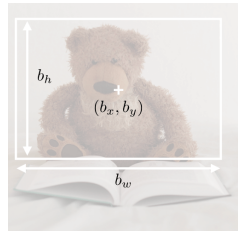
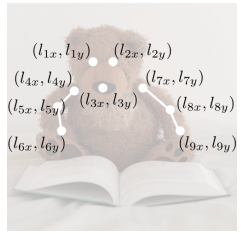
$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

### 1.6 Object detection

□ **Types of models** – There are 3 main types of object recognition algorithms, for which the nature of what is predicted is different. They are described in the table below:

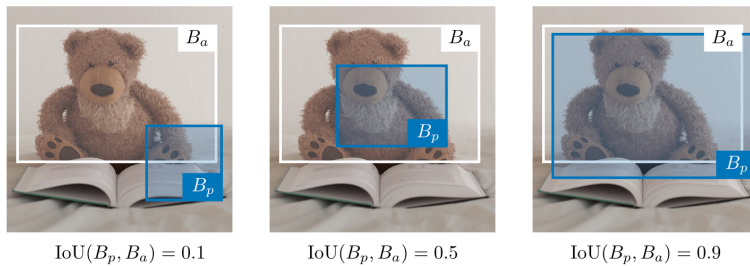
Image classification	Classification w. localization	Detection
<ul style="list-style-type: none"> <li>- Classifies a picture</li> <li>- Predicts probability of object</li> </ul>	<ul style="list-style-type: none"> <li>- Detects object in a picture</li> <li>- Predicts probability of object and where it is located</li> </ul>	<ul style="list-style-type: none"> <li>- Detects up to several objects in a picture</li> <li>- Predicts probabilities of objects and where they are located</li> </ul>
Traditional CNN	Simplified YOLO, R-CNN	YOLO, R-CNN

□ **Detection** – In the context of object detection, different methods are used depending on whether we just want to locate the object or detect a more complex shape in the image. The two main ones are summed up in the table below:

Bounding box detection	Landmark detection
Detects the part of the image where the object is located	- Detects a shape or characteristics of an object (e.g. eyes) - More granular
	
Box of center $(b_x, b_y)$ , height $b_h$ and width $b_w$	Reference points $(l_{1x}, l_{1y}), \dots, (l_{nx}, l_{ny})$

□ **Intersection over Union** – Intersection over Union, also known as IoU, is a function that quantifies how correctly positioned a predicted bounding box  $B_p$  is over the actual bounding box  $B_a$ . It is defined as:

$$\text{IoU}(B_p, B_a) = \frac{B_p \cap B_a}{B_p \cup B_a}$$

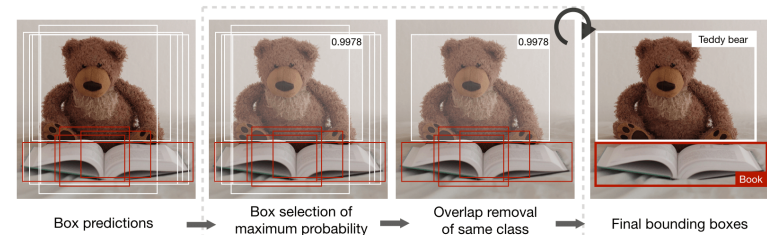


*Remark: we always have  $\text{IoU} \in [0, 1]$ . By convention, a predicted bounding box  $B_p$  is considered as being reasonably good if  $\text{IoU}(B_p, B_a) \geq 0.5$ .*

□ **Anchor boxes** – Anchor boxing is a technique used to predict overlapping bounding boxes. In practice, the network is allowed to predict more than one box simultaneously, where each box prediction is constrained to have a given set of geometrical properties. For instance, the first prediction can potentially be a rectangular box of a given form, while the second will be another rectangular box of a different geometrical form.

□ **Non-max suppression** – The non-max suppression technique aims at removing duplicate overlapping bounding boxes of a same object by selecting the most representative ones. After having removed all boxes having a probability prediction lower than 0.6, the following steps are repeated while there are boxes remaining:

- Step 1: Pick the box with the largest prediction probability.
- Step 2: Discard any box having an  $\text{IoU} \geq 0.5$  with the previous box.



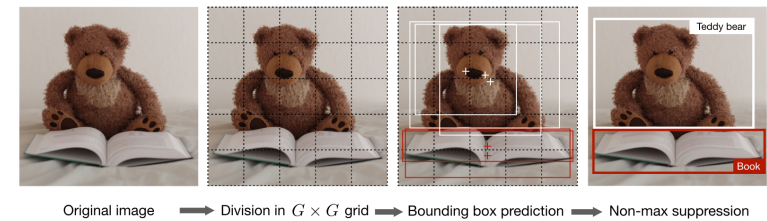
□ **YOLO** – You Only Look Once (YOLO) is an object detection algorithm that performs the following steps:

- Step 1: Divide the input image into a  $G \times G$  grid.
- Step 2: For each grid cell, run a CNN that predicts  $y$  of the following form:

$$y = \left[ \underbrace{p_c, b_x, b_y, b_h, b_w, c_1, c_2, \dots, c_p, \dots}_{\text{repeated } k \text{ times}} \right]^T \in \mathbb{R}^{G \times G \times k \times (5+p)}$$

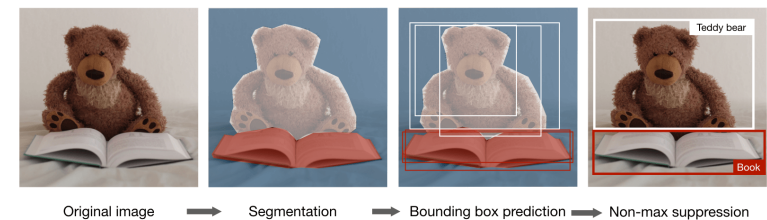
where  $p_c$  is the probability of detecting an object,  $b_x, b_y, b_h, b_w$  are the properties of the detected bounding box,  $c_1, \dots, c_p$  is a one-hot representation of which of the  $p$  classes were detected, and  $k$  is the number of anchor boxes.

- Step 3: Run the non-max suppression algorithm to remove any potential duplicate overlapping bounding boxes.



*Remark: when  $p_c = 0$ , then the network does not detect any object. In that case, the corresponding predictions  $b_x, \dots, c_p$  have to be ignored.*

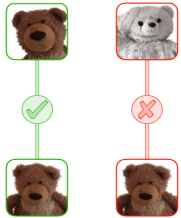
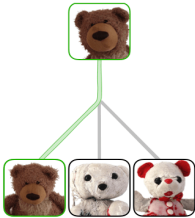
□ **R-CNN** – Region with Convolutional Neural Networks (R-CNN) is an object detection algorithm that first segments the image to find potential relevant bounding boxes and then run the detection algorithm to find most probable objects in those bounding boxes.



*Remark: although the original algorithm is computationally expensive and slow, newer architectures enabled the algorithm to run faster, such as Fast R-CNN and Faster R-CNN.*

### 1.6.1 Face verification and recognition

□ **Types of models** – Two main types of model are summed up in table below:

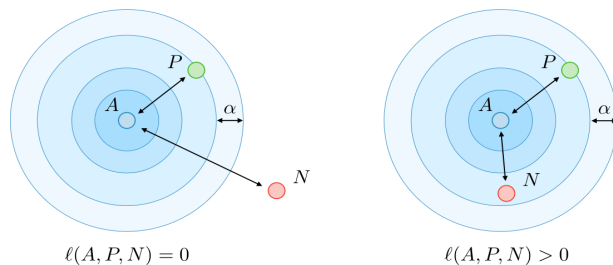
Face verification	Face recognition
- Is this the correct person? - One-to-one lookup	- Is this one of the $K$ persons in the database? - One-to-many lookup
<p>Query</p>  <p>Reference</p>	<p>Query</p>  <p>Database</p>

□ **One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted  $d(\text{image 1}, \text{image 2})$ .

□ **Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. For a given input image  $x^{(i)}$ , the encoded output is often noted as  $f(x^{(i)})$ .

□ **Triplet loss** – The triplet loss  $\ell$  is a loss function computed on the embedding representation of a triplet of images  $A$  (anchor),  $P$  (positive) and  $N$  (negative). The anchor and the positive example belong to a same class, while the negative example to another one. By calling  $\alpha \in \mathbb{R}^+$  the margin parameter, this loss is defined as follows:

$$\ell(A, P, N) = \max(d(A, P) - d(A, N) + \alpha, 0)$$



### 1.6.2 Neural style transfer

□ **Motivation** – The goal of neural style transfer is to generate an image  $G$  based on a given content  $C$  and a given style  $S$ .



□ **Activation** – In a given layer  $l$ , the activation is noted  $a^{[l]}$  and is of dimensions  $n_H \times n_w \times n_c$

□ **Content cost function** – The content cost function  $J_{\text{content}}(C, G)$  is used to determine how the generated image  $G$  differs from the original content image  $C$ . It is defined as follows:

$$J_{\text{content}}(C, G) = \frac{1}{2} \|a^{[l](C)} - a^{[l](G)}\|^2$$

□ **Style matrix** – The style matrix  $G^{[l]}$  of a given layer  $l$  is a Gram matrix where each of its elements  $G_{kk'}^{[l]}$  quantifies how correlated the channels  $k$  and  $k'$  are. It is defined with respect to activations  $a^{[l]}$  as follows:

$$G_{kk'}^{[l]} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_w} a_{ijk}^{[l]} a_{ijk'}^{[l]}$$

*Remark: the style matrix for the style image and the generated image are noted  $G^{[l](S)}$  and  $G^{[l](G)}$  respectively.*

□ **Style cost function** – The style cost function  $J_{\text{style}}(S, G)$  is used to determine how the generated image  $G$  differs from the style  $S$ . It is defined as follows:

$$J_{\text{style}}^{[l]}(S, G) = \frac{1}{(2n_H n_w n_c)^2} \|G^{[l](S)} - G^{[l](G)}\|_F^2 = \frac{1}{(2n_H n_w n_c)^2} \sum_{k, k'=1}^{n_c} \left(G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)}\right)^2$$

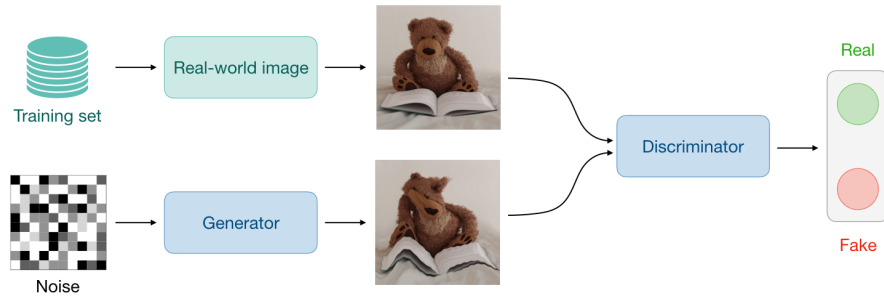
□ **Overall cost function** – The overall cost function is defined as being a combination of the content and style cost functions, weighted by parameters  $\alpha, \beta$ , as follows:

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$$

*Remark: a higher value of  $\alpha$  will make the model care more about the content while a higher value of  $\beta$  will make it care more about the style.*

### 1.6.3 Architectures using computational tricks

□ **Generative Adversarial Network** – Generative adversarial networks, also known as GANs, are composed of a generative and a discriminative model, where the generative model aims at generating the most truthful output that will be fed into the discriminative which aims at differentiating the generated and true image.



Remark: use cases using variants of GANs include text to image, music generation and synthesis.

□ **ResNet** – The Residual Network architecture (also called ResNet) uses residual blocks with a high number of layers meant to decrease the training error. The residual block has the following characterizing equation:

$$a^{[l+2]} = g(a^{[l]} + z^{[l+2]})$$

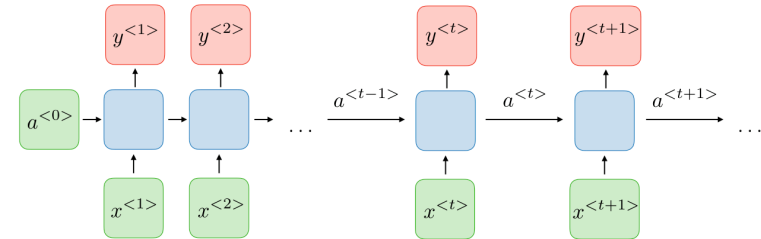
□ **Inception Network** – This architecture uses inception modules and aims at giving a try at different convolutions in order to increase its performance. In particular, it uses the  $1 \times 1$  convolution trick to lower the burden of computation.

\* \* \*

## 2 Recurrent Neural Networks

### 2.1 Overview

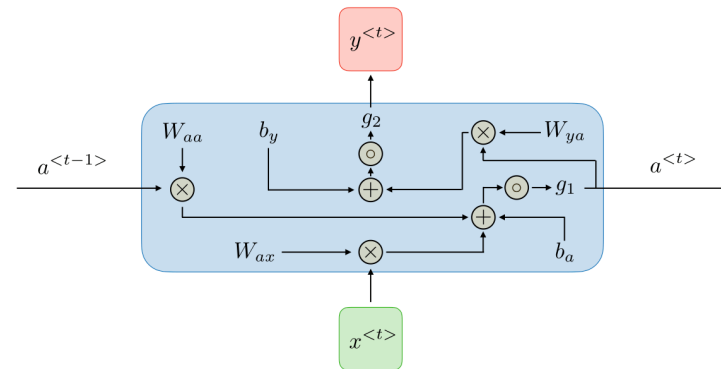
□ **Architecture of a traditional RNN** – Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



For each timestep  $t$ , the activation  $a^{<t>}$  and the output  $y^{<t>}$  are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where  $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$  are coefficients that are shared temporally and  $g_1, g_2$  activation functions



The pros and cons of a typical RNN architecture are summed up in the table below:

Advantages	Drawbacks
<ul style="list-style-type: none"> <li>- Possibility of processing input of any length</li> <li>- Model size not increasing with size of input</li> <li>- Computation takes into account historical information</li> <li>- Weights are shared across time</li> </ul>	<ul style="list-style-type: none"> <li>- Computation being slow</li> <li>- Difficulty of accessing information from a long time ago</li> <li>- Cannot consider any future input for the current state</li> </ul>

□ **Applications of RNNs** – RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

□ **Loss function** – In the case of a recurrent neural network, the loss function  $\mathcal{L}$  of all time steps is defined based on the loss at every time step as follows:

$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}(\hat{y}^{<t>}, y^{<t>})$$

□ **Backpropagation through time** – Backpropagation is done at each point in time. At timestep  $T$ , the derivative of the loss  $\mathcal{L}$  with respect to weight matrix  $W$  is expressed as follows:

$$\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^T \frac{\partial \mathcal{L}^{(T)}}{\partial W} \Big|_{(t)}$$

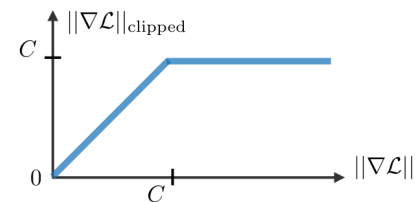
## 2.2 Handling long term dependencies

□ **Commonly used activation functions** – The most common activation functions used in RNN modules are described below:

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$

□ **Vanishing/exploding gradient** – The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

□ **Gradient clipping** – It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.



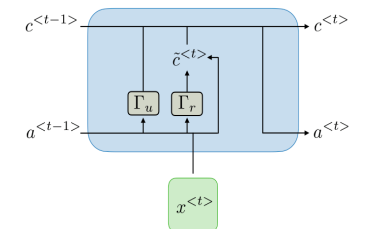
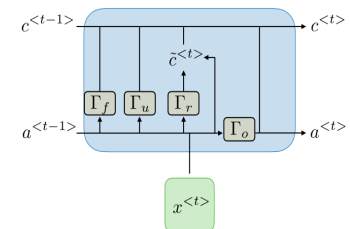
□ **Types of gates** – In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted  $\Gamma$  and are equal to:

$$\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)$$

where  $W, U, b$  are coefficients specific to the gate and  $\sigma$  is the sigmoid function. The main ones are summed up in the table below:

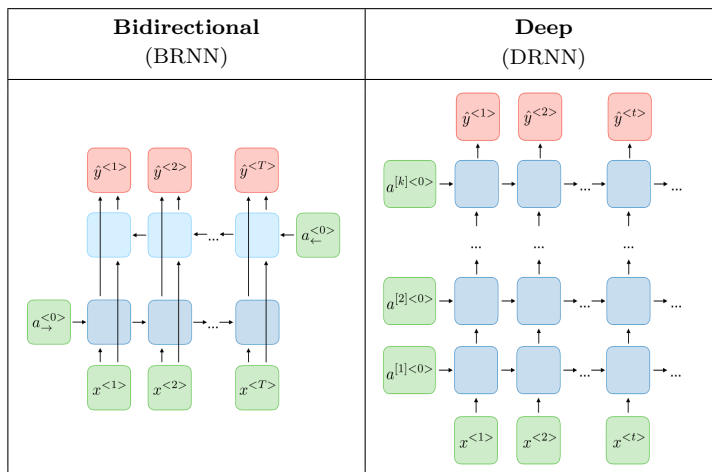
Type of gate	Role	Used in
Update gate $\Gamma_u$	How much past should matter now?	GRU, LSTM
Relevance gate $\Gamma_r$	Drop previous information?	GRU, LSTM
Forget gate $\Gamma_f$	Erase a cell or not?	LSTM
Output gate $\Gamma_o$	How much to reveal of a cell?	LSTM

□ **GRU/LSTM** – Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

	Gated Recurrent Unit (GRU)	Long Short-Term Memory (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \star \tilde{c}^{<t>} + (1 - \Gamma_u) \star c^{<t-1>}$	$\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \star c^{<t>}$
Dependencies		

Remark: the sign  $\star$  denotes the element-wise multiplication between two vectors.

□ **Variants of RNNs** – The table below sums up the other commonly used RNN architectures:

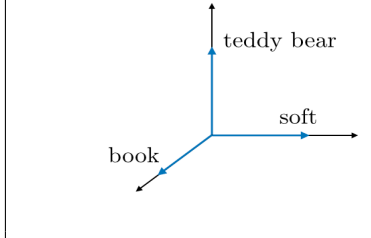
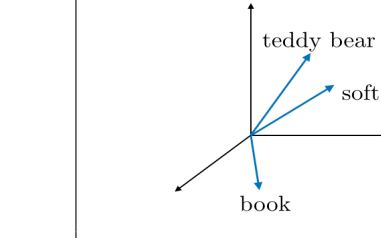


## 2.3 Learning word representation

In this section, we note  $V$  the vocabulary and  $|V|$  its size.

### 2.3.1 Motivation and notations

□ **Representation techniques** – The two main ways of representing words are summed up in the table below:

1-hot representation	Word embedding
	
<ul style="list-style-type: none"> <li>- Noted <math>o_w</math></li> <li>- Naive approach, no similarity information</li> </ul>	<ul style="list-style-type: none"> <li>- Noted <math>e_w</math></li> <li>- Takes into account words similarity</li> </ul>

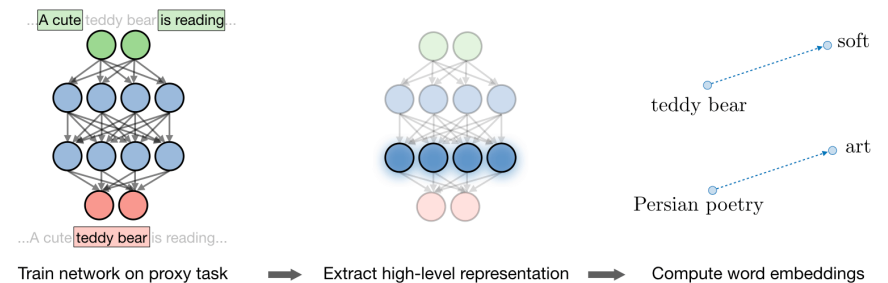
□ **Embedding matrix** – For a given word  $w$ , the embedding matrix  $E$  is a matrix that maps its 1-hot representation  $o_w$  to its embedding  $e_w$  as follows:

$$e_w = E o_w$$

Remark: learning the embedding matrix can be done using target/context likelihood models.

### 2.3.2 Word embeddings

□ **Word2vec** – Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW.



□ **Skip-gram** – The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word  $t$  happening with a context word  $c$ . By noting  $\theta_t$  a parameter associated with  $t$ , the probability  $P(t|c)$  is given by:



$$P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)}$$

*Remark: summing over the whole vocabulary in the denominator of the softmax part makes this model computationally expensive. CBOW is another word2vec model using the surrounding words to predict a given word.*

□ **Negative sampling** – It is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of  $k$  negative examples and 1 positive example. Given a context word  $c$  and a target word  $t$ , the prediction is expressed by:

$$P(y = 1|c,t) = \sigma(\theta_t^T e_c)$$

*Remark: this method is less computationally expensive than the skip-gram model.*

□ **GloVe** – The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurrence matrix  $X$  where each  $X_{i,j}$  denotes the number of times that a target  $i$  occurred with a context  $j$ . Its cost function  $J$  is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij})(\theta_i^T e_j + b_i + b'_j - \log(X_{ij}))^2$$

here  $f$  is a weighting function such that  $X_{i,j} = 0 \implies f(X_{i,j}) = 0$ .

Given the symmetry that  $e$  and  $\theta$  play in this model, the final word embedding  $e_w^{(\text{final})}$  is given by:

$$e_w^{(\text{final})} = \frac{e_w + \theta_w}{2}$$

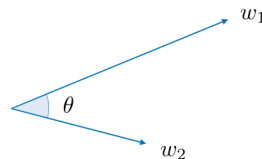
*Remark: the individual components of the learned word embeddings are not necessarily interpretable.*

## 2.4 Comparing words

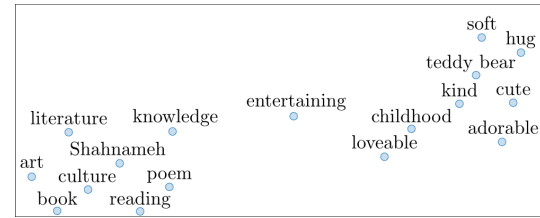
□ **Cosine similarity** – The cosine similarity between words  $w_1$  and  $w_2$  is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{\|w_1\| \|w_2\|} = \cos(\theta)$$

*Remark:  $\theta$  is the angle between words  $w_1$  and  $w_2$ .*



□  **$t$ -SNE** –  $t$ -SNE ( $t$ -distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.



