

# Introduction to Machine Learning

Martino Papa

January 2022

# Indice

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduzione</b>                               | <b>3</b>  |
| 1.1      | Utilizzo di Machine Learning . . . . .            | 3         |
| 1.2      | Machine Learning, AI, Deep Learning . . . . .     | 4         |
| 1.3      | Tipi di apprendimento . . . . .                   | 5         |
| 1.3.1    | Supervised Learning . . . . .                     | 5         |
| 1.3.2    | Unsupervised Learning . . . . .                   | 6         |
| 1.3.3    | Reinforcement Learning . . . . .                  | 6         |
| 1.3.4    | Altre variazioni . . . . .                        | 7         |
| 1.3.5    | Lazy vs Eager learner . . . . .                   | 7         |
| 1.4      | Funzionamento generale . . . . .                  | 7         |
| 1.4.1    | Task . . . . .                                    | 8         |
| 1.4.2    | Data . . . . .                                    | 8         |
| 1.4.3    | Model and Hypothesis space . . . . .              | 9         |
| 1.4.4    | Soluzione . . . . .                               | 9         |
| 1.4.5    | Esempio: Polynomial curve fitting . . . . .       | 10        |
| <b>2</b> | <b>Supervised learning models</b>                 | <b>12</b> |
| 2.1      | K-Nearest neighbor . . . . .                      | 12        |
| 2.1.1    | Come misuriamo la distanza? . . . . .             | 12        |
| 2.1.2    | Decision Boundaries . . . . .                     | 13        |
| 2.1.3    | Curse of Dimensionality . . . . .                 | 13        |
| 2.1.4    | Costo computazionale . . . . .                    | 14        |
| 2.1.5    | Riassumendo . . . . .                             | 14        |
| 2.2      | Linear Model . . . . .                            | 14        |
| 2.2.1    | Definizione metodi lineari . . . . .              | 14        |
| 2.2.2    | Perceptron learning algorithm . . . . .           | 15        |
| 2.2.3    | Multiclass classification . . . . .               | 15        |
| 2.2.4    | Gradient Descent . . . . .                        | 16        |
| 2.2.5    | Regularization . . . . .                          | 17        |
| 2.3      | Support Vector Machines . . . . .                 | 17        |
| 2.3.1    | Hard margin classification . . . . .              | 18        |
| 2.3.2    | Soft margin classification . . . . .              | 18        |
| 2.3.3    | Risolvere il problema di ottimizzazione . . . . . | 18        |
| 2.4      | Decision trees . . . . .                          | 18        |
| 2.4.1    | Learing algorithm . . . . .                       | 19        |
| 2.4.2    | Overfitting . . . . .                             | 20        |
| 2.4.3    | Riassunto . . . . .                               | 20        |
| 2.4.4    | Random Forests . . . . .                          | 21        |
| <b>3</b> | <b>Neural network</b>                             | <b>22</b> |
| 3.1      | Feed-Forward neural network . . . . .             | 23        |
| 3.1.1    | Definizione . . . . .                             | 23        |
| 3.1.2    | Training . . . . .                                | 23        |
| 3.1.3    | First AI winter . . . . .                         | 24        |
| 3.2      | Backpropagation (1986) . . . . .                  | 25        |
| 3.2.1    | Training . . . . .                                | 25        |
| 3.2.2    | Second AI winter . . . . .                        | 26        |
| 3.3      | Scelta di un ottimizzatore . . . . .              | 26        |
| 3.3.1    | Batch gradient descent (BGD) . . . . .            | 26        |
| 3.3.2    | Stochastic gradient descent (SGD) . . . . .       | 26        |
| 3.3.3    | MiniBatches . . . . .                             | 27        |