## 2.5 Language model

□ **Overview** – A language model aims at estimating the probability of a sentence  $P(y)$ .

□  **$n$ -gram model** – This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.

□ **Perplexity** – Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words  $T$ . The perplexity is such that the lower, the better and is defined as follows:

$$PP = \prod_{t=1}^T \left( \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)^{\frac{1}{T}}$$

*Remark: PP is commonly used in  $t$ -SNE.*

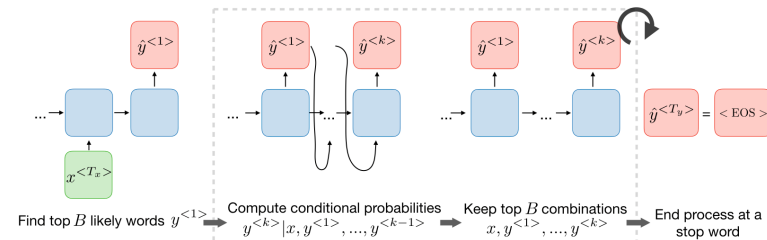
## 2.6 Machine translation

□ **Overview** – A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model. The goal is to find a sentence  $y$  such that:

$$y = \arg \max_{y^{<1>, \dots, y^{<T_y>}} P(y^{<1>, \dots, y^{<T_y>} | x)$$

□ **Beam search** – It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence  $y$  given an input  $x$ .

- Step 1: Find top  $B$  likely words  $y^{<1>}$
- Step 2: Compute conditional probabilities  $y^{<k>} | x, y^{<1>, \dots, y^{<k-1>}$
- Step 3: Keep top  $B$  combinations  $x, y^{<1>, \dots, y^{<k>}$



*Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.*

□ **Beam width** – The beam width  $B$  is a parameter for beam search. Large values of  $B$  yield to better result but with slower performance and increased memory. Small values of  $B$  lead to worse results but is less computationally intensive. A standard value for  $B$  is around 10.

□ **Length normalization** – In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\text{Objective} = \frac{1}{T_y^\alpha} \sum_{t=1}^{T_y} \log \left[ p(y^{<t>} | x, y^{<1>}, \dots, y^{<t-1>}) \right]$$

*Remark: the parameter  $\alpha$  can be seen as a softener, and its value is usually between 0.5 and 1.*

□ **Error analysis** – When obtaining a predicted translation  $\hat{y}$  that is bad, one can wonder why we did not get a good translation  $y^*$  by performing the following error analysis:

Case	$P(y^* x) > P(\hat{y} x)$	$P(y^* x) \leq P(\hat{y} x)$
<b>Root cause</b>	Beam search faulty	RNN faulty
<b>Remedies</b>	Increase beam width	- Try different architecture - Regularize - Get more data

□ **Bleu score** – The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on  $n$ -gram precision. It is defined as follows:

$$\text{bleu score} = \exp \left( \frac{1}{n} \sum_{k=1}^n p_k \right)$$

where  $p_n$  is the bleu score on  $n$ -gram only defined as follows:

$$p_n = \frac{\sum_{\text{n-gram} \in \hat{y}} \text{count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \hat{y}} \text{count}(\text{n-gram})}$$

*Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.*

## 2.7 Attention

□ **Attention model** – This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting  $\alpha^{<t,t'>}$  the amount of attention that the output  $y^{<t>}$  should pay to the activation  $a^{<t'>}$  and  $c^{<t>}$  the context at time  $t$ , we have:

$$c^{<t>} = \sum_{t'} \alpha^{<t,t'>} a^{<t'>} \quad \text{with} \quad \sum_{t'} \alpha^{<t,t'>} = 1$$

*Remark: the attention scores are commonly used in image captioning and machine translation.*



*A cute teddy bear is reading Persian literature*



*A cute teddy bear is reading Persian literature*

□ **Attention weight** – The amount of attention that the output  $y^{<t>}$  should pay to the activation  $a^{<t'>}$  is given by  $\alpha^{<t,t'>}$  computed as follows:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t''=1}^{T_x} \exp(e^{<t,t''>})}$$





*Remark: computation complexity is quadratic with respect to  $T_x$ .*





\* \* \*

### 3 Deep Learning Tips and Tricks

#### 3.1 Data processing

□ **Data augmentation** – Deep learning models usually need a lot of data to be properly trained. It is often useful to get more data from the existing ones using data augmentation techniques. The main ones are summed up in the table below. More precisely, given the following input image, here are the techniques that we can apply:

Original	Flip	Rotation	Random crop
			
- Image without any modification	- Flipped with respect to an axis for which the meaning of the image is preserved	- Rotation with a slight angle - Simulates incorrect horizon calibration	- Random focus on one part of the image - Several random crops can be done in a row

Color shift	Noise addition	Information loss	Contrast change
			
- Nuances of RGB is slightly changed - Captures noise that can occur with light exposure	- Addition of noise - More tolerance to quality variation of inputs	- Parts of image ignored - Mimics potential loss of parts of image	- Luminosity changes - Controls difference in exposition due to time of day

□ **Batch normalization** – It is a step of hyperparameter  $\gamma, \beta$  that normalizes the batch  $\{x_i\}$ . By noting  $\mu_B, \sigma_B^2$  the mean and variance of that we want to correct to the batch, it is done as follows:

$$x_i \leftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

It is usually done after a fully connected/convolutional layer and before a non-linearity layer and aims at allowing higher learning rates and reducing the strong dependence on initialization.

#### 3.2 Training a neural network

##### 3.2.1 Definitions

□ **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

□ **Mini-batch gradient descent** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini-batches, where the number of data points in a batch is a hyperparameter that we can tune.

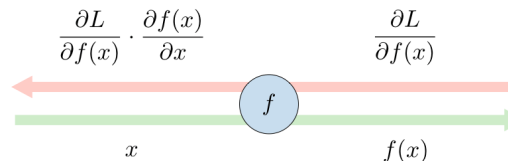
□ **Loss function** – In order to quantify how a given model performs, the loss function  $L$  is usually used to evaluate to what extent the actual outputs  $y$  are correctly predicted by the model outputs  $z$ .

□ **Cross-entropy loss** – In the context of binary classification in neural networks, the cross-entropy loss  $L(z, y)$  is commonly used and is defined as follows:

$$L(z, y) = - \left[ y \log(z) + (1 - y) \log(1 - z) \right]$$

##### 3.2.2 Finding optimal weights

□ **Backpropagation** – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight  $w$  is computed using the chain rule.

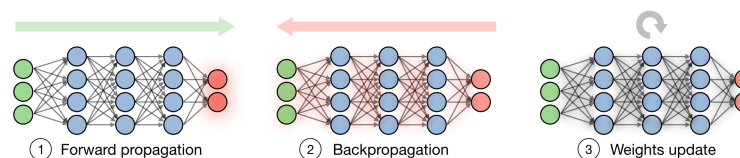


Using this method, each weight is updated with the rule:

$$w \leftarrow w - \alpha \frac{\partial L(z, y)}{\partial w}$$

□ **Updating weights** – In a neural network, weights are updated as follows:

- Step 1: Take a batch of training data and perform forward propagation to compute the loss.
- Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
- Step 3: Use the gradients to update the weights of the network.



### 3.3 Parameter tuning

#### 3.3.1 Weights initialization

□ **Xavier initialization** – Instead of initializing the weights in a purely random manner, Xavier initialization enables to have initial weights that take into account characteristics that are unique to the architecture.

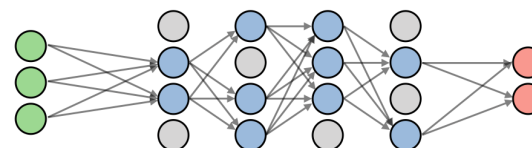
□ **Transfer learning** – Training a deep learning model requires a lot of data and more importantly a lot of time. It is often useful to take advantage of pre-trained weights on huge datasets that took days/weeks to train, and leverage it towards our use case. Depending on how much data we have at hand, here are the different ways to leverage this:

Method	Explanation	Update of $w$	Update of $b$
Momentum	- Dampens oscillations - Improvement to SGD - 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	- Root Mean Square propagation - Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \leftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$
Adam	- Adaptive Moment estimation - Most popular method - 4 parameters to tune	$w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$	$b \leftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$

Remark: other methods include Adadelta, Adagrad and SGD.

#### 3.4 Regularization

□ **Dropout** – Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability  $p > 0$ . It forces the model to avoid relying too much on particular sets of features.



Remark: most deep learning frameworks parametrize dropout through the 'keep' parameter  $1 - p$ .

□ **Weight regularization** – In order to make sure that the weights are not too large and that the model is not overfitting the training set, regularization techniques are usually performed on the model weights. The main ones are summed up in the table below:

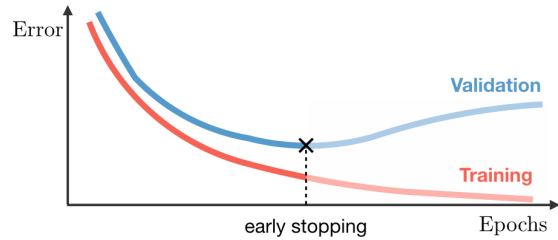
LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
$\dots + \lambda \ \theta\ _1$ $\lambda \in \mathbb{R}$	$\dots + \lambda \ \theta\ _2^2$ $\lambda \in \mathbb{R}$	$\dots + \lambda \left[ (1 - \alpha) \ \theta\ _1 + \alpha \ \theta\ _2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0, 1]$

#### 3.3.2 Optimizing convergence

□ **Learning rate** – The learning rate, often noted  $\alpha$  or sometimes  $\eta$ , indicates at which pace the weights get updated. It can be fixed or adaptively changed. The current most popular method is called Adam, which is a method that adapts the learning rate.

□ **Adaptive learning rates** – Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution. While Adam optimizer is the most commonly used technique, others can also be useful. They are summed up in the table below:

□ **Early stopping** – This regularization technique stops the training process as soon as the validation loss reaches a plateau or starts to increase.



### 3.5 Good practices

□ **Overfitting small batch** – When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself. In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it. If it cannot, it means that the model is either too complex or not complex enough to even overfit on a small batch, let alone a normal-sized training set.

□ **Gradient checking** – Gradient checking is a method used during the implementation of the backward pass of a neural network. It compares the value of the analytical gradient to the numerical gradient at given points and plays the role of a sanity-check for correctness.

	Numerical gradient	Analytical gradient
<b>Formula</b>	$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h}$	$\frac{df}{dx}(x) = f'(x)$
<b>Comments</b>	<ul style="list-style-type: none"> <li>- Expensive; loss has to be computed two times per dimension</li> <li>- Used to verify correctness of analytical implementation</li> <li>- Trade-off in choosing <math>h</math> not too small (numerical instability) nor too large (poor gradient approx.)</li> </ul>	<ul style="list-style-type: none"> <li>- 'Exact' result</li> <li>- Direct computation</li> <li>- Used in the final implementation</li> </ul>

\* \* \*

# Super VIP Cheatsheet: Artificial Intelligence

Afshine AMIDI and Shervine AMIDI

September 8, 2019

## Contents

<b>1</b>	<b>Reflex-based models</b>	<b>2</b>	<b>3</b>	<b>Variables-based models</b>	<b>12</b>
1.1	Linear predictors . . . . .	2	3.1	Constraint satisfaction problems . . . . .	12
1.1.1	Classification . . . . .	2	3.1.1	Factor graphs . . . . .	12
1.1.2	Regression . . . . .	2	3.1.2	Dynamic ordering . . . . .	12
1.2	Loss minimization . . . . .	2	3.1.3	Approximate methods . . . . .	13
1.3	Non-linear predictors . . . . .	3	3.1.4	Factor graph transformations . . . . .	13
1.4	Stochastic gradient descent . . . . .	3	3.2	Bayesian networks . . . . .	14
1.5	Fine-tuning models . . . . .	3	3.2.1	Introduction . . . . .	14
1.6	Unsupervised Learning . . . . .	4	3.2.2	Probabilistic programs . . . . .	15
1.6.1	$k$ -means . . . . .	4	3.2.3	Inference . . . . .	15
1.6.2	Principal Component Analysis . . . . .	4	<b>4</b>	<b>Logic-based models</b>	<b>16</b>
<b>2</b>	<b>States-based models</b>	<b>5</b>	4.1	Basics . . . . .	16
2.1	Search optimization . . . . .	5	4.2	Knowledge base . . . . .	17
2.1.1	Tree search . . . . .	5	4.3	Propositional logic . . . . .	18
2.1.2	Graph search . . . . .	6	4.4	First-order logic . . . . .	18
2.1.3	Learning costs . . . . .	7			
2.1.4	A* search . . . . .	7			
2.1.5	Relaxation . . . . .	8			
2.2	Markov decision processes . . . . .	8			
2.2.1	Notations . . . . .	8			
2.2.2	Applications . . . . .	9			
2.2.3	When unknown transitions and rewards . . . . .	9			
2.3	Game playing . . . . .	10			
2.3.1	Speeding up minimax . . . . .	11			
2.3.2	Simultaneous games . . . . .	11			
2.3.3	Non-zero-sum games . . . . .	12			

## 1 Reflex-based models

### 1.1 Linear predictors

In this section, we will go through reflex-based models that can improve with experience, by going through samples that have input-output pairs.

□ **Feature vector** – The feature vector of an input  $x$  is noted  $\phi(x)$  and is such that:

$$\phi(x) = \begin{bmatrix} \phi_1(x) \\ \vdots \\ \phi_d(x) \end{bmatrix} \in \mathbb{R}^d$$

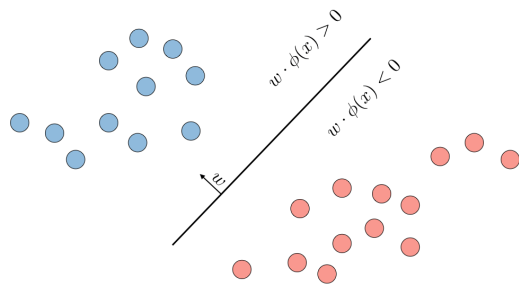
□ **Score** – The score  $s(x,w)$  of an example  $(\phi(x),y) \in \mathbb{R}^d \times \mathbb{R}$  associated to a linear model of weights  $w \in \mathbb{R}^d$  is given by the inner product:

$$s(x,w) = w \cdot \phi(x)$$

#### 1.1.1 Classification

□ **Linear classifier** – Given a weight vector  $w \in \mathbb{R}^d$  and a feature vector  $\phi(x) \in \mathbb{R}^d$ , the binary linear classifier  $f_w$  is given by:

$$f_w(x) = \text{sign}(s(x,w)) = \begin{cases} +1 & \text{if } w \cdot \phi(x) > 0 \\ -1 & \text{if } w \cdot \phi(x) < 0 \\ ? & \text{if } w \cdot \phi(x) = 0 \end{cases}$$



□ **Margin** – The margin  $m(x,y,w) \in \mathbb{R}$  of an example  $(\phi(x),y) \in \mathbb{R}^d \times \{-1, +1\}$  associated to a linear model of weights  $w \in \mathbb{R}^d$  quantifies the confidence of the prediction: larger values are better. It is given by:

$$m(x,y,w) = s(x,w) \times y$$

#### 1.1.2 Regression

□ **Linear regression** – Given a weight vector  $w \in \mathbb{R}^d$  and a feature vector  $\phi(x) \in \mathbb{R}^d$ , the output of a linear regression of weights  $w$  denoted as  $f_w$  is given by:

$$f_w(x) = s(x,w)$$

□ **Residual** – The residual  $\text{res}(x,y,w) \in \mathbb{R}$  is defined as being the amount by which the prediction  $f_w(x)$  overshoots the target  $y$ :

$$\text{res}(x,y,w) = f_w(x) - y$$

### 1.2 Loss minimization

□ **Loss function** – A loss function  $\text{Loss}(x,y,w)$  quantifies how unhappy we are with the weights  $w$  of the model in the prediction task of output  $y$  from input  $x$ . It is a quantity we want to minimize during the training process.

□ **Classification case** – The classification of a sample  $x$  of true label  $y \in \{-1, +1\}$  with a linear model of weights  $w$  can be done with the predictor  $f_w(x) \triangleq \text{sign}(s(x,w))$ . In this situation, a metric of interest quantifying the quality of the classification is given by the margin  $m(x,y,w)$ , and can be used with the following loss functions:

Name	Zero-one loss	Hinge loss	Logistic loss
$\text{Loss}(x,y,w)$	$1_{\{m(x,y,w) \leq 0\}}$	$\max(1 - m(x,y,w), 0)$	$\log(1 + e^{-m(x,y,w)})$
Illustration			

□ **Regression case** – The prediction of a sample  $x$  of true label  $y \in \mathbb{R}$  with a linear model of weights  $w$  can be done with the predictor  $f_w(x) \triangleq s(x,w)$ . In this situation, a metric of interest quantifying the quality of the regression is given by the margin  $\text{res}(x,y,w)$  and can be used with the following loss functions:

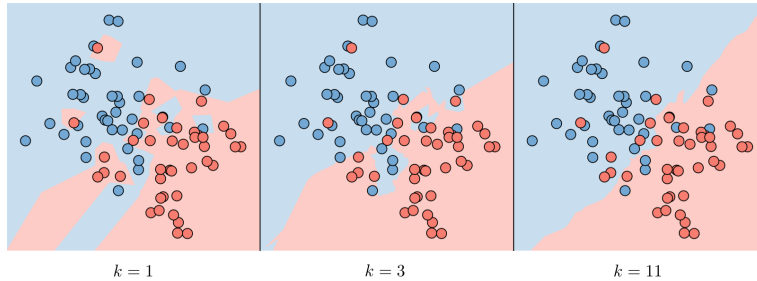
Name	Squared loss	Absolute deviation loss
$\text{Loss}(x,y,w)$	$(\text{res}(x,y,w))^2$	$ \text{res}(x,y,w) $
Illustration		

□ **Loss minimization framework** – In order to train a model, we want to minimize the training loss is defined as follows:

$$\text{TrainLoss}(w) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x,y,w)$$

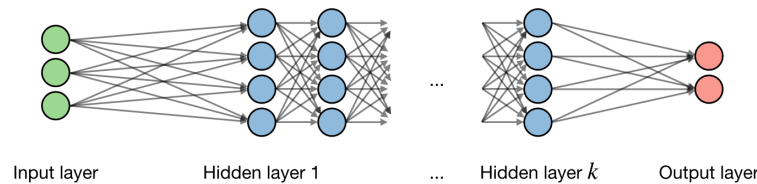
### 1.3 Non-linear predictors

□ **k-nearest neighbors** – The k-nearest neighbors algorithm, commonly known as k-NN, is a non-parametric approach where the response of a data point is determined by the nature of its k neighbors from the training set. It can be used in both classification and regression settings.



*Remark: the higher the parameter k, the higher the bias, and the lower the parameter k, the higher the variance.*

□ **Neural networks** – Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks. The vocabulary around neural networks architectures is described in the figure below:



By noting  $i$  the  $i^{th}$  layer of the network and  $j$  the  $j^{th}$  hidden unit of the layer, we have:

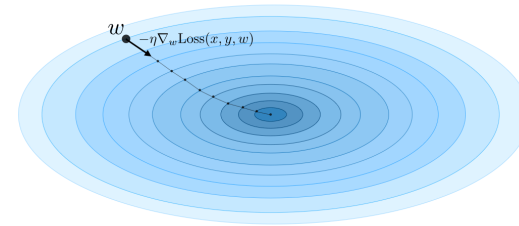
$$z_j^{[i]} = w_j^{[i]T} x + b_j^{[i]}$$

where we note  $w, b, x, z$  the weight, bias, input and non-activated output of the neuron respectively.

### 1.4 Stochastic gradient descent

□ **Gradient descent** – By noting  $\eta \in \mathbb{R}$  the learning rate (also called step size), the update rule for gradient descent is expressed with the learning rate and the loss function  $\text{Loss}(x,y,w)$  as follows:

$$w \leftarrow w - \eta \nabla_w \text{Loss}(x,y,w)$$



□ **Stochastic updates** – Stochastic gradient descent (SGD) updates the parameters of the model one training example  $(\phi(x),y) \in \mathcal{D}_{\text{train}}$  at a time. This method leads to sometimes noisy, but fast updates.

□ **Batch updates** – Batch gradient descent (BGD) updates the parameters of the model one batch of examples (e.g. the entire training set) at a time. This method computes stable update directions, at a greater computational cost.

### 1.5 Fine-tuning models

□ **Hypothesis class** – A hypothesis class  $\mathcal{F}$  is the set of possible predictors with a fixed  $\phi(x)$  and varying  $w$ :

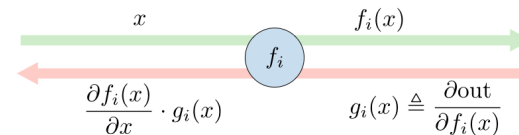
$$\mathcal{F} = \{f_w : w \in \mathbb{R}^d\}$$

□ **Logistic function** – The logistic function  $\sigma$ , also called the sigmoid function, is defined as:

$$\forall z \in ]-\infty, +\infty[, \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

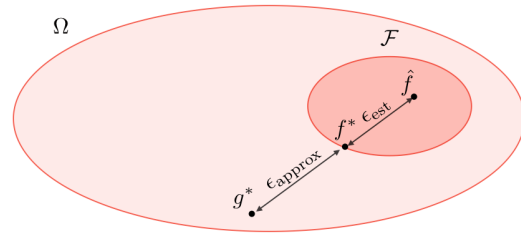
*Remark: we have  $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ .*

□ **Backpropagation** – The forward pass is done through  $f_i$ , which is the value for the sub-expression rooted at  $i$ , while the backward pass is done through  $g_i = \frac{\partial \text{out}}{\partial f_i}$  and represents how  $f_i$  influences the output.



□ **Approximation and estimation error** – The approximation error  $\epsilon_{\text{approx}}$  represents how far the entire hypothesis class  $\mathcal{F}$  is from the target predictor  $g^*$ , while the estimation error  $\epsilon_{\text{est}}$  quantifies how good the predictor  $\hat{f}$  is with respect to the best predictor  $f^*$  of the hypothesis class  $\mathcal{F}$ .





□ **Regularization** – The regularization procedure aims at avoiding the model to overfit the data and thus deals with high variance issues. The following table sums up the different types of commonly used regularization techniques:

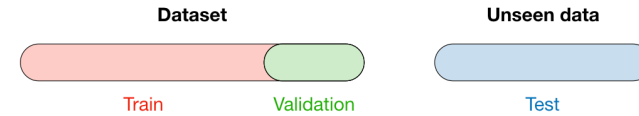
LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
$\dots + \lambda \ \theta\ _1$ $\lambda \in \mathbb{R}$	$\dots + \lambda \ \theta\ _2^2$ $\lambda \in \mathbb{R}$	$\dots + \lambda \left[ (1 - \alpha) \ \theta\ _1 + \alpha \ \theta\ _2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0,1]$

□ **Hyperparameters** – Hyperparameters are the properties of the learning algorithm, and include features, regularization parameter  $\lambda$ , number of iterations  $T$ , step size  $\eta$ , etc.

□ **Sets vocabulary** – When selecting a model, we distinguish 3 different parts of the data that we have as follows:

Training set	Validation set	Testing set
- Model is trained - Usually 80 of the dataset	- Model is assessed - Usually 20 of the dataset - Also called hold-out	- Model gives predictions - Unseen data or development set

Once the model has been chosen, it is trained on the entire dataset and tested on the unseen test set. These are represented in the figure below:



## 1.6 Unsupervised Learning

The class of unsupervised learning methods aims at discovering the structure of the data, which may have of rich latent structures.

### 1.6.1 k-means

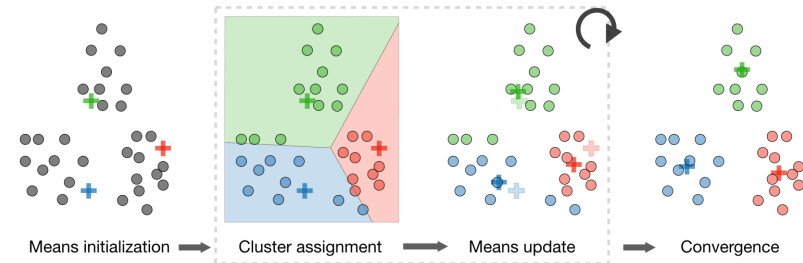
□ **Clustering** – Given a training set of input points  $\mathcal{D}_{\text{train}}$ , the goal of a clustering algorithm is to assign each point  $\phi(x_i)$  to a cluster  $z_i \in \{1, \dots, k\}$ .

□ **Objective function** – The loss function for one of the main clustering algorithms,  $k$ -means, is given by:

$$\text{Loss}_{k\text{-means}}(x, \mu) = \sum_{i=1}^n \|\phi(x_i) - \mu_{z_i}\|^2$$

□ **Algorithm** – After randomly initializing the cluster centroids  $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^n$ , the  $k$ -means algorithm repeats the following step until convergence:

$$z_i = \arg \min_j \|\phi(x_i) - \mu_j\|^2 \quad \text{and} \quad \mu_j = \frac{\sum_{i=1}^m 1_{\{z_i=j\}} \phi(x_i)}{\sum_{i=1}^m 1_{\{z_i=j\}}}$$



### 1.6.2 Principal Component Analysis

□ **Eigenvalue, eigenvector** – Given a matrix  $A \in \mathbb{R}^{n \times n}$ ,  $\lambda$  is said to be an eigenvalue of  $A$  if there exists a vector  $z \in \mathbb{R}^n \setminus \{0\}$ , called eigenvector, such that we have:

$$Az = \lambda z$$

□ **Spectral theorem** – Let  $A \in \mathbb{R}^{n \times n}$ . If  $A$  is symmetric, then  $A$  is diagonalizable by a real orthogonal matrix  $U \in \mathbb{R}^{n \times n}$ . By noting  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ , we have:

$$\exists \Lambda \text{ diagonal, } A = U\Lambda U^T$$

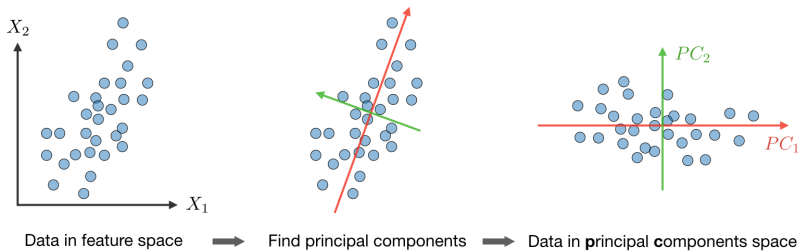
*Remark: the eigenvector associated with the largest eigenvalue is called principal eigenvector of matrix  $A$ .*

□ **Algorithm** – The Principal Component Analysis (PCA) procedure is a dimension reduction technique that projects the data on  $k$  dimensions by maximizing the variance of the data as follows:

- Step 1: Normalize the data to have a mean of 0 and standard deviation of 1.

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \quad \text{where} \quad \mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)} \quad \text{and} \quad \sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$$

- Step 2: Compute  $\Sigma = \frac{1}{m} \sum_{i=1}^m x^{(i)}x^{(i)T} \in \mathbb{R}^{n \times n}$ , which is symmetric with real eigenvalues.
- Step 3: Compute  $u_1, \dots, u_k \in \mathbb{R}^n$  the  $k$  orthogonal principal eigenvectors of  $\Sigma$ , i.e. the orthogonal eigenvectors of the  $k$  largest eigenvalues.
- Step 4: Project the data on  $\text{span}_{\mathbb{R}}(u_1, \dots, u_k)$ . This procedure maximizes the variance among all  $k$ -dimensional spaces.



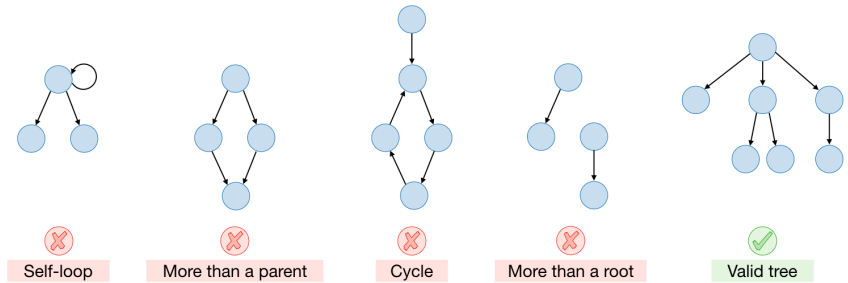
## 2 States-based models

### 2.1 Search optimization

In this section, we assume that by accomplishing action  $a$  from state  $s$ , we deterministically arrive in state  $\text{Succ}(s,a)$ . The goal here is to determine a sequence of actions  $(a_1, a_2, a_3, a_4, \dots)$  that starts from an initial state and leads to an end state. In order to solve this kind of problem, our objective will be to find the minimum cost path by using states-based models.

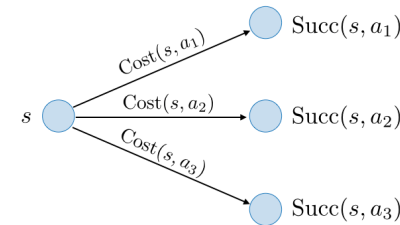
#### 2.1.1 Tree search

This category of states-based algorithms explores all possible states and actions. It is quite memory efficient, and is suitable for huge state spaces but the runtime can become exponential in the worst cases.



□ **Search problem** – A search problem is defined with:

- a starting state  $s_{\text{start}}$
- possible actions  $\text{Actions}(s)$  from state  $s$
- action cost  $\text{Cost}(s,a)$  from state  $s$  with action  $a$
- successor  $\text{Succ}(s,a)$  of state  $s$  after action  $a$
- whether an end state was reached  $\text{IsEnd}(s)$

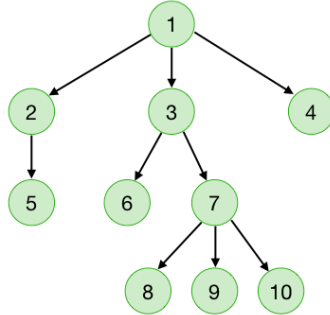


The objective is to find a path that minimizes the cost.

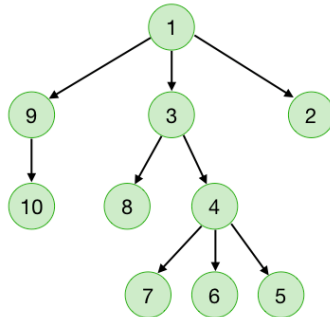
□ **Backtracking search** – Backtracking search is a naive recursive algorithm that tries all possibilities to find the minimum cost path. Here, action costs can be either positive or negative.

□ **Breadth-first search (BFS)** – Breadth-first search is a graph search algorithm that does a level-by-level traversal. We can implement it iteratively with the help of a queue that stores at

each step future nodes to be visited. For this algorithm, we can assume action costs to be equal to a constant  $c \geq 0$ .



**Depth-first search (DFS)** – Depth-first search is a search algorithm that traverses a graph by following each path as deep as it can. We can implement it recursively, or iteratively with the help of a stack that stores at each step future nodes to be visited. For this algorithm, action costs are assumed to be equal to 0.



**Iterative deepening** – The iterative deepening trick is a modification of the depth-first search algorithm so that it stops after reaching a certain depth, which guarantees optimality when all action costs are equal. Here, we assume that action costs are equal to a constant  $c \geq 0$ .

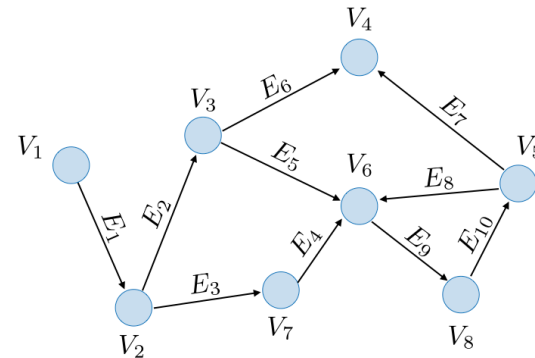
**Tree search algorithms summary** – By noting  $b$  the number of actions per state,  $d$  the solution depth, and  $D$  the maximum depth, we have:

Algorithm	Action costs	Space	Time
Backtracking search	any	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
Breadth-first search	$c \geq 0$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^d)$
Depth-first search	0	$\mathcal{O}(D)$	$\mathcal{O}(b^D)$
DFS-Iterative deepening	$c \geq 0$	$\mathcal{O}(d)$	$\mathcal{O}(b^d)$

### 2.1.2 Graph search

This category of states-based algorithms aims at constructing optimal paths, enabling exponential savings. In this section, we will focus on dynamic programming and uniform cost search.

**Graph** – A graph is comprised of a set of vertices  $V$  (also called nodes) as well as a set of edges  $E$  (also called links).

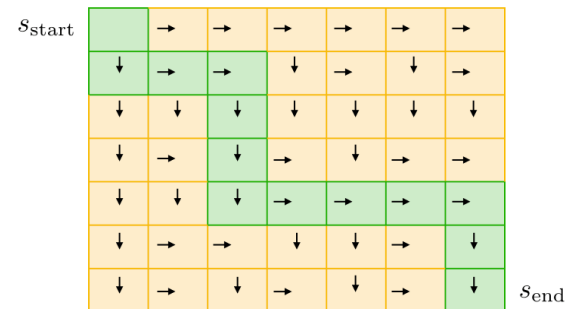


*Remark: a graph is said to be acyclic when there is no cycle.*

**State** – A state is a summary of all past actions sufficient to choose future actions optimally.

**Dynamic programming** – Dynamic programming (DP) is a backtracking search algorithm with memoization (i.e. partial results are saved) whose goal is to find a minimum cost path from state  $s$  to an end state  $s_{end}$ . It can potentially have exponential savings compared to traditional graph search algorithms, and has the property to only work for acyclic graphs. For any given state  $s$ , the future cost is computed as follows:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s,a) + \text{FutureCost}(\text{Succ}(s,a))] & \text{otherwise} \end{cases}$$

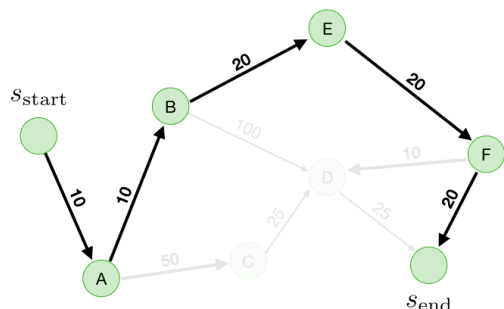


*Remark: the figure above illustrates a bottom-to-top approach whereas the formula provides the intuition of a top-to-bottom problem resolution.*

**Types of states** – The table below presents the terminology when it comes to states in the context of uniform cost search:

State	Explanation
Explored $\mathcal{E}$	States for which the optimal path has already been found
Frontier $\mathcal{F}$	States seen for which we are still figuring out how to get there with the cheapest cost
Unexplored $\mathcal{U}$	States not seen yet

□ **Uniform cost search** – Uniform cost search (UCS) is a search algorithm that aims at finding the shortest path from a state  $s_{start}$  to an end state  $s_{end}$ . It explores states  $s$  in increasing order of  $PastCost(s)$  and relies on the fact that all action costs are non-negative.



Remark 1: the UCS algorithm is logically equivalent to Dijkstra’s algorithm.  
 Remark 2: the algorithm would not work for a problem with negative action costs, and adding a positive constant to make them non-negative would not solve the problem since this would end up being a different problem.

□ **Correctness theorem** – When a state  $s$  is popped from the frontier  $\mathcal{F}$  and moved to explored set  $\mathcal{E}$ , its priority is equal to  $PastCost(s)$  which is the minimum cost path from  $s_{start}$  to  $s$ .

□ **Graph search algorithms summary** – By noting  $N$  the number of total states,  $n$  of which are explored before the end state  $s_{end}$ , we have:

Algorithm	Acyclicity	Costs	Time/space
Dynamic programming	yes	any	$\mathcal{O}(N)$
Uniform cost search	no	$c \geq 0$	$\mathcal{O}(n \log(n))$

Remark: the complexity countdown supposes the number of possible actions per state to be constant.

### 2.1.3 Learning costs

Suppose we are not given the values of  $Cost(s,a)$ , we want to estimate these quantities from a training set of minimizing-cost-path sequence of actions  $(a_1, a_2, \dots, a_k)$ .

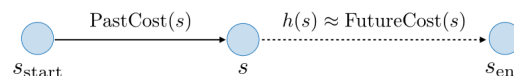
□ **Structured perceptron** – The structured perceptron is an algorithm aiming at iteratively learning the cost of each state-action pair. At each step, it:

- decreases the estimated cost of each state-action of the true minimizing path  $y$  given by the training data,
- increases the estimated cost of each state-action of the current predicted path  $y'$  inferred from the learned weights.

Remark: there are several versions of the algorithm, one of which simplifies the problem to only learning the cost of each action  $a$ , and the other parametrizes  $Cost(s,a)$  to a feature vector of learnable weights.

### 2.1.4 A\* search

□ **Heuristic function** – A heuristic is a function  $h$  over states  $s$ , where each  $h(s)$  aims at estimating  $FutureCost(s)$ , the cost of the path from  $s$  to  $s_{end}$ .



□ **Algorithm** – A\* is a search algorithm that aims at finding the shortest path from a state  $s$  to an end state  $s_{end}$ . It explores states  $s$  in increasing order of  $PastCost(s) + h(s)$ . It is equivalent to a uniform cost search with edge costs  $Cost'(s,a)$  given by:

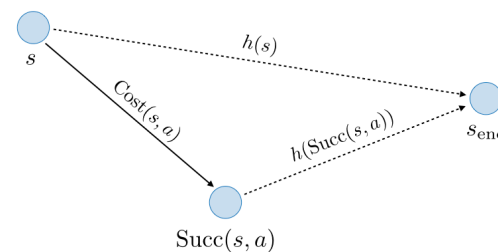
$$Cost'(s,a) = Cost(s,a) + h(Succ(s,a)) - h(s)$$

Remark: this algorithm can be seen as a biased version of UCS exploring states estimated to be closer to the end state.

□ **Consistency** – A heuristic  $h$  is said to be consistent if it satisfies the two following properties:

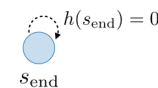
- For all states  $s$  and actions  $a$ ,

$$h(s) \leq Cost(s,a) + h(Succ(s,a))$$



- The end state verifies the following:

$$h(s_{end}) = 0$$



□ **Correctness** – If  $h$  is consistent, then  $A^*$  returns the minimum cost path.

□ **Admissibility** – A heuristic  $h$  is said to be admissible if we have:

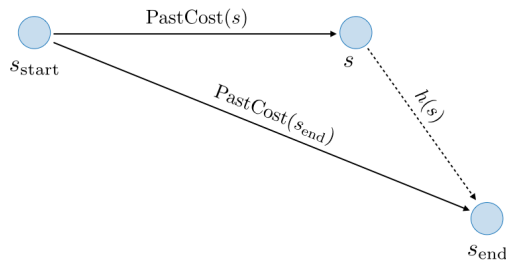
$$h(s) \leq \text{FutureCost}(s)$$

□ **Theorem** – Let  $h(s)$  be a given heuristic. We have:

$$h(s) \text{ consistent} \implies h(s) \text{ admissible}$$

□ **Efficiency** –  $A^*$  explores all states  $s$  satisfying the following equation:

$$\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$$



*Remark: larger values of  $h(s)$  is better as this equation shows it will restrict the set of states  $s$  going to be explored.*

### 2.1.5 Relaxation

It is a framework for producing consistent heuristics. The idea is to find closed-form reduced costs by removing constraints and use them as heuristics.

□ **Relaxed search problem** – The relaxation of search problem  $P$  with costs  $\text{Cost}$  is noted  $P_{\text{rel}}$  with costs  $\text{Cost}_{\text{rel}}$ , and satisfies the identity:

$$\text{Cost}_{\text{rel}}(s,a) \leq \text{Cost}(s,a)$$

□ **Relaxed heuristic** – Given a relaxed search problem  $P_{\text{rel}}$ , we define the relaxed heuristic  $h(s) = \text{FutureCost}_{\text{rel}}(s)$  as the minimum cost path from  $s$  to an end state in the graph of costs  $\text{Cost}_{\text{rel}}(s,a)$ .

□ **Consistency of relaxed heuristics** – Let  $P_{\text{rel}}$  be a given relaxed problem. By theorem, we have:

$$h(s) = \text{FutureCost}_{\text{rel}}(s) \implies h(s) \text{ consistent}$$

□ **Tradeoff when choosing heuristic** – We have to balance two aspects in choosing a heuristic:

- Computational efficiency:  $h(s) = \text{FutureCost}_{\text{rel}}(s)$  must be easy to compute. It has to produce a closed form, easier search and independent subproblems.
- Good enough approximation: the heuristic  $h(s)$  should be close to  $\text{FutureCost}(s)$  and we have thus to not remove too many constraints.

□ **Max heuristic** – Let  $h_1(s), h_2(s)$  be two heuristics. We have the following property:

$$h_1(s), h_2(s) \text{ consistent} \implies h(s) = \max\{h_1(s), h_2(s)\} \text{ consistent}$$

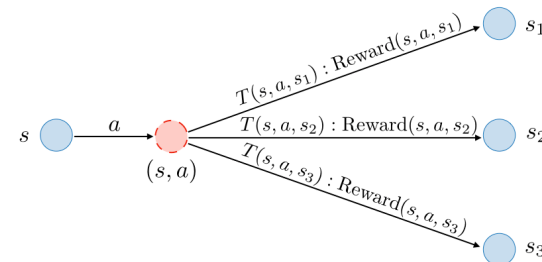
## 2.2 Markov decision processes

In this section, we assume that performing action  $a$  from state  $s$  can lead to several states  $s'_1, s'_2, \dots$  in a probabilistic manner. In order to find our way between an initial state and an end state, our objective will be to find the maximum value policy by using Markov decision processes that help us cope with randomness and uncertainty.

### 2.2.1 Notations

□ **Definition** – The objective of a Markov decision process is to maximize rewards. It is defined with:

- a starting state  $s_{\text{start}}$
- possible actions  $\text{Actions}(s)$  from state  $s$
- transition probabilities  $T(s,a,s')$  from  $s$  to  $s'$  with action  $a$
- rewards  $\text{Reward}(s,a,s')$  from  $s$  to  $s'$  with action  $a$
- whether an end state was reached  $\text{IsEnd}(s)$
- a discount factor  $0 \leq \gamma \leq 1$



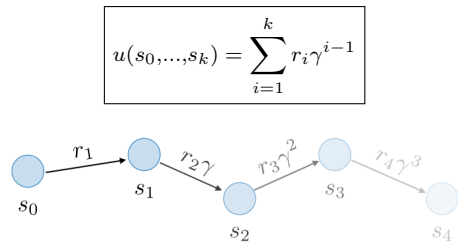
□ **Transition probabilities** – The transition probability  $T(s,a,s')$  specifies the probability of going to state  $s'$  after action  $a$  is taken in state  $s$ . Each  $s' \mapsto T(s,a,s')$  is a probability distribution, which means that:

$$\forall s,a, \sum_{s' \in \text{States}} T(s,a,s') = 1$$

□ **Policy** – A policy  $\pi$  is a function that maps each state  $s$  to an action  $a$ , i.e.

$$\pi : s \mapsto a$$

□ **Utility** – The utility of a path  $(s_0, \dots, s_k)$  is the discounted sum of the rewards on that path. In other words,



Remark: the figure above is an illustration of the case  $k = 4$ .