|          |  |           |
|----------|--|-----------|
| 3.4      | Convolutional neural network (CNN) . . . . .           | 27        |
| 3.4.1    | Architettura . . . . .                                 | 28        |
| 3.4.2    | Applicazioni . . . . .                                 | 28        |
| 3.5      | Recurrent neural network (RNN) . . . . .               | 29        |
| <b>4</b> | <b>Unsupervised learning</b>                           | <b>30</b> |
| 4.1      | Dimensionality reduction . . . . .                     | 30        |
| 4.1.1    | Principal Component Analysis (PCA) . . . . .           | 30        |
| 4.1.2    | Altre tecniche per dimensionality reduction . . . . .  | 31        |
| 4.2      | Clustering . . . . .                                   | 31        |
| 4.2.1    | K-Means clustering . . . . .                           | 31        |
| 4.2.2    | Expectation Maximization (EM) clustering . . . . .     | 33        |
| 4.2.3    | Spectral clustering . . . . .                          | 34        |
| 4.3      | Density estimation . . . . .                           | 35        |
| 4.3.1    | Variational AutoEncoders (VAE) . . . . .               | 36        |
| 4.3.2    | Generative adversial networks (GANs) . . . . .         | 37        |
| 4.3.3    | Denoising diffusion models . . . . .                   | 38        |
| <b>5</b> | <b>Reinforcement Learing</b>                           | <b>40</b> |
| 5.1      | Markov decision process (MDP) . . . . .                | 40        |
| 5.1.1    | MDP loop . . . . .                                     | 41        |
| 5.1.2    | Reinforcement learning vs Supervised learing . . . . . | 41        |
| 5.2      | Reinforcement learning methods . . . . .               | 41        |
| 5.2.1    | Value based methods . . . . .                          | 42        |
| 5.2.2    | Policy gradient methods . . . . .                      | 43        |

# Capitolo 1

## Introduzione

**Definizione 1.1** (Machine Learning, ML). Vederemo più di una definizione, ognuna di esse introduce dei concetti essenziali:

- ML è lo studio di algoritmi che migliorano automaticamente attraverso l'esperienza. (Wikipedia)
- ML è la scienza che permette ai computer di agire senza essere stati programmati esplicitamente. (A. Samuel)
- ML tratta la scoperta automatica di regolarità nei dati attraverso l'uso di algoritmi. (Christopher M. Bishop)
- ML si occupa di sviluppare metodi che possano automaticamente scoprire pattern nei dati al fine di prevedere dati futuri. (Kevin P. Murphy)
- ML è un programma che attraverso l'**esperienza E** impara a svolgere delle classi di **task T** rispettivamente a una **misura** di performance **P**. (T. Mitchell)

Possiamo quindi concludere che un algoritmo di apprendimento ben definito è dato da una tripla  $\langle T, P, E \rangle$ .

esempi di  $\langle T, P, E \rangle$ :

T: Riconoscimento di parole scritte a mano.

P: Percentuale di parole classificate correttamente

E: Database di immagini di parole già classificate da umani

T: Categorizzare le email in spam o legittime

P: Percentuale di email classificate correttamente

E: Database di email classificate da umani

### 1.1 Utilizzo di Machine Learning

Machine Learning viene utilizzato quando:

- non sono disponibili competenze umane (viaggi su Marte)
- gli umani non possono spiegare le loro competenze (speech recognition)
- i modelli devono essere customizzati (medicina)
- i modelli sono basati su una quantità di dati massiccia

NB: Machine Learning non è sempre utile! Ad esempio non c'è bisogno di applicare l'apprendimento per calcolare una busta paga.

#### Recognition Patterns

- riconoscimento caratteri
- riconoscimento di espressioni facciali e emozioni
- riconoscimento di patologie

**Generating Patterns** Generazione di immagini o di sequenze di movimenti.

**Recognizing Anomalies** Riconoscimento di transazioni inusuali da parte di carte di credito.

## Prediction

- previsione dei mercati finanziari
- previsione dei traiettorie (autonomus driving)
- predizione delle mosse migliori in un videogame

## Successi del Machine Learning

- Face detection (2002)
- Pedestrian detection (2005)
- Body Tracking RGB-D

## Discipline correlate

- data mining: analisi dei dati senza predizioni.
- inferenza in statistica: processo che consiste nel trarre conclusioni su una popolazione estesa basandosi su un sottoinsieme di essa.
- pattern recognition
- signal processing
- ottimizzazione

## 1.2 Machine Learing, AI, Deep Learning

**Definizione 1.2** (Artificial Intelligence, AI). Si occupa di ideare programmi che apprendono dalla loro esperienza similmente agli esseri umani.

**Definizione 1.3** (Deep Learning). Con Deep Learning si indica l'utilizzo di una **rete neurale** con molteplici layer di nodi tra l'input e l'output. Ogni layer rappresenta un livello di astrazione.