□ **Q-value** – The  $Q$ -value of a policy  $\pi$  by taking action  $a$  from state  $s$ , also noted  $Q_\pi(s, a)$ , is the expected utility of taking action  $a$  from state  $s$  and then following policy  $\pi$ . It is defined as follows:

$$Q_\pi(s, a) = \sum_{s' \in \text{States}} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

□ **Value of a policy** – The value of a policy  $\pi$  from state  $s$ , also noted  $V_\pi(s)$ , is the expected utility by following policy  $\pi$  from state  $s$  over random paths. It is defined as follows:

$$V_\pi(s) = Q_\pi(s, \pi(s))$$

Remark:  $V_\pi(s)$  is equal to 0 if  $s$  is an end state.

### 2.2.2 Applications

□ **Policy evaluation** – Given a policy  $\pi$ , policy evaluation is an iterative algorithm that computes  $V_\pi$ . It is done as follows:

- **Initialization**: for all states  $s$ , we have

$$V_\pi^{(0)}(s) \leftarrow 0$$

- **Iteration**: for  $t$  from 1 to  $T_{PE}$ , we have

$$\forall s, \quad V_\pi^{(t)}(s) \leftarrow Q_\pi^{(t-1)}(s, \pi(s))$$

with

$$Q_\pi^{(t-1)}(s, \pi(s)) = \sum_{s' \in \text{States}} T(s, \pi(s), s') [\text{Reward}(s, \pi(s), s') + \gamma V_\pi^{(t-1)}(s')]$$

Remark: by noting  $S$  the number of states,  $A$  the number of actions per state,  $S'$  the number of successors and  $T$  the number of iterations, then the time complexity is of  $\mathcal{O}(T_{PE}SS')$ .

□ **Optimal Q-value** – The optimal  $Q$ -value  $Q_{\text{opt}}(s, a)$  of state  $s$  with action  $a$  is defined to be the maximum  $Q$ -value attained by any policy starting. It is computed as follows:

$$Q_{\text{opt}}(s, a) = \sum_{s' \in \text{States}} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')]$$

□ **Optimal value** – The optimal value  $V_{\text{opt}}(s)$  of state  $s$  is defined as being the maximum value attained by any policy. It is computed as follows:

$$V_{\text{opt}}(s) = \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

□ **Optimal policy** – The optimal policy  $\pi_{\text{opt}}$  is defined as being the policy that leads to the optimal values. It is defined by:

$$\forall s, \quad \pi_{\text{opt}}(s) = \operatorname{argmax}_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

□ **Value iteration** – Value iteration is an algorithm that finds the optimal value  $V_{\text{opt}}$  as well as the optimal policy  $\pi_{\text{opt}}$ . It is done as follows:

- **Initialization**: for all states  $s$ , we have

$$V_{\text{opt}}^{(0)}(s) \leftarrow 0$$

- **Iteration**: for  $t$  from 1 to  $T_{VI}$ , we have

$$\forall s, \quad V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} Q_{\text{opt}}^{(t-1)}(s, a)$$

with

$$Q_{\text{opt}}^{(t-1)}(s, a) = \sum_{s' \in \text{States}} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s')]$$

Remark: if we have either  $\gamma < 1$  or the MDP graph being acyclic, then the value iteration algorithm is guaranteed to converge to the correct answer.

### 2.2.3 When unknown transitions and rewards

Now, let's assume that the transition probabilities and the rewards are unknown.

□ **Model-based Monte Carlo** – The model-based Monte Carlo method aims at estimating  $T(s, a, s')$  and  $\text{Reward}(s, a, s')$  using Monte Carlo simulation with:

$$\widehat{T}(s, a, s') = \frac{\# \text{ times } (s, a, s') \text{ occurs}}{\# \text{ times } (s, a) \text{ occurs}}$$

and

$$\widehat{\text{Reward}}(s, a, s') = r \text{ in } (s, a, r, s')$$

These estimations will be then used to deduce  $Q$ -values, including  $Q_\pi$  and  $Q_{\text{opt}}$ .

*Remark: model-based Monte Carlo is said to be off-policy, because the estimation does not depend on the exact policy.*

□ **Model-free Monte Carlo** – The model-free Monte Carlo method aims at directly estimating  $Q_\pi$ , as follows:

$$\widehat{Q}_\pi(s,a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a$$

where  $u_t$  denotes the utility starting at step  $t$  of a given episode.

*Remark: model-free Monte Carlo is said to be on-policy, because the estimated value is dependent on the policy  $\pi$  used to generate the data.*

□ **Equivalent formulation** – By introducing the constant  $\eta = \frac{1}{1+(\#\text{updates to } (s,a))}$  and for each  $(s,a,u)$  of the training set, the update rule of model-free Monte Carlo has a convex combination formulation:

$$\widehat{Q}_\pi(s,a) \leftarrow (1 - \eta)\widehat{Q}_\pi(s,a) + \eta u$$

as well as a stochastic gradient formulation:

$$\widehat{Q}_\pi(s,a) \leftarrow \widehat{Q}_\pi(s,a) - \eta(\widehat{Q}_\pi(s,a) - u)$$

□ **SARSA** – State-action-reward-state-action (SARSA) is a bootstrapping method estimating  $Q_\pi$  by using both raw data and estimates as part of the update rule. For each  $(s,a,r,s',a')$ , we have:

$$\widehat{Q}_\pi(s,a) \leftarrow (1 - \eta)\widehat{Q}_\pi(s,a) + \eta \left[ r + \gamma \widehat{Q}_\pi(s',a') \right]$$

*Remark: the SARSA estimate is updated on the fly as opposed to the model-free Monte Carlo one where the estimate can only be updated at the end of the episode.*

□ **Q-learning** – Q-learning is an off-policy algorithm that produces an estimate for  $Q_{\text{opt}}$ . On each  $(s,a,r,s',a')$ , we have:

$$\widehat{Q}_{\text{opt}}(s,a) \leftarrow (1 - \eta)\widehat{Q}_{\text{opt}}(s,a) + \eta \left[ r + \gamma \max_{a' \in \text{Actions}(s')} \widehat{Q}_{\text{opt}}(s',a') \right]$$

□ **Epsilon-greedy** – The epsilon-greedy policy is an algorithm that balances exploration with probability  $\epsilon$  and exploitation with probability  $1 - \epsilon$ . For a given state  $s$ , the policy  $\pi_{\text{act}}$  is computed as follows:

$$\pi_{\text{act}}(s) = \begin{cases} \operatorname{argmax}_{a \in \text{Actions}} \widehat{Q}_{\text{opt}}(s,a) & \text{with proba } 1 - \epsilon \\ \text{random from Actions}(s) & \text{with proba } \epsilon \end{cases}$$

### 2.3 Game playing

In games (e.g. chess, backgammon, Go), other agents are present and need to be taken into account when constructing our policy.

□ **Game tree** – A game tree is a tree that describes the possibilities of a game. In particular, each node is a decision point for a player and each root-to-leaf path is a possible outcome of the game.

□ **Two-player zero-sum game** – It is a game where each state is fully observed and such that players take turns. It is defined with:

- a starting state  $s_{\text{start}}$
- possible actions  $\text{Actions}(s)$  from state  $s$
- successors  $\text{Succ}(s,a)$  from states  $s$  with actions  $a$
- whether an end state was reached  $\text{IsEnd}(s)$
- the agent's utility  $\text{Utility}(s)$  at end state  $s$
- the player  $\text{Player}(s)$  who controls state  $s$

*Remark: we will assume that the utility of the agent has the opposite sign of the one of the opponent.*

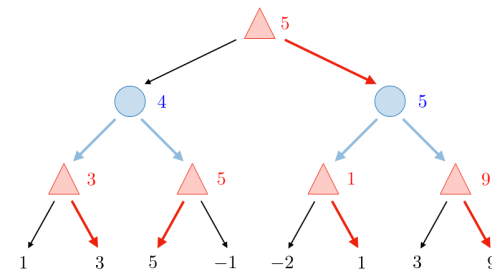
□ **Types of policies** – There are two types of policies:

- Deterministic policies, noted  $\pi_p(s)$ , which are actions that player  $p$  takes in state  $s$ .
- Stochastic policies, noted  $\pi_p(s,a) \in [0,1]$ , which are probabilities that player  $p$  takes action  $a$  in state  $s$ .

□ **Expectimax** – For a given state  $s$ , the expectimax value  $V_{\text{exptmax}}(s)$  is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy  $\pi_{\text{opp}}$ . It is computed as follows:

$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s,a) V_{\text{exptmax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

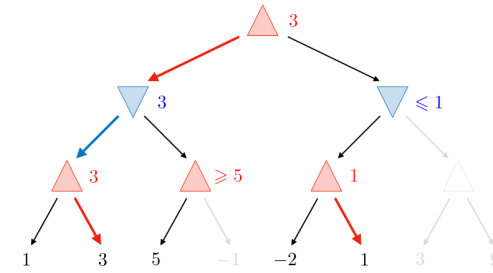
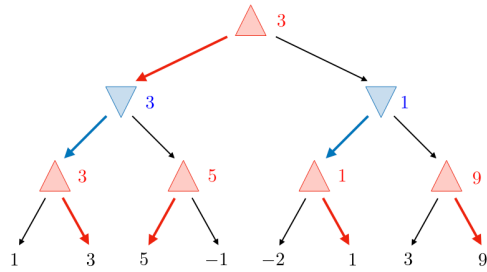
*Remark: expectimax is the analog of value iteration for MDPs.*



□ **Minimax** – The goal of minimax policies is to find an optimal policy against an adversary by assuming the worst case, i.e. that the opponent is doing everything to minimize the agent's utility. It is done as follows:

$$V_{\text{minimax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minimax}}(\text{Succ}(s,a)) & \text{Player}(s) = \text{opp} \end{cases}$$

Remark: we can extract  $\pi_{max}$  and  $\pi_{min}$  from the minimax value  $V_{minimax}$ .



□ **Minimax properties** – By noting  $V$  the value function, there are 3 properties around minimax to have in mind:

- *Property 1:* if the agent were to change its policy to any  $\pi_{agent}$ , then the agent would be no better off.

$$\forall \pi_{agent}, \quad V(\pi_{max}, \pi_{min}) \geq V(\pi_{agent}, \pi_{min})$$

- *Property 2:* if the opponent changes its policy from  $\pi_{min}$  to  $\pi_{opp}$ , then he will be no better off.

$$\forall \pi_{opp}, \quad V(\pi_{max}, \pi_{min}) \leq V(\pi_{max}, \pi_{opp})$$

- *Property 3:* if the opponent is known to be not playing the adversarial policy, then the minimax policy might not be optimal for the agent.

$$\forall \pi, \quad V(\pi_{max}, \pi) \leq V(\pi_{exptmax}, \pi)$$

In the end, we have the following relationship:

$$V(\pi_{exptmax}, \pi_{min}) \leq V(\pi_{max}, \pi_{min}) \leq V(\pi_{max}, \pi) \leq V(\pi_{exptmax}, \pi)$$

### 2.3.1 Speeding up minimax

□ **Evaluation function** – An evaluation function is a domain-specific and approximate estimate of the value  $V_{minimax}(s)$ . It is noted  $Eval(s)$ .

Remark:  $FutureCost(s)$  is an analogy for search problems.

□ **Alpha-beta pruning** – Alpha-beta pruning is a domain-general exact method optimizing the minimax algorithm by avoiding the unnecessary exploration of parts of the game tree. To do so, each player keeps track of the best value they can hope for (stored in  $\alpha$  for the maximizing player and in  $\beta$  for the minimizing player). At a given step, the condition  $\beta < \alpha$  means that the optimal path is not going to be in the current branch as the earlier player had a better option at their disposal.

□ **TD learning** – Temporal difference (TD) learning is used when we don't know the transitions/rewards. The value is based on exploration policy. To be able to use it, we need to know rules of the game  $Succ(s,a)$ . For each  $(s,a,r,s')$ , the update is done as follows:

$$w \leftarrow w - \eta [V(s,w) - (r + \gamma V(s',w))] \nabla_w V(s,w)$$

### 2.3.2 Simultaneous games

This is the contrary of turn-based games, where there is no ordering on the player's moves.

□ **Single-move simultaneous game** – Let there be two players  $A$  and  $B$ , with given possible actions. We note  $V(a,b)$  to be  $A$ 's utility if  $A$  chooses action  $a$ ,  $B$  chooses action  $b$ .  $V$  is called the payoff matrix.

□ **Strategies** – There are two main types of strategies:

- A pure strategy is a single action:

$$a \in \text{Actions}$$

- A mixed strategy is a probability distribution over actions:

$$\forall a \in \text{Actions}, \quad 0 \leq \pi(a) \leq 1$$

□ **Game evaluation** – The value of the game  $V(\pi_A, \pi_B)$  when player  $A$  follows  $\pi_A$  and player  $B$  follows  $\pi_B$  is such that:

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a) \pi_B(b) V(a,b)$$

□ **Minimax theorem** – By noting  $\pi_A, \pi_B$  ranging over mixed strategies, for every simultaneous two-player zero-sum game with a finite number of actions, we have:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B)$$



### 2.3.3 Non-zero-sum games

□ **Payoff matrix** – We define  $V_p(\pi_A, \pi_B)$  to be the utility for player  $p$ .

□ **Nash equilibrium** – A Nash equilibrium is  $(\pi_A^*, \pi_B^*)$  such that no player has an incentive to change its strategy. We have:

$$\boxed{\forall \pi_A, V_A(\pi_A^*, \pi_B^*) \geq V_A(\pi_A, \pi_B^*)} \quad \text{and} \quad \boxed{\forall \pi_B, V_B(\pi_A^*, \pi_B^*) \geq V_B(\pi_A^*, \pi_B)}$$

*Remark: in any finite-player game with finite number of actions, there exists at least one Nash equilibrium.*

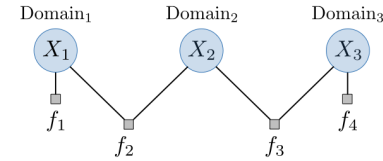
## 3 Variables-based models

### 3.1 Constraint satisfaction problems

In this section, our objective is to find maximum weight assignments of variable-based models. One advantage compared to states-based models is that these algorithms are more convenient to encode problem-specific constraints.

#### 3.1.1 Factor graphs

□ **Definition** – A factor graph, also referred to as a Markov random field, is a set of variables  $X = (X_1, \dots, X_n)$  where  $X_i \in \text{Domain}_i$  and  $m$  factors  $f_1, \dots, f_m$  with each  $f_j(X) \geq 0$ .



□ **Scope and arity** – The scope of a factor  $f_j$  is the set of variables it depends on. The size of this set is called the arity.

*Remark: factors of arity 1 and 2 are called unary and binary respectively.*

□ **Assignment weight** – Each assignment  $x = (x_1, \dots, x_n)$  yields a weight  $\text{Weight}(x)$  defined as being the product of all factors  $f_j$  applied to that assignment. Its expression is given by:

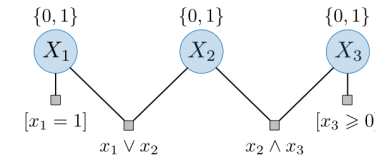
$$\boxed{\text{Weight}(x) = \prod_{j=1}^m f_j(x)}$$

□ **Constraint satisfaction problem** – A constraint satisfaction problem (CSP) is a factor graph where all factors are binary; we call them to be constraints:

$$\boxed{\forall j \in [1, m], f_j(x) \in \{0, 1\}}$$

Here, the constraint  $j$  with assignment  $x$  is said to be satisfied if and only if  $f_j(x) = 1$ .

□ **Consistent assignment** – An assignment  $x$  of a CSP is said to be consistent if and only if  $\text{Weight}(x) = 1$ , i.e. all constraints are satisfied.



#### 3.1.2 Dynamic ordering

□ **Dependent factors** – The set of dependent factors of variable  $X_i$  with partial assignment  $x$  is called  $D(x, X_i)$ , and denotes the set of factors that link  $X_i$  to already assigned variables.

□ **Backtracking search** – Backtracking search is an algorithm used to find maximum weight assignments of a factor graph. At each step, it chooses an unassigned variable and explores its values by recursion. Dynamic ordering (*i.e.* choice of variables and values) and lookahead (*i.e.* early elimination of inconsistent options) can be used to explore the graph more efficiently, although the worst-case runtime stays exponential:  $O(|\text{Domain}|^n)$ .

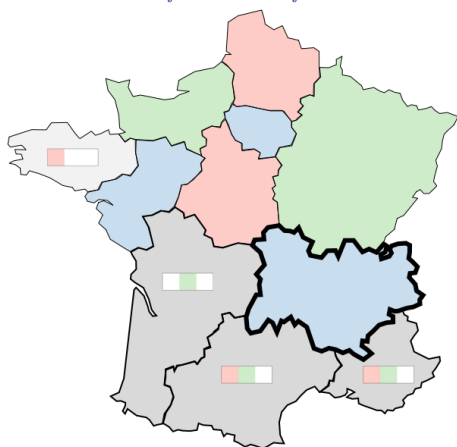
□ **Forward checking** – It is a one-step lookahead heuristic that preemptively removes inconsistent values from the domains of neighboring variables. It has the following characteristics:

- After assigning a variable  $X_i$ , it eliminates inconsistent values from the domains of all its neighbors.
- If any of these domains becomes empty, we stop the local backtracking search.
- If we un-assign a variable  $X_i$ , we have to restore the domain of its neighbors.

□ **Most constrained variable** – It is a variable-level ordering heuristic that selects the next unassigned variable that has the fewest consistent values. This has the effect of making inconsistent assignments to fail earlier in the search, which enables more efficient pruning.

□ **Least constrained value** – It is a value-level ordering heuristic that assigns the next value that yields the highest number of consistent values of neighboring variables. Intuitively, this procedure chooses first the values that are most likely to work.

*Remark: in practice, this heuristic is useful when all factors are constraints.*



The example above is an illustration of the 3-color problem with backtracking search coupled with most constrained variable exploration and least constrained value heuristic, as well as forward checking at each step.

□ **Arc consistency** – We say that arc consistency of variable  $X_l$  with respect to  $X_k$  is enforced when for each  $x_l \in \text{Domain}_l$ :

- unary factors of  $X_l$  are non-zero,
- there exists at least one  $x_k \in \text{Domain}_k$  such that any factor between  $X_l$  and  $X_k$  is non-zero.

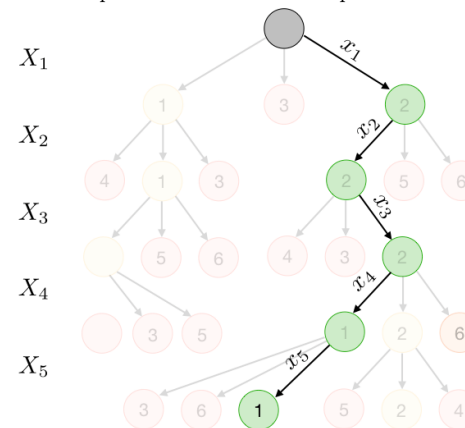
□ **AC-3** – The AC-3 algorithm is a multi-step lookahead heuristic that applies forward checking to all relevant variables. After a given assignment, it performs forward checking and then successively enforces arc consistency with respect to the neighbors of variables for which the domain change during the process.

*Remark: AC-3 can be implemented both iteratively and recursively.*

### 3.1.3 Approximate methods

□ **Beam search** – Beam search is an approximate algorithm that extends partial assignments of  $n$  variables of branching factor  $b = |\text{Domain}|$  by exploring the  $K$  top paths at each step. The beam size  $K \in \{1, \dots, b^n\}$  controls the tradeoff between efficiency and accuracy. This algorithm has a time complexity of  $O(n \cdot Kb \log(Kb))$ .

The example below illustrates a possible beam search of parameters  $K = 2$ ,  $b = 3$  and  $n = 5$ .



*Remark:  $K = 1$  corresponds to greedy search whereas  $K \rightarrow +\infty$  is equivalent to BFS tree search.*

□ **Iterated conditional modes** – Iterated conditional modes (ICM) is an iterative approximate algorithm that modifies the assignment of a factor graph one variable at a time until convergence. At step  $i$ , we assign to  $X_i$  the value  $v$  that maximizes the product of all factors connected to that variable.

*Remark: ICM may get stuck in local minima.*

□ **Gibbs sampling** – Gibbs sampling is an iterative approximate method that modifies the assignment of a factor graph one variable at a time until convergence. At step  $i$ :

- we assign to each element  $u \in \text{Domain}_i$  a weight  $w(u)$  that is the product of all factors connected to that variable,
- we sample  $v$  from the probability distribution induced by  $w$  and assign it to  $X_i$ .

*Remark: Gibbs sampling can be seen as the probabilistic counterpart of ICM. It has the advantage to be able to escape local minima in most cases.*

### 3.1.4 Factor graph transformations

□ **Independence** – Let  $A, B$  be a partitioning of the variables  $X$ . We say that  $A$  and  $B$  are independent if there are no edges between  $A$  and  $B$  and we write:

$$A, B \text{ independent} \iff A \perp\!\!\!\perp B$$

*Remark: independence is the key property that allows us to solve subproblems in parallel.*

□ **Conditional independence** – We say that  $A$  and  $B$  are conditionally independent given  $C$  if conditioning on  $C$  produces a graph in which  $A$  and  $B$  are independent. In this case, it is written:

$$A \text{ and } B \text{ cond. indep. given } C \iff A \perp\!\!\!\perp B | C$$

□ **Conditioning** – Conditioning is a transformation aiming at making variables independent that breaks up a factor graph into smaller pieces that can be solved in parallel and can use backtracking. In order to condition on a variable  $X_i = v$ , we do as follows:

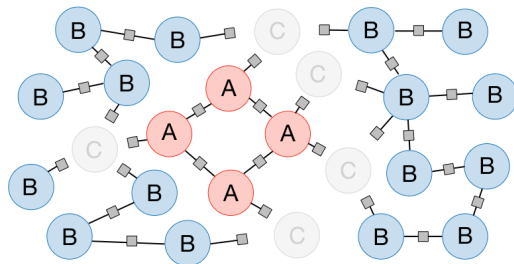
- Consider all factors  $f_1, \dots, f_k$  that depend on  $X_i$
- Remove  $X_i$  and  $f_1, \dots, f_k$
- Add  $g_j(x)$  for  $j \in \{1, \dots, k\}$  defined as:

$$g_j(x) = f_j(x \cup \{X_i : v\})$$

□ **Markov blanket** – Let  $A \subseteq X$  be a subset of variables. We define  $\text{MarkovBlanket}(A)$  to be the neighbors of  $A$  that are not in  $A$ .

□ **Proposition** – Let  $C = \text{MarkovBlanket}(A)$  and  $B = X \setminus (A \cup C)$ . Then we have:

$$A \perp\!\!\!\perp B | C$$



□ **Elimination** – Elimination is a factor graph transformation that removes  $X_i$  from the graph and solves a small subproblem conditioned on its Markov blanket as follows:

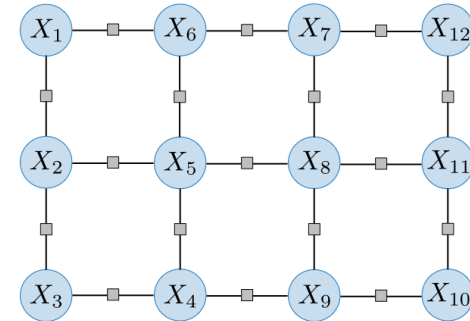
- Consider all factors  $f_{i,1}, \dots, f_{i,k}$  that depend on  $X_i$
- Remove  $X_i$  and  $f_{i,1}, \dots, f_{i,k}$
- Add  $f_{\text{new},i}(x)$  defined as:

$$f_{\text{new},i}(x) = \max_{x_i} \prod_{l=1}^k f_{i,l}(x)$$

□ **Treewidth** – The treewidth of a factor graph is the maximum arity of any factor created by variable elimination with the best variable ordering. In other words,

$$\text{Treewidth} = \min_{\text{orderings } i \in \{1, \dots, n\}} \max \text{arity}(f_{\text{new},i})$$

The example below illustrates the case of a factor graph of treewidth 3.



Remark: finding the best variable ordering is a NP-hard problem.

### 3.2 Bayesian networks

In this section, our goal will be to compute conditional probabilities. What is the probability of a query given evidence?

#### 3.2.1 Introduction

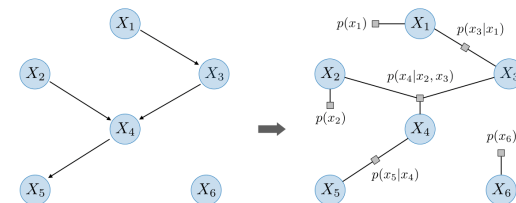
□ **Explaining away** – Suppose causes  $C_1$  and  $C_2$  influence an effect  $E$ . Conditioning on the effect  $E$  and on one of the causes (say  $C_1$ ) changes the probability of the other cause (say  $C_2$ ). In this case, we say that  $C_1$  has explained away  $C_2$ .

□ **Directed acyclic graph** – A directed acyclic graph (DAG) is a finite directed graph with no directed cycles.

□ **Bayesian network** – A Bayesian network is a directed acyclic graph (DAG) that specifies a joint distribution over random variables  $X = (X_1, \dots, X_n)$  as a product of local conditional distributions, one for each node:

$$P(X_1 = x_1, \dots, X_n = x_n) \triangleq \prod_{i=1}^n p(x_i | x_{\text{Parents}(i)})$$

Remark: Bayesian networks are factor graphs imbued with the language of probability.



□ **Locally normalized** – For each  $x_{\text{Parents}(i)}$ , all factors are local conditional distributions. Hence they have to satisfy:

$$\sum_{x_i} p(x_i | x_{\text{Parents}(i)}) = 1$$

As a result, sub-Bayesian networks and conditional distributions are consistent.

*Remark: local conditional distributions are the true conditional distributions.*

□ **Marginalization** – The marginalization of a leaf node yields a Bayesian network without that node.

### 3.2.2 Probabilistic programs

□ **Concept** – A probabilistic program randomizes variables assignment. That way, we can write down complex Bayesian networks that generate assignments without us having to explicitly specify associated probabilities.

*Remark: examples of probabilistic programs include Hidden Markov model (HMM), factorial HMM, naive Bayes, latent Dirichlet allocation, diseases and symptoms and stochastic block models.*

□ **Summary** – The table below summarizes the common probabilistic programs as well as their applications:

Program	Algorithm	Illustration	Example
Markov Model	$X_i \sim p(X_i   X_{i-1})$		Language modeling
Hidden Markov Model (HMM)	$H_t \sim p(H_t   H_{t-1})$ $E_t \sim p(E_t   H_t)$		Object tracking

Factorial HMM	$H_t^o \underset{o \in \{a,b\}}{\sim} p(H_t^o   H_{t-1}^o)$ $E_t \sim p(E_t   H_t^a, H_t^b)$		Multiple object tracking
Naive Bayes	$Y \sim p(Y)$ $W_i \sim p(W_i   Y)$		Document classification
Latent Dirichlet Allocation (LDA)	$\alpha \in \mathbb{R}^K$ distribution $Z_i \sim p(Z_i   \alpha)$ $W_i \sim p(W_i   Z_i)$		Topic modeling

### 3.2.3 Inference

□ **General probabilistic inference strategy** – The strategy to compute the probability  $P(Q | E = e)$  of query  $Q$  given evidence  $E = e$  is as follows:

- Step 1: Remove variables that are not ancestors of the query  $Q$  or the evidence  $E$  by marginalization
- Step 2: Convert Bayesian network to factor graph
- Step 3: Condition on the evidence  $E = e$
- Step 4: Remove nodes disconnected from the query  $Q$  by marginalization
- Step 5: Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering)

□ **Forward-backward algorithm** – This algorithm computes the exact value of  $P(H = h_k | E = e)$  (smoothing query) for any  $k \in \{1, \dots, L\}$  in the case of an HMM of size  $L$ . To do so, we proceed in 3 steps:

- Step 1: for  $i \in \{1, \dots, L\}$ , compute  $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1}) p(h_i | h_{i-1}) p(e_i | h_i)$
- Step 2: for  $i \in \{L, \dots, 1\}$ , compute  $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1}) p(h_{i+1} | h_i) p(e_{i+1} | h_{i+1})$
- Step 3: for  $i \in \{1, \dots, L\}$ , compute  $S_i(h_i) = \frac{F_i(h_i) B_i(h_i)}{\sum_{h_i} F_i(h_i) B_i(h_i)}$

with the convention  $F_0 = B_{L+1} = 1$ . From this procedure and these notations, we get that

$$P(H = h_k | E = e) = S_k(h_k)$$

*Remark: this algorithm interprets each assignment to be a path where each edge  $h_{i-1} \rightarrow h_i$  is of weight  $p(h_i|h_{i-1})p(e_i|h_i)$ .*

□ **Gibbs sampling** – This algorithm is an iterative approximate method that uses a small set of assignments (particles) to represent a large probability distribution. From a random assignment  $x$ , Gibbs sampling performs the following steps for  $i \in \{1, \dots, n\}$  until convergence:

- For all  $u \in \text{Domain}_i$ , compute the weight  $w(u)$  of assignment  $x$  where  $X_i = u$
- Sample  $v$  from the probability distribution induced by  $w$ :  $v \sim P(X_i = v | X_{-i} = x_{-i})$
- Set  $X_i = v$

*Remark:  $X_{-i}$  denotes  $X \setminus \{X_i\}$  and  $x_{-i}$  represents the corresponding assignment.*

□ **Particle filtering** – This algorithm approximates the posterior density of state variables given the evidence of observation variables by keeping track of  $K$  particles at a time. Starting from a set of particles  $C$  of size  $K$ , we run the following 3 steps iteratively:

- **Step 1: proposal** - For each old particle  $x_{t-1} \in C$ , sample  $x$  from the transition probability distribution  $p(x|x_{t-1})$  and add  $x$  to a set  $C'$ .
- **Step 2: weighting** - Weigh each  $x$  of the set  $C'$  by  $w(x) = p(e_t|x)$ , where  $e_t$  is the evidence observed at time  $t$ .
- **Step 3: resampling** - Sample  $K$  elements from the set  $C'$  using the probability distribution induced by  $w$  and store them in  $C$ : these are the current particles  $x_t$ .

*Remark: a more expensive version of this algorithm also keeps track of past particles in the proposal step.*

□ **Maximum likelihood** – If we don't know the local conditional distributions, we can learn them using maximum likelihood.

$$\max_{\theta} \prod_{x \in \mathcal{D}_{\text{train}}} p(X = x; \theta)$$

□ **Laplace smoothing** – For each distribution  $d$  and partial assignment  $(x_{\text{Parents}(i)}, x_i)$ , add  $\lambda$  to count $_d(x_{\text{Parents}(i)}, x_i)$ , then normalize to get probability estimates.

□ **Algorithm** – The Expectation-Maximization (EM) algorithm gives an efficient method at estimating the parameter  $\theta$  through maximum likelihood estimation by repeatedly constructing a lower-bound on the likelihood (E-step) and optimizing that lower bound (M-step) as follows:

- **E-step:** Evaluate the posterior probability  $q(h)$  that each data point  $e$  came from a particular cluster  $h$  as follows:

$$q(h) = P(H = h | E = e; \theta)$$

- **M-step:** Use the posterior probabilities  $q(h)$  as cluster specific weights on data points  $e$  to determine  $\theta$  through maximum likelihood.

## 4 Logic-based models

### 4.1 Basics

□ **Syntax of propositional logic** – By noting  $f, g$  formulas, and  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$  connectives, we can write the following logical expressions:

Name	Symbol	Meaning	Illustration
Affirmation	$f$	$f$	
Negation	$\neg f$	not $f$	
Conjunction	$f \wedge g$	$f$ and $g$	
Disjunction	$f \vee g$	$f$ or $g$	
Implication	$f \rightarrow g$	if $f$ then $g$	
Biconditional	$f \leftrightarrow g$	$f$ , that is to say $g$	

*Remark: formulas can be built up recursively out of these connectives.*

□ **Model** – A model  $w$  denotes an assignment of binary weights to propositional symbols.

*Example: the set of truth values  $w = \{A : 0, B : 1, C : 0\}$  is one possible model to the propositional symbols  $A, B$  and  $C$ .*

□ **Interpretation function** – The interpretation function  $\mathcal{I}(f, w)$  outputs whether model  $w$  satisfies formula  $f$ :

$$\mathcal{I}(f, w) \in \{0, 1\}$$

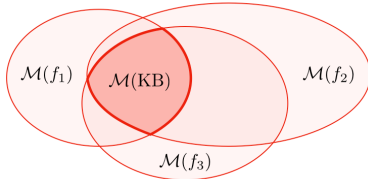
□ **Set of models** –  $\mathcal{M}(f)$  denotes the set of models  $w$  that satisfy formula  $f$ . Mathematically speaking, we define it as follows:

$$\forall w \in \mathcal{M}(f), \mathcal{I}(f,w) = 1$$

## 4.2 Knowledge base

□ **Definition** – The knowledge base KB is the conjunction of all formulas that have been considered so far. The set of models of the knowledge base is the intersection of the set of models that satisfy each formula. In other words:

$$\mathcal{M}(\text{KB}) = \bigcap_{f \in \text{KB}} \mathcal{M}(f)$$



□ **Probabilistic interpretation** – The probability that query  $f$  is evaluated to 1 can be seen as the proportion of models  $w$  of the knowledge base KB that satisfy  $f$ , i.e.:

$$P(f|\text{KB}) = \frac{\sum_{w \in \mathcal{M}(\text{KB}) \cap \mathcal{M}(f)} P(W = w)}{\sum_{w \in \mathcal{M}(\text{KB})} P(W = w)}$$

□ **Satisfiability** – The knowledge base KB is said to be satisfiable if at least one model  $w$  satisfies all its constraints. In other words:

$$\text{KB satisfiable} \iff \mathcal{M}(\text{KB}) \neq \emptyset$$

*Remark:  $\mathcal{M}(\text{KB})$  denotes the set of models compatible with all the constraints of the knowledge base.*

□ **Relation between formulas and knowledge base** – We define the following properties between the knowledge base KB and a new formula  $f$ :

Name	Mathematical formulation	Illustration	Notes
KB entails $f$	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \mathcal{M}(\text{KB})$		- $f$ does not bring any new information - Also written $\text{KB} \models f$
KB contradicts $f$	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \emptyset$		- No model satisfies the constraints after adding $f$ Equivalent to $\text{KB} \models \neg f$
$f$ contingent to KB	$\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \neq \emptyset$ and $\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \neq \mathcal{M}(\text{KB})$		- $f$ does not contradict KB - $f$ adds a non-trivial amount of information to KB

□ **Model checking** – A model checking algorithm takes as input a knowledge base KB and outputs whether it is satisfiable or not.

*Remark: popular model checking algorithms include DPLL and WalkSat.*

□ **Inference rule** – An inference rule of premises  $f_1, \dots, f_k$  and conclusion  $g$  is written:

$$\frac{f_1, \dots, f_k}{g}$$

□ **Forward inference algorithm** – From a set of inference rules Rules, this algorithm goes through all possible  $f_1, \dots, f_k$  and adds  $g$  to the knowledge base KB if a matching rule exists. This process is repeated until no more additions can be made to KB.

□ **Derivation** – We say that KB derives  $f$  (written  $\text{KB} \vdash f$ ) with rules Rules if  $f$  already is in KB or gets added during the forward inference algorithm using the set of rules Rules.

□ **Properties of inference rules** – A set of inference rules Rules can have the following properties:

Name	Mathematical formulation	Notes
Soundness	$\{f : \text{KB} \vdash f\} \subseteq \{f : \text{KB} \models f\}$	- Inferred formulas are entailed by KB - Can be checked one rule at a time - "Nothing but the truth"
Completeness	$\{f : \text{KB} \vdash f\} \supseteq \{f : \text{KB} \models f\}$	- Formulas entailing KB are either already in the knowledge base or inferred from it - "The whole truth"

### 4.3 Propositional logic

In this section, we will go through logic-based models that use logical formulas and inference rules. The idea here is to balance expressivity and computational efficiency.

□ **Horn clause** – By noting  $p_1, \dots, p_k$  and  $q$  propositional symbols, a Horn clause has the form:

$$(p_1 \wedge \dots \wedge p_k) \rightarrow q$$

*Remark: when  $q = \text{false}$ , it is called a "goal clause", otherwise we denote it as a "definite clause".*

□ **Modus ponens inference rule** – For propositional symbols  $f_1, \dots, f_k$  and  $p$ , the modus ponens rule is written:

$$\frac{f_1, \dots, f_k, (f_1 \wedge \dots \wedge f_k) \rightarrow p}{p}$$

*Remark: it takes linear time to apply this rule, as each application generate a clause that contains a single propositional symbol.*

□ **Completeness** – Modus ponens is complete with respect to Horn clauses if we suppose that KB contains only Horn clauses and  $p$  is an entailed propositional symbol. Applying modus ponens will then derive  $p$ .

□ **Conjunctive normal form** – A conjunctive normal form (CNF) formula is a conjunction of clauses, where each clause is a disjunction of atomic formulas.

*Remark: in other words, CNFs are  $\wedge$  of  $\vee$ .*

□ **Equivalent representation** – Every formula in propositional logic can be written into an equivalent CNF formula. The table below presents general conversion properties:

Rule name		Initial	Converted
Eliminate	$\leftrightarrow$	$f \leftrightarrow g$	$(f \rightarrow g) \wedge (g \rightarrow f)$
	$\rightarrow$	$f \rightarrow g$	$\neg f \vee g$
	$\neg\neg$	$\neg\neg f$	$f$
Distribute	$\neg$ over $\wedge$	$\neg(f \wedge g)$	$\neg f \vee \neg g$
	$\neg$ over $\vee$	$\neg(f \vee g)$	$\neg f \wedge \neg g$
	$\vee$ over $\wedge$	$f \vee (g \wedge h)$	$(f \vee g) \wedge (f \vee h)$

□ **Resolution inference rule** – For propositional symbols  $f_1, \dots, f_n$ , and  $g_1, \dots, g_m$  as well as  $p$ , the resolution rule is written:

$$\frac{f_1 \vee \dots \vee f_n \vee p, \neg p \vee g_1 \vee \dots \vee g_m}{f_1 \vee \dots \vee f_n \vee g_1 \vee \dots \vee g_m}$$

*Remark: it can take exponential time to apply this rule, as each application generates a clause that has a subset of the propositional symbols.*

□ **Resolution-based inference** – The resolution-based inference algorithm follows the following steps:

- Step 1: Convert all formulas into CNF

- Step 2: Repeatedly apply resolution rule
- Step 3: Return unsatisfiable if and only if False is derived

### 4.4 First-order logic

The idea here is that variables yield compact knowledge representations.

□ **Model** – A model  $w$  in first-order logic maps:

- constant symbols to objects
- predicate symbols to tuple of objects

□ **Horn clause** – By noting  $x_1, \dots, x_n$  variables and  $a_1, \dots, a_k, b$  atomic formulas, the first-order logic version of a horn clause has the form:

$$\forall x_1, \dots, \forall x_n, (a_1 \wedge \dots \wedge a_k) \rightarrow b$$

□ **Substitution** – A substitution  $\theta$  maps variables to terms and  $\text{Subst}(\theta, f)$  denotes the result of substitution  $\theta$  on  $f$ .

□ **Unification** – Unification takes two formulas  $f$  and  $g$  and returns the most general substitution  $\theta$  that makes them equal:

$$\text{Unify}[f, g] = \theta \quad \text{s.t.} \quad \text{Subst}[\theta, f] = \text{Subst}[\theta, g]$$

*Note: Unify[f, g] returns Fail if no such  $\theta$  exists.*

□ **Modus ponens** – By noting  $x_1, \dots, x_n$  variables,  $a_1, \dots, a_k$  and  $a'_1, \dots, a'_k$  atomic formulas and by calling  $\theta = \text{Unify}(a'_1 \wedge \dots \wedge a'_k, a_1 \wedge \dots \wedge a_k)$  the first-order logic version of modus ponens can be written:

$$\frac{a'_1, \dots, a'_k \quad \forall x_1, \dots, \forall x_n (a_1 \wedge \dots \wedge a_k) \rightarrow b}{\text{Subst}[\theta, b]}$$

□ **Completeness** – Modus ponens is complete for first-order logic with only Horn clauses.

□ **Resolution rule** – By noting  $f_1, \dots, f_n, g_1, \dots, g_m, p, q$  formulas and by calling  $\theta = \text{Unify}(p, q)$ , the first-order logic version of the resolution rule can be written:

$$\frac{f_1 \vee \dots \vee f_n \vee p, \neg q \vee g_1 \vee \dots \vee g_m}{\text{Subst}[\theta, f_1 \vee \dots \vee f_n \vee g_1 \vee \dots \vee g_m]}$$

□ **Semi-decidability** – First-order logic, even restricted to only Horn clauses, is semi-decidable.

- if  $\text{KB} \models f$ , forward inference on complete inference rules will prove  $f$  in finite time
- if  $\text{KB} \not\models f$ , no algorithm can show this in finite time

# Super VIP Cheatsheet: Machine Learning

Afshine AMIDI and Shervine AMIDI

October 6, 2018

## Contents

<b>1 Supervised Learning</b>	<b>2</b>	<b>4 Machine Learning Tips and Tricks</b>	<b>10</b>
1.1 Introduction to Supervised Learning . . . . .	2	4.1 Metrics . . . . .	10
1.2 Notations and general concepts . . . . .	2	4.1.1 Classification . . . . .	10
1.3 Linear models . . . . .	2	4.1.2 Regression . . . . .	10
1.3.1 Linear regression . . . . .	2	4.2 Model selection . . . . .	11
1.3.2 Classification and logistic regression . . . . .	3	4.3 Diagnostics . . . . .	11
1.3.3 Generalized Linear Models . . . . .	3	<b>5 Refreshers</b>	<b>12</b>
1.4 Support Vector Machines . . . . .	3	5.1 Probabilities and Statistics . . . . .	12
1.5 Generative Learning . . . . .	4	5.1.1 Introduction to Probability and Combinatorics . . . . .	12
1.5.1 Gaussian Discriminant Analysis . . . . .	4	5.1.2 Conditional Probability . . . . .	12
1.5.2 Naive Bayes . . . . .	4	5.1.3 Random Variables . . . . .	13
1.6 Tree-based and ensemble methods . . . . .	4	5.1.4 Jointly Distributed Random Variables . . . . .	13
1.7 Other non-parametric approaches . . . . .	4	5.1.5 Parameter estimation . . . . .	14
1.8 Learning Theory . . . . .	5	5.2 Linear Algebra and Calculus . . . . .	14
<b>2 Unsupervised Learning</b>	<b>6</b>	5.2.1 General notations . . . . .	14
2.1 Introduction to Unsupervised Learning . . . . .	6	5.2.2 Matrix operations . . . . .	15
2.2 Clustering . . . . .	6	5.2.3 Matrix properties . . . . .	15
2.2.1 Expectation-Maximization . . . . .	6	5.2.4 Matrix calculus . . . . .	16
2.2.2 $k$ -means clustering . . . . .	6		
2.2.3 Hierarchical clustering . . . . .	6		
2.2.4 Clustering assessment metrics . . . . .	6		
2.3 Dimension reduction . . . . .	7		
2.3.1 Principal component analysis . . . . .	7		
2.3.2 Independent component analysis . . . . .	7		
<b>3 Deep Learning</b>	<b>8</b>		
3.1 Neural Networks . . . . .	8		
3.2 Convolutional Neural Networks . . . . .	8		
3.3 Recurrent Neural Networks . . . . .	8		
3.4 Reinforcement Learning and Control . . . . .	9		



# 1 Supervised Learning

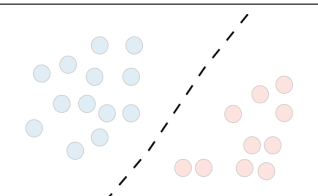
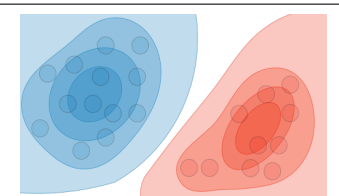
## 1.1 Introduction to Supervised Learning

Given a set of data points  $\{x^{(1)}, \dots, x^{(m)}\}$  associated to a set of outcomes  $\{y^{(1)}, \dots, y^{(m)}\}$ , we want to build a classifier that learns how to predict  $y$  from  $x$ .