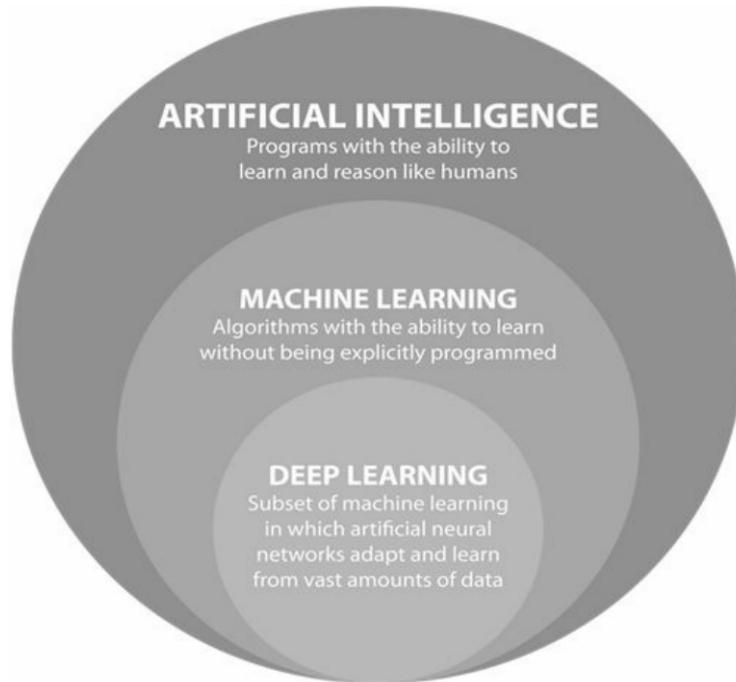


Figura 1.1: AI, Machine Learning, Deep Learning

## Deep Learning, perchè adesso?

- disponibilità di una grande quantità di dati
- accesso a una grande potenza di calcolo
- incremento delle ricerche teoriche e algoritmiche sul Machine Learning
- aumento del supporto da parte delle industrie

### 1.3 Tipi di apprendimento

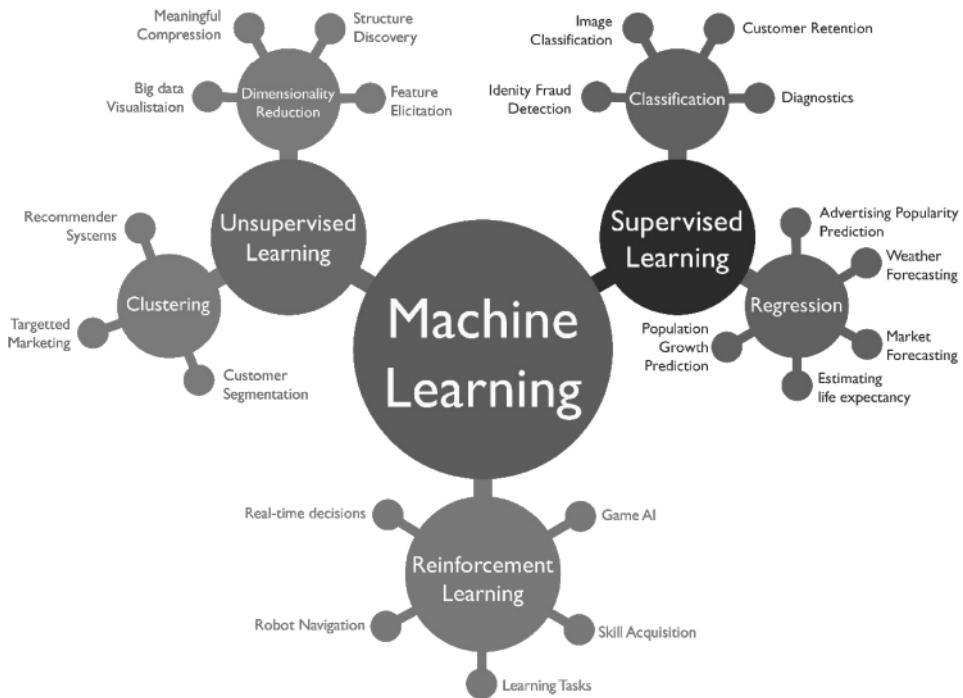


Figura 1.2: Tipi di apprendimento

#### 1.3.1 Supervised Learning

Supervised Learning è un paradigma di ML per problemi nei quali i dati a disposizione consistono in esempi etichettati (labeled).