□ **Type of prediction** – The different types of predictive models are summed up in the table below:

	<b>Regression</b>	<b>Classifier</b>
<b>Outcome</b>	Continuous	Class
<b>Examples</b>	Linear regression	Logistic regression, SVM, Naive Bayes

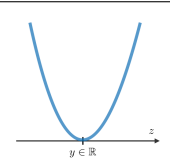
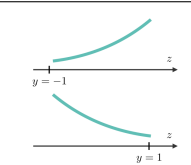
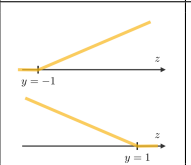
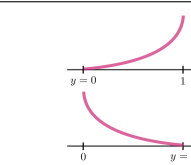
□ **Type of model** – The different models are summed up in the table below:

	<b>Discriminative model</b>	<b>Generative model</b>
<b>Goal</b>	Directly estimate $P(y x)$	Estimate $P(x y)$ to deduce $P(y x)$
<b>What's learned</b>	Decision boundary	Probability distributions of the data
<b>Illustration</b>		
<b>Examples</b>	Regressions, SVMs	GDA, Naive Bayes

## 1.2 Notations and general concepts

□ **Hypothesis** – The hypothesis is noted  $h_\theta$  and is the model that we choose. For a given input data  $x^{(i)}$ , the model prediction output is  $h_\theta(x^{(i)})$ .

□ **Loss function** – A loss function is a function  $L : (z, y) \in \mathbb{R} \times Y \mapsto L(z, y) \in \mathbb{R}$  that takes as inputs the predicted value  $z$  corresponding to the real data value  $y$  and outputs how different they are. The common loss functions are summed up in the table below:

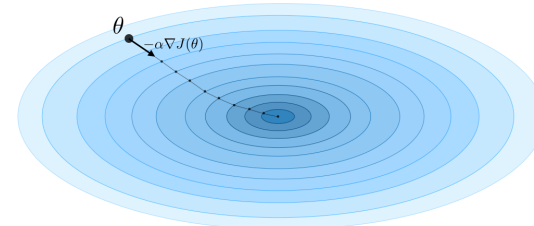
<b>Least squared</b>	<b>Logistic</b>	<b>Hinge</b>	<b>Cross-entropy</b>
$\frac{1}{2}(y - z)^2$	$\log(1 + \exp(-yz))$	$\max(0, 1 - yz)$	$-[y \log(z) + (1 - y) \log(1 - z)]$
			
Linear regression	Logistic regression	SVM	Neural Network

□ **Cost function** – The cost function  $J$  is commonly used to assess the performance of a model, and is defined with the loss function  $L$  as follows:

$$J(\theta) = \sum_{i=1}^m L(h_\theta(x^{(i)}), y^{(i)})$$

□ **Gradient descent** – By noting  $\alpha \in \mathbb{R}$  the learning rate, the update rule for gradient descent is expressed with the learning rate and the cost function  $J$  as follows:

$$\theta \leftarrow \theta - \alpha \nabla J(\theta)$$



*Remark: Stochastic gradient descent (SGD) is updating the parameter based on each training example, and batch gradient descent is on a batch of training examples.*

□ **Likelihood** – The likelihood of a model  $L(\theta)$  given parameters  $\theta$  is used to find the optimal parameters  $\theta$  through maximizing the likelihood. In practice, we use the log-likelihood  $\ell(\theta) = \log(L(\theta))$  which is easier to optimize. We have:

$$\theta^{\text{opt}} = \arg \max_{\theta} L(\theta)$$

□ **Newton's algorithm** – The Newton's algorithm is a numerical method that finds  $\theta$  such that  $\ell'(\theta) = 0$ . Its update rule is as follows:

$$\theta \leftarrow \theta - \frac{\ell'(\theta)}{\ell''(\theta)}$$

*Remark: the multidimensional generalization, also known as the Newton-Raphson method, has the following update rule:*

$$\theta \leftarrow \theta - (\nabla_{\theta}^2 \ell(\theta))^{-1} \nabla_{\theta} \ell(\theta)$$

## 1.3 Linear models

### 1.3.1 Linear regression

We assume here that  $y|x; \theta \sim \mathcal{N}(\mu, \sigma^2)$

□ **Normal equations** – By noting  $X$  the matrix design, the value of  $\theta$  that minimizes the cost function is a closed-form solution such that:

$$\theta = (X^T X)^{-1} X^T y$$

□ **LMS algorithm** – By noting  $\alpha$  the learning rate, the update rule of the Least Mean Squares (LMS) algorithm for a training set of  $m$  data points, which is also known as the Widrow-Hoff learning rule, is as follows:

$$\forall j, \theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^m [y^{(i)} - h_{\theta}(x^{(i)})] x_j^{(i)}$$

*Remark: the update rule is a particular case of the gradient ascent.*

□ **LWR** – Locally Weighted Regression, also known as LWR, is a variant of linear regression that weights each training example in its cost function by  $w^{(i)}(x)$ , which is defined with parameter  $\tau \in \mathbb{R}$  as:

$$w^{(i)}(x) = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

### 1.3.2 Classification and logistic regression

□ **Sigmoid function** – The sigmoid function  $g$ , also known as the logistic function, is defined as follows:

$$\forall z \in \mathbb{R}, g(z) = \frac{1}{1 + e^{-z}} \in ]0,1[$$

□ **Logistic regression** – We assume here that  $y|x; \theta \sim \text{Bernoulli}(\phi)$ . We have the following form:

$$\phi = p(y = 1|x; \theta) = \frac{1}{1 + \exp(-\theta^T x)} = g(\theta^T x)$$

*Remark: there is no closed form solution for the case of logistic regressions.*

□ **Softmax regression** – A softmax regression, also called a multiclass logistic regression, is used to generalize logistic regression when there are more than 2 outcome classes. By convention, we set  $\theta_K = 0$ , which makes the Bernoulli parameter  $\phi_i$  of each class  $i$  equal to:

$$\phi_i = \frac{\exp(\theta_i^T x)}{\sum_{j=1}^K \exp(\theta_j^T x)}$$

### 1.3.3 Generalized Linear Models

□ **Exponential family** – A class of distributions is said to be in the exponential family if it can be written in terms of a natural parameter, also called the canonical parameter or link function,  $\eta$ , a sufficient statistic  $T(y)$  and a log-partition function  $a(\eta)$  as follows:

$$p(y; \eta) = b(y) \exp(\eta T(y) - a(\eta))$$

*Remark: we will often have  $T(y) = y$ . Also,  $\exp(-a(\eta))$  can be seen as a normalization parameter that will make sure that the probabilities sum to one.*

Here are the most common exponential distributions summed up in the following table:

Distribution	$\eta$	$T(y)$	$a(\eta)$	$b(y)$
Bernoulli	$\log\left(\frac{\phi}{1-\phi}\right)$	$y$	$\log(1 + \exp(\eta))$	1
Gaussian	$\mu$	$y$	$\frac{\eta^2}{2}$	$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{y^2}{2}\right)$
Poisson	$\log(\lambda)$	$y$	$e^{\eta}$	$\frac{1}{y!}$
Geometric	$\log(1 - \phi)$	$y$	$\log\left(\frac{e^{\eta}}{1 - e^{\eta}}\right)$	1

□ **Assumptions of GLMs** – Generalized Linear Models (GLM) aim at predicting a random variable  $y$  as a function of  $x \in \mathbb{R}^{n+1}$  and rely on the following 3 assumptions:

- (1)  $y|x; \theta \sim \text{ExpFamily}(\eta)$       (2)  $h_{\theta}(x) = E[y|x; \theta]$       (3)  $\eta = \theta^T x$

*Remark: ordinary least squares and logistic regression are special cases of generalized linear models.*

### 1.4 Support Vector Machines

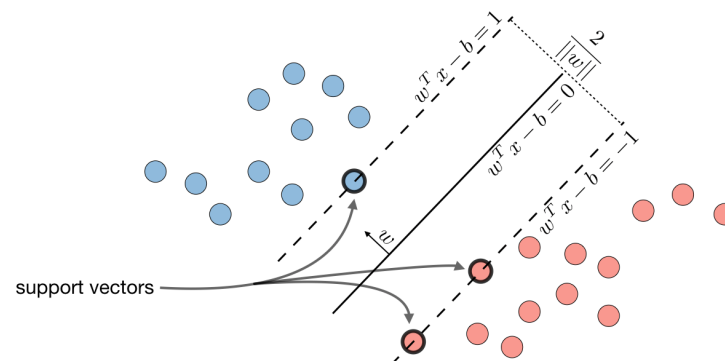
The goal of support vector machines is to find the line that maximizes the minimum distance to the line.

□ **Optimal margin classifier** – The optimal margin classifier  $h$  is such that:

$$h(x) = \text{sign}(w^T x - b)$$

where  $(w, b) \in \mathbb{R}^n \times \mathbb{R}$  is the solution of the following optimization problem:

$$\min \frac{1}{2} \|w\|^2 \quad \text{such that} \quad y^{(i)}(w^T x^{(i)} - b) \geq 1$$



*Remark: the line is defined as  $w^T x - b = 0$ .*

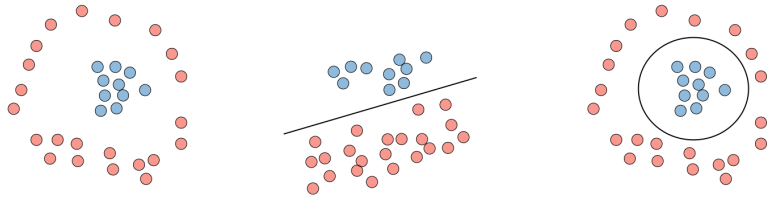
□ **Hinge loss** – The hinge loss is used in the setting of SVMs and is defined as follows:

$$L(z, y) = [1 - yz]_+ = \max(0, 1 - yz)$$

□ **Kernel** – Given a feature mapping  $\phi$ , we define the kernel  $K$  to be defined as:

$$K(x, z) = \phi(x)^T \phi(z)$$

In practice, the kernel  $K$  defined by  $K(x, z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right)$  is called the Gaussian kernel and is commonly used.



Non-linear separability  $\longrightarrow$  Use of a kernel mapping  $\phi$   $\longrightarrow$  Decision boundary in the original space

*Remark: we say that we use the "kernel trick" to compute the cost function using the kernel because we actually don't need to know the explicit mapping  $\phi$ , which is often very complicated. Instead, only the values  $K(x, z)$  are needed.*

□ **Lagrangian** – We define the Lagrangian  $\mathcal{L}(w, b)$  as follows:

$$\mathcal{L}(w, b) = f(w) + \sum_{i=1}^l \beta_i h_i(w)$$

*Remark: the coefficients  $\beta_i$  are called the Lagrange multipliers.*

### 1.5 Generative Learning

A generative model first tries to learn how the data is generated by estimating  $P(x|y)$ , which we can then use to estimate  $P(y|x)$  by using Bayes' rule.

#### 1.5.1 Gaussian Discriminant Analysis

□ **Setting** – The Gaussian Discriminant Analysis assumes that  $y$  and  $x|y = 0$  and  $x|y = 1$  are such that:

$$y \sim \text{Bernoulli}(\phi)$$

$$x|y = 0 \sim \mathcal{N}(\mu_0, \Sigma) \quad \text{and} \quad x|y = 1 \sim \mathcal{N}(\mu_1, \Sigma)$$

□ **Estimation** – The following table sums up the estimates that we find when maximizing the likelihood:

$\hat{\phi}$	$\hat{\mu}_j \quad (j = 0, 1)$	$\hat{\Sigma}$
$\frac{1}{m} \sum_{i=1}^m 1_{\{y^{(i)}=1\}}$	$\frac{\sum_{i=1}^m 1_{\{y^{(i)}=j\}} x^{(i)}}{\sum_{i=1}^m 1_{\{y^{(i)}=j\}}}$	$\frac{1}{m} \sum_{i=1}^m (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T$

### 1.5.2 Naive Bayes

□ **Assumption** – The Naive Bayes model supposes that the features of each data point are all independent:

$$P(x|y) = P(x_1, x_2, \dots | y) = P(x_1|y)P(x_2|y)\dots = \prod_{i=1}^n P(x_i|y)$$

□ **Solutions** – Maximizing the log-likelihood gives the following solutions, with  $k \in \{0, 1\}$ ,  $l \in \llbracket 1, L \rrbracket$

$$P(y = k) = \frac{1}{m} \times \#\{j|y^{(j)} = k\}$$

$$P(x_i = l|y = k) = \frac{\#\{j|y^{(j)} = k \text{ and } x_i^{(j)} = l\}}{\#\{j|y^{(j)} = k\}}$$

*Remark: Naive Bayes is widely used for text classification and spam detection.*

### 1.6 Tree-based and ensemble methods

These methods can be used for both regression and classification problems.

□ **CART** – Classification and Regression Trees (CART), commonly known as decision trees, can be represented as binary trees. They have the advantage to be very interpretable.

□ **Random forest** – It is a tree-based technique that uses a high number of decision trees built out of randomly selected sets of features. Contrary to the simple decision tree, it is highly uninterpretable but its generally good performance makes it a popular algorithm.

*Remark: random forests are a type of ensemble methods.*

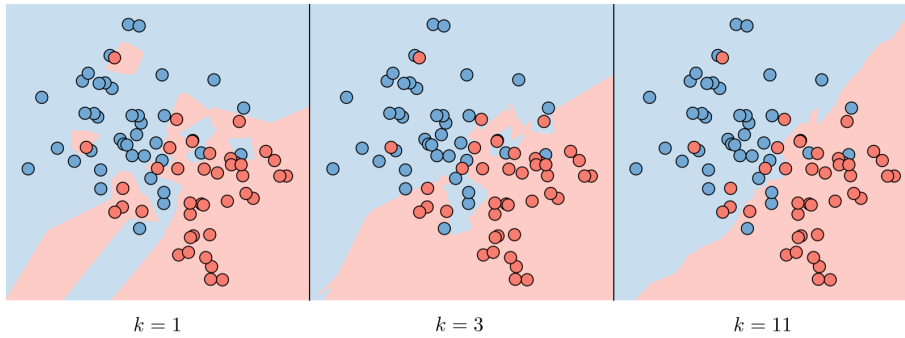
□ **Boosting** – The idea of boosting methods is to combine several weak learners to form a stronger one. The main ones are summed up in the table below:

Adaptive boosting	Gradient boosting
- High weights are put on errors to improve at the next boosting step - Known as Adaboost	- Weak learners trained on remaining errors

### 1.7 Other non-parametric approaches

□ **k-nearest neighbors** – The  $k$ -nearest neighbors algorithm, commonly known as  $k$ -NN, is a non-parametric approach where the response of a data point is determined by the nature of its  $k$  neighbors from the training set. It can be used in both classification and regression settings.

*Remark: The higher the parameter  $k$ , the higher the bias, and the lower the parameter  $k$ , the higher the variance.*



$$\exists h \in \mathcal{H}, \quad \forall i \in [1, d], \quad h(x^{(i)}) = y^{(i)}$$

□ **Upper bound theorem** – Let  $\mathcal{H}$  be a finite hypothesis class such that  $|\mathcal{H}| = k$  and let  $\delta$  and the sample size  $m$  be fixed. Then, with probability of at least  $1 - \delta$ , we have:

$$\epsilon(\hat{h}) \leq \left( \min_{h \in \mathcal{H}} \epsilon(h) \right) + 2\sqrt{\frac{1}{2m} \log \left( \frac{2k}{\delta} \right)}$$

□ **VC dimension** – The Vapnik-Chervonenkis (VC) dimension of a given infinite hypothesis class  $\mathcal{H}$ , noted  $\text{VC}(\mathcal{H})$  is the size of the largest set that is shattered by  $\mathcal{H}$ .

*Remark: the VC dimension of  $\mathcal{H} = \{\text{set of linear classifiers in 2 dimensions}\}$  is 3.*



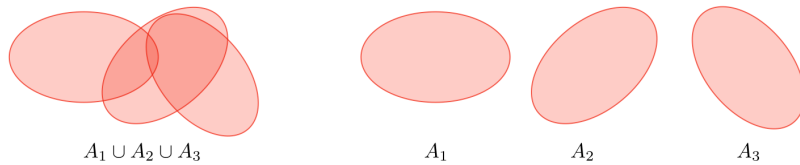
□ **Theorem (Vapnik)** – Let  $\mathcal{H}$  be given, with  $\text{VC}(\mathcal{H}) = d$  and  $m$  the number of training examples. With probability at least  $1 - \delta$ , we have:

$$\epsilon(\hat{h}) \leq \left( \min_{h \in \mathcal{H}} \epsilon(h) \right) + O \left( \sqrt{\frac{d}{m} \log \left( \frac{m}{d} \right)} + \frac{1}{m} \log \left( \frac{1}{\delta} \right) \right)$$

### 1.8 Learning Theory

□ **Union bound** – Let  $A_1, \dots, A_k$  be  $k$  events. We have:

$$P(A_1 \cup \dots \cup A_k) \leq P(A_1) + \dots + P(A_k)$$



□ **Hoeffding inequality** – Let  $Z_1, \dots, Z_m$  be  $m$  iid variables drawn from a Bernoulli distribution of parameter  $\phi$ . Let  $\hat{\phi}$  be their sample mean and  $\gamma > 0$  fixed. We have:

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2 \exp(-2\gamma^2 m)$$

*Remark: this inequality is also known as the Chernoff bound.*

□ **Training error** – For a given classifier  $h$ , we define the training error  $\hat{\epsilon}(h)$ , also known as the empirical risk or empirical error, to be as follows:

$$\hat{\epsilon}(h) = \frac{1}{m} \sum_{i=1}^m 1_{\{h(x^{(i)}) \neq y^{(i)}\}}$$

□ **Probably Approximately Correct (PAC)** – PAC is a framework under which numerous results on learning theory were proved, and has the following set of assumptions:

- the training and testing sets follow the same distribution
- the training examples are drawn independently

□ **Shattering** – Given a set  $S = \{x^{(1)}, \dots, x^{(d)}\}$ , and a set of classifiers  $\mathcal{H}$ , we say that  $\mathcal{H}$  shatters  $S$  if for any set of labels  $\{y^{(1)}, \dots, y^{(d)}\}$ , we have:

## 2 Unsupervised Learning

### 2.1 Introduction to Unsupervised Learning

□ **Motivation** – The goal of unsupervised learning is to find hidden patterns in unlabeled data  $\{x^{(1)}, \dots, x^{(m)}\}$ .

□ **Jensen's inequality** – Let  $f$  be a convex function and  $X$  a random variable. We have the following inequality:

$$E[f(X)] \geq f(E[X])$$

### 2.2 Clustering

#### 2.2.1 Expectation-Maximization

□ **Latent variables** – Latent variables are hidden/unobserved variables that make estimation problems difficult, and are often denoted  $z$ . Here are the most common settings where there are latent variables:

Setting	Latent variable $z$	$x z$	Comments
Mixture of $k$ Gaussians	Multinomial( $\phi$ )	$\mathcal{N}(\mu_j, \Sigma_j)$	$\mu_j \in \mathbb{R}^n, \phi \in \mathbb{R}^k$
Factor analysis	$\mathcal{N}(0, I)$	$\mathcal{N}(\mu + \Lambda z, \psi)$	$\mu_j \in \mathbb{R}^n$

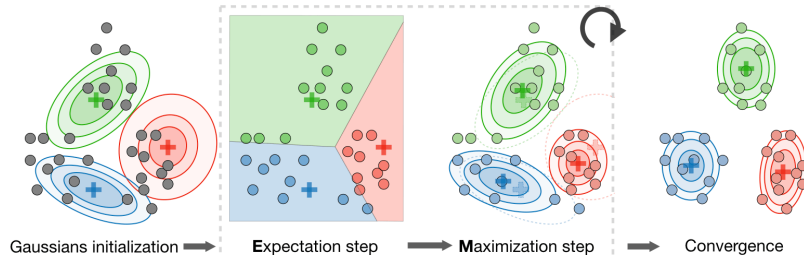
□ **Algorithm** – The Expectation-Maximization (EM) algorithm gives an efficient method at estimating the parameter  $\theta$  through maximum likelihood estimation by repeatedly constructing a lower-bound on the likelihood (E-step) and optimizing that lower bound (M-step) as follows:

- **E-step:** Evaluate the posterior probability  $Q_i(z^{(i)})$  that each data point  $x^{(i)}$  came from a particular cluster  $z^{(i)}$  as follows:

$$Q_i(z^{(i)}) = P(z^{(i)}|x^{(i)}; \theta)$$

- **M-step:** Use the posterior probabilities  $Q_i(z^{(i)})$  as cluster specific weights on data points  $x^{(i)}$  to separately re-estimate each cluster model as follows:

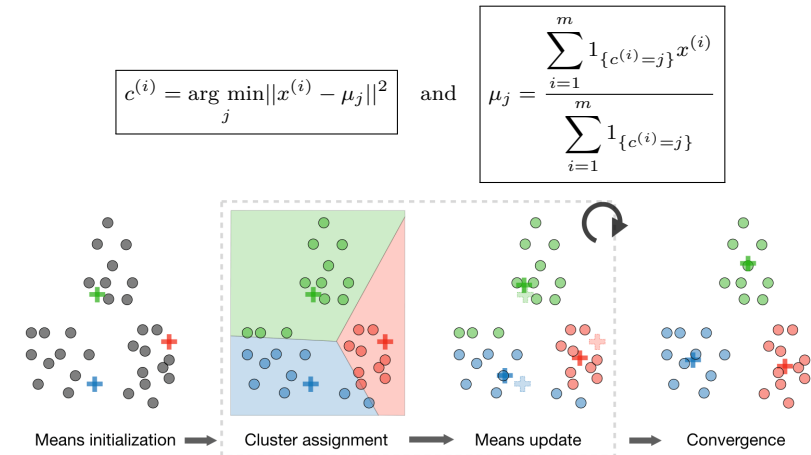
$$\theta_i = \operatorname{argmax}_{\theta} \sum_i \int_{z^{(i)}} Q_i(z^{(i)}) \log \left( \frac{P(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right) dz^{(i)}$$



#### 2.2.2 $k$ -means clustering

We note  $c^{(i)}$  the cluster of data point  $i$  and  $\mu_j$  the center of cluster  $j$ .

□ **Algorithm** – After randomly initializing the cluster centroids  $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^n$ , the  $k$ -means algorithm repeats the following step until convergence:



□ **Distortion function** – In order to see if the algorithm converges, we look at the distortion function defined as follows:

$$J(c, \mu) = \sum_{i=1}^m \|x^{(i)} - \mu_{c^{(i)}}\|^2$$

#### 2.2.3 Hierarchical clustering

□ **Algorithm** – It is a clustering algorithm with an agglomerative hierarchical approach that build nested clusters in a successive manner.

□ **Types** – There are different sorts of hierarchical clustering algorithms that aims at optimizing different objective functions, which is summed up in the table below:

Ward linkage	Average linkage	Complete linkage
Minimize within cluster distance	Minimize average distance between cluster pairs	Minimize maximum distance of between cluster pairs

#### 2.2.4 Clustering assessment metrics

In an unsupervised learning setting, it is often hard to assess the performance of a model since we don't have the ground truth labels as was the case in the supervised learning setting.

□ **Silhouette coefficient** – By noting  $a$  and  $b$  the mean distance between a sample and all other points in the same class, and between a sample and all other points in the next nearest cluster, the silhouette coefficient  $s$  for a single sample is defined as follows:

$$s = \frac{b - a}{\max(a, b)}$$

□ **Calinski-Harabaz index** – By noting  $k$  the number of clusters,  $B_k$  and  $W_k$  the between and within-clustering dispersion matrices respectively defined as

$$B_k = \sum_{j=1}^k n_{c^{(j)}} (\mu_{c^{(j)}} - \mu)(\mu_{c^{(j)}} - \mu)^T, \quad W_k = \sum_{i=1}^m (x^{(i)} - \mu_{c^{(i)}})(x^{(i)} - \mu_{c^{(i)}})^T$$

the Calinski-Harabaz index  $s(k)$  indicates how well a clustering model defines its clusters, such that the higher the score, the more dense and well separated the clusters are. It is defined as follows:

$$s(k) = \frac{\text{Tr}(B_k)}{\text{Tr}(W_k)} \times \frac{N - k}{k - 1}$$

### 2.3 Dimension reduction

#### 2.3.1 Principal component analysis

It is a dimension reduction technique that finds the variance maximizing directions onto which to project the data.

□ **Eigenvalue, eigenvector** – Given a matrix  $A \in \mathbb{R}^{n \times n}$ ,  $\lambda$  is said to be an eigenvalue of  $A$  if there exists a vector  $z \in \mathbb{R}^n \setminus \{0\}$ , called eigenvector, such that we have:

$$Az = \lambda z$$

□ **Spectral theorem** – Let  $A \in \mathbb{R}^{n \times n}$ . If  $A$  is symmetric, then  $A$  is diagonalizable by a real orthogonal matrix  $U \in \mathbb{R}^{n \times n}$ . By noting  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ , we have:

$$\exists \Lambda \text{ diagonal, } A = U\Lambda U^T$$

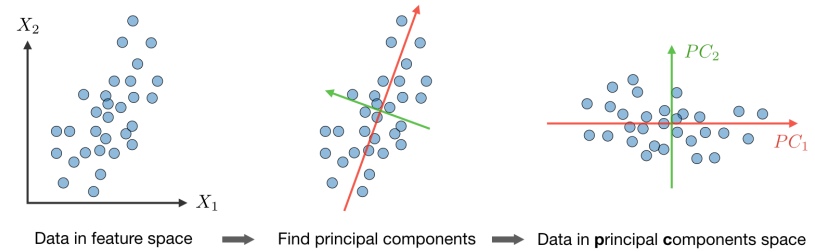
*Remark: the eigenvector associated with the largest eigenvalue is called principal eigenvector of matrix  $A$ .*

□ **Algorithm** – The Principal Component Analysis (PCA) procedure is a dimension reduction technique that projects the data on  $k$  dimensions by maximizing the variance of the data as follows:

- Step 1: Normalize the data to have a mean of 0 and standard deviation of 1.

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j} \quad \text{where} \quad \mu_j = \frac{1}{m} \sum_{i=1}^m x_j^{(i)} \quad \text{and} \quad \sigma_j^2 = \frac{1}{m} \sum_{i=1}^m (x_j^{(i)} - \mu_j)^2$$

- Step 2: Compute  $\Sigma = \frac{1}{m} \sum_{i=1}^m x^{(i)} x^{(i)T} \in \mathbb{R}^{n \times n}$ , which is symmetric with real eigenvalues.
- Step 3: Compute  $u_1, \dots, u_k \in \mathbb{R}^n$  the  $k$  orthogonal principal eigenvectors of  $\Sigma$ , i.e. the orthogonal eigenvectors of the  $k$  largest eigenvalues.
- Step 4: Project the data on  $\text{span}_{\mathbb{R}}(u_1, \dots, u_k)$ . This procedure maximizes the variance among all  $k$ -dimensional spaces.



#### 2.3.2 Independent component analysis

It is a technique meant to find the underlying generating sources.

□ **Assumptions** – We assume that our data  $x$  has been generated by the  $n$ -dimensional source vector  $s = (s_1, \dots, s_n)$ , where  $s_i$  are independent random variables, via a mixing and non-singular matrix  $A$  as follows:

$$x = As$$

The goal is to find the unmixing matrix  $W = A^{-1}$  by an update rule.

□ **Bell and Sejnowski ICA algorithm** – This algorithm finds the unmixing matrix  $W$  by following the steps below:

- Write the probability of  $x = As = W^{-1}s$  as:

$$p(x) = \prod_{i=1}^n p_s(w_i^T x) \cdot |W|$$

- Write the log likelihood given our training data  $\{x^{(i)}, i \in [1, m]\}$  and by noting  $g$  the sigmoid function as:

$$l(W) = \sum_{i=1}^m \left( \sum_{j=1}^n \log \left( g'(w_j^T x^{(i)}) \right) + \log |W| \right)$$

Therefore, the stochastic gradient ascent learning rule is such that for each training example  $x^{(i)}$ , we update  $W$  as follows:

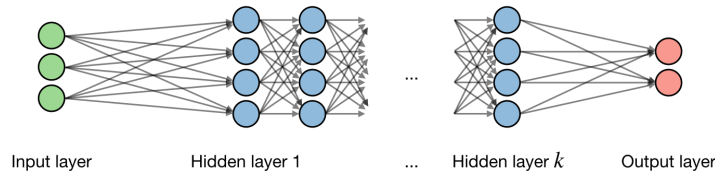
$$W \leftarrow W + \alpha \left( \begin{pmatrix} 1 - 2g(w_1^T x^{(i)}) \\ 1 - 2g(w_2^T x^{(i)}) \\ \vdots \\ 1 - 2g(w_n^T x^{(i)}) \end{pmatrix} x^{(i)T} + (W^T)^{-1} \right)$$

### 3 Deep Learning

#### 3.1 Neural Networks

Neural networks are a class of models that are built with layers. Commonly used types of neural networks include convolutional and recurrent neural networks.

□ **Architecture** – The vocabulary around neural networks architectures is described in the figure below:



By noting  $i$  the  $i^{th}$  layer of the network and  $j$  the  $j^{th}$  hidden unit of the layer, we have:

$$z_j^{[i]} = w_j^{[i]T} x + b_j^{[i]}$$

where we note  $w, b, z$  the weight, bias and output respectively.

□ **Activation function** – Activation functions are used at the end of a hidden unit to introduce non-linear complexities to the model. Here are the most common ones:

Sigmoid	Tanh	ReLU	Leaky ReLU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$

□ **Cross-entropy loss** – In the context of neural networks, the cross-entropy loss  $L(z, y)$  is commonly used and is defined as follows:

$$L(z, y) = - \left[ y \log(z) + (1 - y) \log(1 - z) \right]$$

□ **Learning rate** – The learning rate, often noted  $\eta$ , indicates at which pace the weights get updated. This can be fixed or adaptively changed. The current most popular method is called Adam, which is a method that adapts the learning rate.

□ **Backpropagation** – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to weight  $w$  is computed using chain rule and is of the following form:

$$\frac{\partial L(z, y)}{\partial w} = \frac{\partial L(z, y)}{\partial a} \times \frac{\partial a}{\partial z} \times \frac{\partial z}{\partial w}$$

As a result, the weight is updated as follows:

$$w \leftarrow w - \eta \frac{\partial L(z, y)}{\partial w}$$

□ **Updating weights** – In a neural network, weights are updated as follows:

- **Step 1:** Take a batch of training data.
- **Step 2:** Perform forward propagation to obtain the corresponding loss.
- **Step 3:** Backpropagate the loss to get the gradients.
- **Step 4:** Use the gradients to update the weights of the network.

□ **Dropout** – Dropout is a technique meant at preventing overfitting the training data by dropping out units in a neural network. In practice, neurons are either dropped with probability  $p$  or kept with probability  $1 - p$ .

#### 3.2 Convolutional Neural Networks

□ **Convolutional layer requirement** – By noting  $W$  the input volume size,  $F$  the size of the convolutional layer neurons,  $P$  the amount of zero padding, then the number of neurons  $N$  that fit in a given volume is such that:

$$N = \frac{W - F + 2P}{S} + 1$$

□ **Batch normalization** – It is a step of hyperparameter  $\gamma, \beta$  that normalizes the batch  $\{x_i\}$ . By noting  $\mu_B, \sigma_B^2$  the mean and variance of that we want to correct to the batch, it is done as follows:

$$x_i \leftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

It is usually done after a fully connected/convolutional layer and before a non-linearity layer and aims at allowing higher learning rates and reducing the strong dependence on initialization.

#### 3.3 Recurrent Neural Networks

□ **Types of gates** – Here are the different types of gates that we encounter in a typical recurrent neural network:

Input gate	Forget gate	Output gate	Gate
Write to cell or not?	Erase a cell or not?	Reveal a cell or not?	How much writing?

□ **LSTM** – A long short-term memory (LSTM) network is a type of RNN model that avoids the vanishing gradient problem by adding 'forget' gates.

### 3.4 Reinforcement Learning and Control

The goal of reinforcement learning is for an agent to learn how to evolve in an environment.

□ **Markov decision processes** – A Markov decision process (MDP) is a 5-tuple  $(S, \mathcal{A}, \{P_{sa}\}, \gamma, R)$  where:

- $S$  is the set of states
- $\mathcal{A}$  is the set of actions
- $\{P_{sa}\}$  are the state transition probabilities for  $s \in S$  and  $a \in \mathcal{A}$
- $\gamma \in [0, 1[$  is the discount factor
- $R : S \times \mathcal{A} \rightarrow \mathbb{R}$  or  $R : S \rightarrow \mathbb{R}$  is the reward function that the algorithm wants to maximize

□ **Policy** – A policy  $\pi$  is a function  $\pi : S \rightarrow \mathcal{A}$  that maps states to actions.

*Remark: we say that we execute a given policy  $\pi$  if given a state  $s$  we take the action  $a = \pi(s)$ .*

□ **Value function** – For a given policy  $\pi$  and a given state  $s$ , we define the value function  $V^\pi$  as follows:

$$V^\pi(s) = E \left[ R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots | s_0 = s, \pi \right]$$

□ **Bellman equation** – The optimal Bellman equations characterizes the value function  $V^{\pi^*}$  of the optimal policy  $\pi^*$ :

$$V^{\pi^*}(s) = R(s) + \max_{a \in \mathcal{A}} \gamma \sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s')$$

*Remark: we note that the optimal policy  $\pi^*$  for a given state  $s$  is such that:*

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in S} P_{sa}(s') V^*(s')$$

□ **Value iteration algorithm** – The value iteration algorithm is in two steps:

- We initialize the value:

$$V_0(s) = 0$$

- We iterate the value based on the values before:

$$V_{i+1}(s) = R(s) + \max_{a \in \mathcal{A}} \left[ \sum_{s' \in S} \gamma P_{sa}(s') V_i(s') \right]$$

□ **Maximum likelihood estimate** – The maximum likelihood estimates for the state transition probabilities are as follows:

$$P_{sa}(s') = \frac{\text{\#times took action } a \text{ in state } s \text{ and got to } s'}{\text{\#times took action } a \text{ in state } s}$$

□ **Q-learning** – Q-learning is a model-free estimation of  $Q$ , which is done as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[ R(s, a, s') + \gamma \max_{a'} Q(s', a') - Q(s, a) \right]$$



## 4 Machine Learning Tips and Tricks

### 4.1 Metrics

Given a set of data points  $\{x^{(1)}, \dots, x^{(m)}\}$ , where each  $x^{(i)}$  has  $n$  features, associated to a set of outcomes  $\{y^{(1)}, \dots, y^{(m)}\}$ , we want to assess a given classifier that learns how to predict  $y$  from  $x$ .

#### 4.1.1 Classification

In a context of a binary classification, here are the main metrics that are important to track to assess the performance of the model.

□ **Confusion matrix** – The confusion matrix is used to have a more complete picture when assessing the performance of a model. It is defined as follows:

		Predicted class	
		+	-
Actual class	+	<b>TP</b> True Positives	<b>FN</b> False Negatives Type II error
	-	<b>FP</b> False Positives Type I error	<b>TN</b> True Negatives

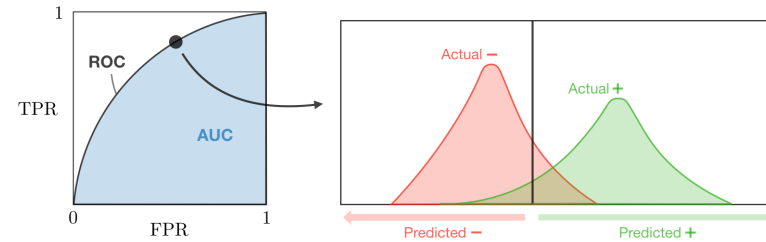
□ **Main metrics** – The following metrics are commonly used to assess the performance of classification models:

Metric	Formula	Interpretation
Accuracy	$\frac{TP + TN}{TP + TN + FP + FN}$	Overall performance of model
Precision	$\frac{TP}{TP + FP}$	How accurate the positive predictions are
Recall Sensitivity	$\frac{TP}{TP + FN}$	Coverage of actual positive sample
Specificity	$\frac{TN}{TN + FP}$	Coverage of actual negative sample
F1 score	$\frac{2TP}{2TP + FP + FN}$	Hybrid metric useful for unbalanced classes

□ **ROC** – The receiver operating curve, also noted ROC, is the plot of TPR versus FPR by varying the threshold. These metrics are summed up in the table below:

Metric	Formula	Equivalent
True Positive Rate TPR	$\frac{TP}{TP + FN}$	Recall, sensitivity
False Positive Rate FPR	$\frac{FP}{TN + FP}$	1-specificity

□ **AUC** – The area under the receiving operating curve, also noted AUC or AUROC, is the area below the ROC as shown in the following figure:



#### 4.1.2 Regression

□ **Basic metrics** – Given a regression model  $f$ , the following metrics are commonly used to assess the performance of the model:

Total sum of squares	Explained sum of squares	Residual sum of squares
$SS_{\text{tot}} = \sum_{i=1}^m (y_i - \bar{y})^2$	$SS_{\text{reg}} = \sum_{i=1}^m (f(x_i) - \bar{y})^2$	$SS_{\text{res}} = \sum_{i=1}^m (y_i - f(x_i))^2$

□ **Coefficient of determination** – The coefficient of determination, often noted  $R^2$  or  $r^2$ , provides a measure of how well the observed outcomes are replicated by the model and is defined as follows:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

□ **Main metrics** – The following metrics are commonly used to assess the performance of regression models, by taking into account the number of variables  $n$  that they take into consideration:

Mallow's Cp	AIC	BIC	Adjusted $R^2$
$\frac{SS_{\text{res}} + 2(n+1)\hat{\sigma}^2}{m}$	$2[(n+2) - \log(L)]$	$\log(m)(n+2) - 2\log(L)$	$1 - \frac{(1-R^2)(m-1)}{m-n-1}$

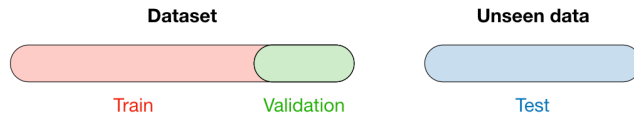
where  $L$  is the likelihood and  $\hat{\sigma}^2$  is an estimate of the variance associated with each response.

### 4.2 Model selection

□ **Vocabulary** – When selecting a model, we distinguish 3 different parts of the data that we have as follows:

Training set	Validation set	Testing set
<ul style="list-style-type: none"> <li>- Model is trained</li> <li>- Usually 80% of the dataset</li> </ul>	<ul style="list-style-type: none"> <li>- Model is assessed</li> <li>- Usually 20% of the dataset</li> <li>- Also called hold-out or development set</li> </ul>	<ul style="list-style-type: none"> <li>- Model gives predictions</li> <li>- Unseen data</li> </ul>

Once the model has been chosen, it is trained on the entire dataset and tested on the unseen test set. These are represented in the figure below:



□ **Cross-validation** – Cross-validation, also noted CV, is a method that is used to select a model that does not rely too much on the initial training set. The different types are summed up in the table below:

<i>k</i> -fold	Leave- <i>p</i> -out
<ul style="list-style-type: none"> <li>- Training on <math>k - 1</math> folds and assessment on the remaining one</li> <li>- Generally <math>k = 5</math> or <math>10</math></li> </ul>	<ul style="list-style-type: none"> <li>- Training on <math>n - p</math> observations and assessment on the <math>p</math> remaining ones</li> <li>- Case <math>p = 1</math> is called leave-one-out</li> </ul>

The most commonly used method is called *k*-fold cross-validation and splits the training data into *k* folds to validate the model on one fold while training the model on the *k* - 1 other folds, all of this *k* times. The error is then averaged over the *k* folds and is named cross-validation error.

Fold	Dataset	Validation error	Cross-validation error
1		$\epsilon_1$	$\frac{\epsilon_1 + \dots + \epsilon_k}{k}$
2		$\epsilon_2$	
⋮	⋮	⋮	
<i>k</i>		$\epsilon_k$	

□ **Regularization** – The regularization procedure aims at avoiding the model to overfit the data and thus deals with high variance issues. The following table sums up the different types of commonly used regularization techniques:

LASSO	Ridge	Elastic Net
<ul style="list-style-type: none"> <li>- Shrinks coefficients to 0</li> <li>- Good for variable selection</li> </ul>	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
$\dots + \lambda \ \theta\ _1$ $\lambda \in \mathbb{R}$	$\dots + \lambda \ \theta\ _2^2$ $\lambda \in \mathbb{R}$	$\dots + \lambda \left[ (1 - \alpha) \ \theta\ _1 + \alpha \ \theta\ _2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0, 1]$

□ **Model selection** – Train model on training set, then evaluate on the development set, then pick best performance model on the development set, and retrain all of that model on the whole training set.

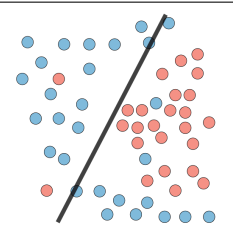
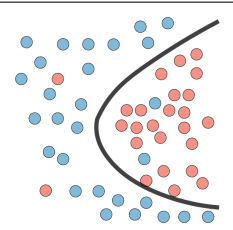
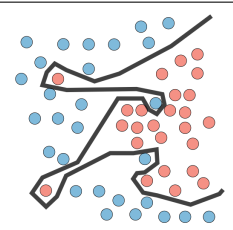
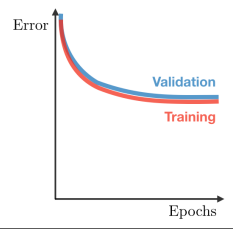
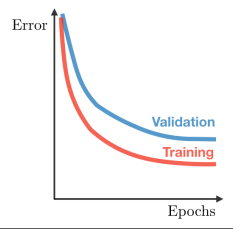
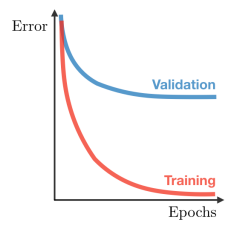
### 4.3 Diagnostics

□ **Bias** – The bias of a model is the difference between the expected prediction and the correct model that we try to predict for given data points.

□ **Variance** – The variance of a model is the variability of the model prediction for given data points.

□ **Bias/variance tradeoff** – The simpler the model, the higher the bias, and the more complex the model, the higher the variance.