Dato un insieme di dati  $\tau = \{(x_1, y_1), \dots, (x_m, y_m)\}$ , dove  $x_i$  corrisponde all'oggetto e  $y_i$  alla sua etichetta. Si apprende una funzione

$$f : \mathbb{R}^d \rightarrow \mathcal{Y} \quad (1.1)$$

che dato un oggetto  $x$  predice la sua etichetta  $y = f(x)$ .

**Classification** La label corrisponde a una categoria.

Applicazioni:

- riconoscimento facciale
- riconoscimento dei caratteri
- spam detection
- diagnosi mediche
- biometrica

**Regression** La label è una variabile reale ovvero  $y \in \mathbb{R}$ .

Applicazioni:

- economia/finanza, predizione dei valori azionari
- guida autonoma, predizione dell'angolo di sterzo, ...
- previsioni meteo

**Ranking** La label corrisponde a una posizione in classifica,  $y \in \mathbb{N}$ .

Applicazioni:

- scegliete le pagine web più pertinenti per una ricerca
- stabilire le preferenze di un utente (es. Netflix “my list”)
- data un immagine qualunque selezionare la più simile all'interno di un set

### 1.3.2 Unsupervised Learning

Unsupervised Learning è un paradigma di ML per problemi nei quali i dati a disposizione **non** sono etichettati. L'idea è quella di trovare dei pattern all'interno dei dati.

$$\tau = \{x_1, \dots, x_m\} \quad (1.2)$$

**Clustering** Si occupa di suddividere i dati in sottogruppi (**cluster**).

Applicazioni:

- analisi delle cerchie di amici nei social network
- raggruppare gli individui per la loro similitudine genetica
- image segmentation

**Anomaly Detection** Analizza un insieme di eventi o oggetti e individua quelli anomali o inusuali.

Applicazioni:

- riconoscere transazioni Anomalies
- video sorveglianza

**Dimensionality Reduction** Riduzione delle variabili considerate ottenendo un set di **variabili principali**.

### 1.3.3 Reinforcement Learning

Un agente (**agent**) impara dall'ambiente (**environment**) interagendo con esso e ricevendo ricompense (**rewards**) per aver completato delle azioni (**actions**).

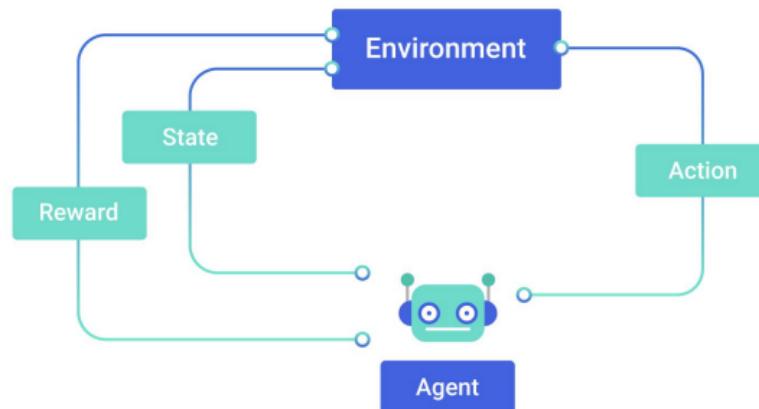


Figura 1.3: Reinforcement Learning

### 1.3.4 Altre variazioni

#### Tecniche di apprendimento

- **semi-supervised learning:** paradigma di ML che sfrutta dati etichettati (labeled) per etichettare una mole maggiore di dati inizialmente non etichettati. È impiegato in quanto è spesso più costoso ottenere labeled data.
- **active learning:** paradigma di ML in cui l'algoritmo può interrogare un utente (**teacher**) che gli indicherà quali risultati sono rilevanti al fine dell'apprendimento.

#### Tipi di modelli

- **generative o discriminative:** i modelli discriminative disegnano bordi nello spazio per dividere i dati mentre quelli generative provano a modellare l'effettiva distribuzione di ogni classe.
- **parametric o non-parametric:** nei modelli parametrici le informazioni sono rappresentate attraverso dei parametri. Quelli non parametrici sono spesso usati nei casi in cui i dati non possono essere descritti come numeri discreti.