	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none"> <li>- High training error</li> <li>- Training error close to test error</li> <li>- High bias</li> </ul>	<ul style="list-style-type: none"> <li>- Training error slightly lower than test error</li> </ul>	<ul style="list-style-type: none"> <li>- Low training error</li> <li>- Training error much lower than test error</li> <li>- High variance</li> </ul>
Regression			

<b>Classification</b>			
<b>Deep learning</b>			
<b>Remedies</b>	<ul style="list-style-type: none"> <li>- Complexify model</li> <li>- Add more features</li> <li>- Train longer</li> </ul>		<ul style="list-style-type: none"> <li>- Regularize</li> <li>- Get more data</li> </ul>

□ **Error analysis** – Error analysis is analyzing the root cause of the difference in performance between the current and the perfect models.

□ **Ablative analysis** – Ablative analysis is analyzing the root cause of the difference in performance between the current and the baseline models.

## 5 Refreshers

### 5.1 Probabilities and Statistics

#### 5.1.1 Introduction to Probability and Combinatorics

□ **Sample space** – The set of all possible outcomes of an experiment is known as the sample space of the experiment and is denoted by  $S$ .

□ **Event** – Any subset  $E$  of the sample space is known as an event. That is, an event is a set consisting of possible outcomes of the experiment. If the outcome of the experiment is contained in  $E$ , then we say that  $E$  has occurred.

□ **Axioms of probability** – For each event  $E$ , we denote  $P(E)$  as the probability of event  $E$  occurring. By noting  $E_1, \dots, E_n$  mutually exclusive events, we have the 3 following axioms:

$$(1) \quad 0 \leq P(E) \leq 1 \quad (2) \quad P(S) = 1 \quad (3) \quad P\left(\bigcup_{i=1}^n E_i\right) = \sum_{i=1}^n P(E_i)$$

□ **Permutation** – A permutation is an arrangement of  $r$  objects from a pool of  $n$  objects, in a given order. The number of such arrangements is given by  $P(n, r)$ , defined as:

$$P(n, r) = \frac{n!}{(n-r)!}$$

□ **Combination** – A combination is an arrangement of  $r$  objects from a pool of  $n$  objects, where the order does not matter. The number of such arrangements is given by  $C(n, r)$ , defined as:

$$C(n, r) = \frac{P(n, r)}{r!} = \frac{n!}{r!(n-r)!}$$

*Remark: we note that for  $0 \leq r \leq n$ , we have  $P(n, r) \geq C(n, r)$ .*

#### 5.1.2 Conditional Probability

□ **Bayes' rule** – For events  $A$  and  $B$  such that  $P(B) > 0$ , we have:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

*Remark: we have  $P(A \cap B) = P(A)P(B|A) = P(A|B)P(B)$ .*

□ **Partition** – Let  $\{A_i, i \in [1, n]\}$  be such that for all  $i$ ,  $A_i \neq \emptyset$ . We say that  $\{A_i\}$  is a partition if we have:

$$\forall i \neq j, A_i \cap A_j = \emptyset \quad \text{and} \quad \bigcup_{i=1}^n A_i = S$$

*Remark: for any event  $B$  in the sample space, we have  $P(B) = \sum_{i=1}^n P(B|A_i)P(A_i)$ .*

□ **Extended form of Bayes' rule** – Let  $\{A_i, i \in [1, n]\}$  be a partition of the sample space. We have:

$$P(A_k|B) = \frac{P(B|A_k)P(A_k)}{\sum_{i=1}^n P(B|A_i)P(A_i)}$$

□ **Independence** – Two events  $A$  and  $B$  are independent if and only if we have:

$$P(A \cap B) = P(A)P(B)$$

### 5.1.3 Random Variables

□ **Random variable** – A random variable, often noted  $X$ , is a function that maps every element in a sample space to a real line.

□ **Cumulative distribution function (CDF)** – The cumulative distribution function  $F$ , which is monotonically non-decreasing and is such that  $\lim_{x \rightarrow -\infty} F(x) = 0$  and  $\lim_{x \rightarrow +\infty} F(x) = 1$ , is defined as:

$$F(x) = P(X \leq x)$$

*Remark: we have  $P(a < X \leq B) = F(b) - F(a)$ .*

□ **Probability density function (PDF)** – The probability density function  $f$  is the probability that  $X$  takes on values between two adjacent realizations of the random variable.

□ **Relationships involving the PDF and CDF** – Here are the important properties to know in the discrete (D) and the continuous (C) cases.

Case	CDF $F$	PDF $f$	Properties of PDF
(D)	$F(x) = \sum_{x_i \leq x} P(X = x_i)$	$f(x_j) = P(X = x_j)$	$0 \leq f(x_j) \leq 1$ and $\sum_j f(x_j) = 1$
(C)	$F(x) = \int_{-\infty}^x f(y)dy$	$f(x) = \frac{dF}{dx}$	$f(x) \geq 0$ and $\int_{-\infty}^{+\infty} f(x)dx = 1$

□ **Variance** – The variance of a random variable, often noted  $\text{Var}(X)$  or  $\sigma^2$ , is a measure of the spread of its distribution function. It is determined as follows:

$$\text{Var}(X) = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

□ **Standard deviation** – The standard deviation of a random variable, often noted  $\sigma$ , is a measure of the spread of its distribution function which is compatible with the units of the actual random variable. It is determined as follows:

$$\sigma = \sqrt{\text{Var}(X)}$$

□ **Expectation and Moments of the Distribution** – Here are the expressions of the expected value  $E[X]$ , generalized expected value  $E[g(X)]$ ,  $k^{th}$  moment  $E[X^k]$  and characteristic function  $\psi(\omega)$  for the discrete and continuous cases:

Case	$E[X]$	$E[g(X)]$	$E[X^k]$	$\psi(\omega)$
(D)	$\sum_{i=1}^n x_i f(x_i)$	$\sum_{i=1}^n g(x_i) f(x_i)$	$\sum_{i=1}^n x_i^k f(x_i)$	$\sum_{i=1}^n f(x_i) e^{i\omega x_i}$
(C)	$\int_{-\infty}^{+\infty} x f(x) dx$	$\int_{-\infty}^{+\infty} g(x) f(x) dx$	$\int_{-\infty}^{+\infty} x^k f(x) dx$	$\int_{-\infty}^{+\infty} f(x) e^{i\omega x} dx$

*Remark: we have  $e^{i\omega x} = \cos(\omega x) + i \sin(\omega x)$ .*

□ **Revisiting the  $k^{th}$  moment** – The  $k^{th}$  moment can also be computed with the characteristic function as follows:

$$E[X^k] = \frac{1}{i^k} \left[ \frac{\partial^k \psi}{\partial \omega^k} \right]_{\omega=0}$$

□ **Transformation of random variables** – Let the variables  $X$  and  $Y$  be linked by some function. By noting  $f_X$  and  $f_Y$  the distribution function of  $X$  and  $Y$  respectively, we have:

$$f_Y(y) = f_X(x) \left| \frac{dx}{dy} \right|$$

□ **Leibniz integral rule** – Let  $g$  be a function of  $x$  and potentially  $c$ , and  $a, b$  boundaries that may depend on  $c$ . We have:

$$\frac{\partial}{\partial c} \left( \int_a^b g(x) dx \right) = \frac{\partial b}{\partial c} \cdot g(b) - \frac{\partial a}{\partial c} \cdot g(a) + \int_a^b \frac{\partial g}{\partial c}(x) dx$$

□ **Chebyshev's inequality** – Let  $X$  be a random variable with expected value  $\mu$  and standard deviation  $\sigma$ . For  $k, \sigma > 0$ , we have the following inequality:

$$P(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2}$$

### 5.1.4 Jointly Distributed Random Variables

□ **Conditional density** – The conditional density of  $X$  with respect to  $Y$ , often noted  $f_{X|Y}$ , is defined as follows:

$$f_{X|Y}(x) = \frac{f_{XY}(x, y)}{f_Y(y)}$$

□ **Independence** – Two random variables  $X$  and  $Y$  are said to be independent if we have:

$$f_{XY}(x, y) = f_X(x) f_Y(y)$$

□ **Marginal density and cumulative distribution** – From the joint density probability function  $f_{XY}$ , we have:

Case	Marginal density	Cumulative function
(D)	$f_X(x_i) = \sum_j f_{XY}(x_i, y_j)$	$F_{XY}(x, y) = \sum_{x_i \leq x} \sum_{y_j \leq y} f_{XY}(x_i, y_j)$
(C)	$f_X(x) = \int_{-\infty}^{+\infty} f_{XY}(x, y) dy$	$F_{XY}(x, y) = \int_{-\infty}^x \int_{-\infty}^y f_{XY}(x', y') dx' dy'$

□ **Distribution of a sum of independent random variables** – Let  $Y = X_1 + \dots + X_n$  with  $X_1, \dots, X_n$  independent. We have:

$$\psi_Y(\omega) = \prod_{k=1}^n \psi_{X_k}(\omega)$$

□ **Covariance** – We define the covariance of two random variables  $X$  and  $Y$ , that we note  $\sigma_{XY}^2$  or more commonly  $\text{Cov}(X, Y)$ , as follows:

$$\text{Cov}(X, Y) \triangleq \sigma_{XY}^2 = E[(X - \mu_X)(Y - \mu_Y)] = E[XY] - \mu_X \mu_Y$$

□ **Correlation** – By noting  $\sigma_X, \sigma_Y$  the standard deviations of  $X$  and  $Y$ , we define the correlation between the random variables  $X$  and  $Y$ , noted  $\rho_{XY}$ , as follows:

$$\rho_{XY} = \frac{\sigma_{XY}^2}{\sigma_X \sigma_Y}$$

*Remarks: For any  $X, Y$ , we have  $\rho_{XY} \in [-1, 1]$ . If  $X$  and  $Y$  are independent, then  $\rho_{XY} = 0$ .*

□ **Main distributions** – Here are the main distributions to have in mind:

Type	Distribution	PDF	$\psi(\omega)$	$E[X]$	$\text{Var}(X)$
(D)	$X \sim \mathcal{B}(n, p)$ Binomial	$P(X = x) = \binom{n}{x} p^x q^{n-x}$ $x \in \llbracket 0, n \rrbracket$	$(pe^{i\omega} + q)^n$	$np$	$npq$
	$X \sim \text{Po}(\mu)$ Poisson	$P(X = x) = \frac{\mu^x}{x!} e^{-\mu}$ $x \in \mathbb{N}$	$e^{\mu(e^{i\omega} - 1)}$	$\mu$	$\mu$
(C)	$X \sim \mathcal{U}(a, b)$ Uniform	$f(x) = \frac{1}{b-a}$ $x \in [a, b]$	$\frac{e^{i\omega b} - e^{i\omega a}}{(b-a)i\omega}$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
	$X \sim \mathcal{N}(\mu, \sigma)$ Gaussian	$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$ $x \in \mathbb{R}$	$e^{i\omega\mu - \frac{1}{2}\omega^2\sigma^2}$	$\mu$	$\sigma^2$
	$X \sim \text{Exp}(\lambda)$ Exponential	$f(x) = \lambda e^{-\lambda x}$ $x \in \mathbb{R}_+$	$\frac{1}{1 - \frac{i\omega}{\lambda}}$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$

### 5.1.5 Parameter estimation

□ **Random sample** – A random sample is a collection of  $n$  random variables  $X_1, \dots, X_n$  that are independent and identically distributed with  $X$ .

□ **Estimator** – An estimator  $\hat{\theta}$  is a function of the data that is used to infer the value of an unknown parameter  $\theta$  in a statistical model.

□ **Bias** – The bias of an estimator  $\hat{\theta}$  is defined as being the difference between the expected value of the distribution of  $\hat{\theta}$  and the true value, i.e.:

$$\text{Bias}(\hat{\theta}) = E[\hat{\theta}] - \theta$$

*Remark: an estimator is said to be unbiased when we have  $E[\hat{\theta}] = \theta$ .*

□ **Sample mean and variance** – The sample mean and the sample variance of a random sample are used to estimate the true mean  $\mu$  and the true variance  $\sigma^2$  of a distribution, are noted  $\bar{X}$  and  $s^2$  respectively, and are such that:

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i \quad \text{and} \quad s^2 = \hat{\sigma}^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

□ **Central Limit Theorem** – Let us have a random sample  $X_1, \dots, X_n$  following a given distribution with mean  $\mu$  and variance  $\sigma^2$ , then we have:

$$\bar{X} \underset{n \rightarrow +\infty}{\sim} \mathcal{N}\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

## 5.2 Linear Algebra and Calculus

### 5.2.1 General notations

□ **Vector** – We note  $x \in \mathbb{R}^n$  a vector with  $n$  entries, where  $x_i \in \mathbb{R}$  is the  $i^{th}$  entry:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$$

□ **Matrix** – We note  $A \in \mathbb{R}^{m \times n}$  a matrix with  $m$  rows and  $n$  columns, where  $A_{i,j} \in \mathbb{R}$  is the entry located in the  $i^{th}$  row and  $j^{th}$  column:

$$A = \begin{pmatrix} A_{1,1} & \cdots & A_{1,n} \\ \vdots & & \vdots \\ A_{m,1} & \cdots & A_{m,n} \end{pmatrix} \in \mathbb{R}^{m \times n}$$

*Remark: the vector  $x$  defined above can be viewed as a  $n \times 1$  matrix and is more particularly called a column-vector.*

□ **Identity matrix** – The identity matrix  $I \in \mathbb{R}^{n \times n}$  is a square matrix with ones in its diagonal and zero everywhere else:

$$I = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{pmatrix}$$

Remark: for all matrices  $A \in \mathbb{R}^{n \times n}$ , we have  $A \times I = I \times A = A$ .

□ **Diagonal matrix** – A diagonal matrix  $D \in \mathbb{R}^{n \times n}$  is a square matrix with nonzero values in its diagonal and zero everywhere else:

$$D = \begin{pmatrix} d_1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & d_n \end{pmatrix}$$

Remark: we also note  $D$  as  $\text{diag}(d_1, \dots, d_n)$ .

### 5.2.2 Matrix operations

□ **Vector-vector multiplication** – There are two types of vector-vector products:

- inner product: for  $x, y \in \mathbb{R}^n$ , we have:

$$x^T y = \sum_{i=1}^n x_i y_i \in \mathbb{R}$$

- outer product: for  $x \in \mathbb{R}^m, y \in \mathbb{R}^n$ , we have:

$$xy^T = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_n \\ \vdots & & \vdots \\ x_m y_1 & \cdots & x_m y_n \end{pmatrix} \in \mathbb{R}^{m \times n}$$

□ **Matrix-vector multiplication** – The product of matrix  $A \in \mathbb{R}^{m \times n}$  and vector  $x \in \mathbb{R}^n$  is a vector of size  $\mathbb{R}^m$ , such that:

$$Ax = \begin{pmatrix} a_{r,1}^T x \\ \vdots \\ a_{r,m}^T x \end{pmatrix} = \sum_{i=1}^n a_{c,i} x_i \in \mathbb{R}^m$$

where  $a_{r,i}^T$  are the vector rows and  $a_{c,j}$  are the vector columns of  $A$ , and  $x_i$  are the entries of  $x$ .

□ **Matrix-matrix multiplication** – The product of matrices  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times p}$  is a matrix of size  $\mathbb{R}^{m \times p}$ , such that:

$$AB = \begin{pmatrix} a_{r,1}^T b_{c,1} & \cdots & a_{r,1}^T b_{c,p} \\ \vdots & & \vdots \\ a_{r,m}^T b_{c,1} & \cdots & a_{r,m}^T b_{c,p} \end{pmatrix} = \sum_{i=1}^n a_{c,i} b_{r,i}^T \in \mathbb{R}^{m \times p}$$

where  $a_{r,i}^T, b_{r,i}^T$  are the vector rows and  $a_{c,j}, b_{c,j}$  are the vector columns of  $A$  and  $B$  respectively.

□ **Transpose** – The transpose of a matrix  $A \in \mathbb{R}^{m \times n}$ , noted  $A^T$ , is such that its entries are flipped:

$$\forall i, j, \quad A_{i,j}^T = A_{j,i}$$

Remark: for matrices  $A, B$ , we have  $(AB)^T = B^T A^T$ .

□ **Inverse** – The inverse of an invertible square matrix  $A$  is noted  $A^{-1}$  and is the only matrix such that:

$$AA^{-1} = A^{-1}A = I$$

Remark: not all square matrices are invertible. Also, for matrices  $A, B$ , we have  $(AB)^{-1} = B^{-1}A^{-1}$

□ **Trace** – The trace of a square matrix  $A$ , noted  $\text{tr}(A)$ , is the sum of its diagonal entries:

$$\text{tr}(A) = \sum_{i=1}^n A_{i,i}$$

Remark: for matrices  $A, B$ , we have  $\text{tr}(A^T) = \text{tr}(A)$  and  $\text{tr}(AB) = \text{tr}(BA)$

□ **Determinant** – The determinant of a square matrix  $A \in \mathbb{R}^{n \times n}$ , noted  $|A|$  or  $\det(A)$  is expressed recursively in terms of  $A_{\setminus i, \setminus j}$ , which is the matrix  $A$  without its  $i^{\text{th}}$  row and  $j^{\text{th}}$  column, as follows:

$$\det(A) = |A| = \sum_{j=1}^n (-1)^{i+j} A_{i,j} |A_{\setminus i, \setminus j}|$$

Remark:  $A$  is invertible if and only if  $|A| \neq 0$ . Also,  $|AB| = |A||B|$  and  $|A^T| = |A|$ .

### 5.2.3 Matrix properties

□ **Symmetric decomposition** – A given matrix  $A$  can be expressed in terms of its symmetric and antisymmetric parts as follows:

$$A = \underbrace{\frac{A + A^T}{2}}_{\text{Symmetric}} + \underbrace{\frac{A - A^T}{2}}_{\text{Antisymmetric}}$$

□ **Norm** – A norm is a function  $N : V \rightarrow [0, +\infty[$  where  $V$  is a vector space, and such that for all  $x, y \in V$ , we have:

- $N(x + y) \leq N(x) + N(y)$
- $N(ax) = |a|N(x)$  for  $a$  a scalar
- if  $N(x) = 0$ , then  $x = 0$

For  $x \in V$ , the most commonly used norms are summed up in the table below:

Norm	Notation	Definition	Use case
Manhattan, $L^1$	$\ x\ _1$	$\sum_{i=1}^n  x_i $	LASSO regularization
Euclidean, $L^2$	$\ x\ _2$	$\sqrt{\sum_{i=1}^n x_i^2}$	Ridge regularization
$p$ -norm, $L^p$	$\ x\ _p$	$\left(\sum_{i=1}^n x_i^p\right)^{\frac{1}{p}}$	Hölder inequality
Infinity, $L^\infty$	$\ x\ _\infty$	$\max_i  x_i $	Uniform convergence

□ **Linearly dependence** – A set of vectors is said to be linearly dependent if one of the vectors in the set can be defined as a linear combination of the others.  
*Remark: if no vector can be written this way, then the vectors are said to be linearly independent.*

□ **Matrix rank** – The rank of a given matrix  $A$  is noted  $\text{rank}(A)$  and is the dimension of the vector space generated by its columns. This is equivalent to the maximum number of linearly independent columns of  $A$ .

□ **Positive semi-definite matrix** – A matrix  $A \in \mathbb{R}^{n \times n}$  is positive semi-definite (PSD) and is noted  $A \succeq 0$  if we have:

$$A = A^T \quad \text{and} \quad \forall x \in \mathbb{R}^n, \quad x^T A x \geq 0$$

*Remark: similarly, a matrix  $A$  is said to be positive definite, and is noted  $A \succ 0$ , if it is a PSD matrix which satisfies for all non-zero vector  $x$ ,  $x^T A x > 0$ .*

□ **Eigenvalue, eigenvector** – Given a matrix  $A \in \mathbb{R}^{n \times n}$ ,  $\lambda$  is said to be an eigenvalue of  $A$  if there exists a vector  $z \in \mathbb{R}^n \setminus \{0\}$ , called eigenvector, such that we have:

$$Az = \lambda z$$

□ **Spectral theorem** – Let  $A \in \mathbb{R}^{n \times n}$ . If  $A$  is symmetric, then  $A$  is diagonalizable by a real orthogonal matrix  $U \in \mathbb{R}^{n \times n}$ . By noting  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ , we have:

$$\exists \Lambda \text{ diagonal, } A = U \Lambda U^T$$

□ **Singular-value decomposition** – For a given matrix  $A$  of dimensions  $m \times n$ , the singular-value decomposition (SVD) is a factorization technique that guarantees the existence of  $U$   $m \times m$  unitary,  $\Sigma$   $m \times n$  diagonal and  $V$   $n \times n$  unitary matrices, such that:

$$A = U \Sigma V^T$$

### 5.2.4 Matrix calculus

□ **Gradient** – Let  $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$  be a function and  $A \in \mathbb{R}^{m \times n}$  be a matrix. The gradient of  $f$  with respect to  $A$  is a  $m \times n$  matrix, noted  $\nabla_A f(A)$ , such that:

$$\left(\nabla_A f(A)\right)_{i,j} = \frac{\partial f(A)}{\partial A_{i,j}}$$

*Remark: the gradient of  $f$  is only defined when  $f$  is a function that returns a scalar.*

□ **Hessian** – Let  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  be a function and  $x \in \mathbb{R}^n$  be a vector. The hessian of  $f$  with respect to  $x$  is a  $n \times n$  symmetric matrix, noted  $\nabla_x^2 f(x)$ , such that:

$$\left(\nabla_x^2 f(x)\right)_{i,j} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$$

*Remark: the hessian of  $f$  is only defined when  $f$  is a function that returns a scalar.*

□ **Gradient operations** – For matrices  $A, B, C$ , the following gradient properties are worth having in mind:

$$\nabla_A \text{tr}(AB) = B^T \quad \nabla_{A^T} f(A) = (\nabla_A f(A))^T$$

$$\nabla_A \text{tr}(ABA^T C) = CAB + C^T A B^T \quad \nabla_A |A| = |A| (A^{-1})^T$$