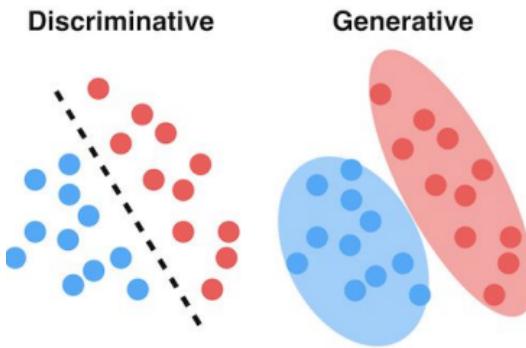


Figura 1.4: discriminative vs. generative

### 1.3.5 Lazy vs Eager learner

**Definizione 1.4** (Lazy learner). Gli algoritmo di tipo lazy learner salvano i dati (**training set**) in memoria (senza elaborarli) e operano su essi quando gli viene fornito un test example.

**Definizione 1.5** (Eager learner). Gli algoritmi di tipo eager learner costruiscono a partire dal training set un modello di classificazione che useranno poi per valutare i test example.

## 1.4 Funzionamento generale

Gli algoritmi di ML si basano su modelli di apprendimento probabilistici. In particolare ricavano dai dati una distribuzione di probabilità (**data generating distribution**) che sfrutteranno poi per risolvere le task a loro affidate.

**Definizione 1.6** (Features). Gli algoritmi descrivono gli esempi proposti tramite delle features. Ad esempio una mela può essere descritta dalle features (red, round, with leaf, ...). Questo causa ovviamente una perdita di dati, l'obiettivo è mitigare il più possibile questa perdita.

**Definizione 1.7** (Misura di probabilità). sia  $X$  un insieme,  $\mathcal{A}$   $\sigma$ -algebra in  $X$ . Una misura di probabilità su  $\mathcal{A}$  è una funzione  $\mathbb{P} : \mathcal{A} \rightarrow [0, 1]$  t.c.

- $\mathbb{P}(\emptyset) = 0$
- $\mathbb{P}(\Omega) = 1$  (normalizzazione)
- se  $\{E_j\}$  è una famiglia numerabile di insiemi di  $\mathcal{A}$  a due a due disgiunti, allora

$$\mathbb{P}\left(\bigcup_j E_j\right) = \sum_j \mathbb{P}(E_j) \tag{1.3}$$

**Definizione 1.8** (Distribuzione). Sia  $(\Omega, \mathcal{A}, \mathbb{P})$  uno spazio di probabilità,  $X$  una variabile aleatoria. Definiamo distribuzione di  $X$

$$\mu_X : \mathcal{B}(\mathbb{R}) \ni I \mapsto \mu_X(I) = \mathbb{P}(X \in I) \in [0, 1] \quad (1.4)$$

sa  $X$  variabile aleatoria discreta allora

$$\mu_X = \left\{ (x_k, p_k) \mid k \in \{0, \dots, n\} \subset \mathbb{N}, x_k \in \mathbb{R}, p_k \geq 0, \sum_{k=0}^n p_k = 1 \right\} \quad (1.5)$$

Per ogni insieme di features grazie alla data generating distribution sarà quindi possibile stabilire un risultato.

### 1.4.1 Task

**Definizione 1.9.** Diremo  $f \in \mathcal{Y}^{\mathcal{X}}$  se  $f : \mathcal{X} \rightarrow \mathcal{Y}$

**Definizione 1.10** (Task). Una task rappresenta il tipo di predizione impiegato per risolvere un problema per certi dati. Formalmente una task è una funzione

$$f : \mathcal{X} \rightarrow \mathcal{Y}, f \in \mathcal{F}_{\text{task}} \subset \mathcal{Y}^{\mathcal{X}} \quad (1.6)$$

La natura di  $\mathcal{X}, \mathcal{Y}$  e  $\mathcal{F}_{\text{task}}$  dipende dal tipo di task.

#### Classification

Nella Classification cerchiamo una funzione  $f$  definita come sopra che assegna ad ogni input una label. Formalmente quindi diremo  $f(x) \in \mathcal{Y} = \{c_1, \dots, c_k\}$  (immagine discreta).

#### Regression

Nella regressione assegnamo ad ogni input una label continua, abbiamo quindi  $f(x) \in \mathbb{R}$ .

#### Density Estimation

L'obiettivo è trovare una distribuzione di probabilità  $f : \mathcal{X} \rightarrow [0, 1]$  che soddisfi i dati  $x \in \mathcal{X}$ .

#### Clustering

Cerchiamo una funzione  $f : \mathcal{X} \rightarrow \mathbb{N}$  che assegna ad ogni input  $x \in \mathcal{X}$  un indice di gruppo (**cluster index**)  $f(x) \in \mathbb{N}$ .

#### Dimensionality Reduction

L'obiettivo è trovare una funzione  $f : \mathcal{X} \rightarrow \mathcal{Y}$  che mappi ogni dato in input  $x \in \mathcal{X}$  su un insieme di dimensione minore  $\mathcal{Y} \ni f(x)$ ,  $\dim(\mathcal{Y}) \ll \dim(\mathcal{X})$ .

### 1.4.2 Data

**Definizione 1.11** (Dati). Informazioni riguardo il problema da risolvere. Definiamo la distribuzione dei dati

$$p_{\text{data}} \in \Delta(\mathcal{A}) \quad (1.7)$$

- In Classification e Regression  $\mathcal{A} = \mathcal{X} \times \mathcal{Y}$ .
- In Estimation, Clustering e Dimensionality Reduction  $\mathcal{A} = \mathcal{X}$ .

La distribuzione dei dati è tipicamente sconosciuta ma possiamo ottenerne una semplificazione.

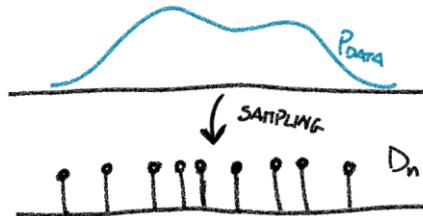


Figura 1.5: semplificazione della distribuzione

I dati, come visto in precedenza, vengono usati per generare un modello di apprendimento probabilistico. Si ricava una distribuzione di probabilità chiamata **data generating distribution**. Sia i dati di allenamento (**training data**) che i dati per i test (**test set**) sono generati basandosi su questa distribuzione.

NB: distribuzione di dati  $\neq$  distribuzione di probabilità.

memo:  $\Delta(\mathcal{A})$  significa “lo spazio  $\mathcal{A}$ ”.

**Definizione 1.12** (Training e validation set). Il training set è l’insieme di dati usato dall’algoritmo per trovare un programma che risolva un determinato problema. I risultati vengono poi testati grazie al validation set.

**Training set design** Il fallimento degli algoritmi di ML è spesso causato da una cattiva selezione dei dati di apprendimento. Il problema sta nel fatto che si potrebbero presentare correlazioni non volute dalle quali l’algoritmo trarrebbe conclusioni errate.

“The US Army trained a program to differentiate American tanks from Russian tanks with 100% accuracy. Only later did analysts realized that the American tanks had been photographed on a sunny day and the Russian tanks had been photographed on a cloudy day. The computer had learned to detect brightness.” [probably a legend]

Figura 1.6: leggenda US Army

### 1.4.3 Model and Hypothesis space

**Definizione 1.13** (Modello). Un modello può essere visto come un “programma” in grado di risolvere un problema. Nello specifico è l’implementazione di una funzione  $f \in \mathcal{F}_{\text{task}}$ .

**Definizione 1.14** (Spazio delle ipotesi). Un insieme di modelli forma uno spazio delle ipotesi (**Hypothesis space**).

$$\mathcal{H} \in \mathcal{F}_{\text{task}} \quad (1.8)$$

L’algoritmo di ML cerca il modello risolutivo per il problema all’interno di un Hypothesis space in modo da restringere i casi possibili.

### 1.4.4 Soluzione

#### Soluzione ideale

L’obiettivo è quello di minimizzare la **funzione di errore**  $E(f; p_{\text{data}})$ . La soluzione ideale sarà quindi

$$f^* \in \arg \min_{f \in \mathcal{F}_{\text{task}}} E(f; p_{\text{data}}) \quad (\text{soluzione ideale})$$

#### Soluzione fattibile

È necessario restringere il focus su una funzione che possa essere effettivamente implementata e calcolata. Per farlo la cerchiamo all’interno di uno spazio delle ipotesi  $\mathcal{H} \in \mathcal{F}_{\text{task}}$ .

$$f_{\mathcal{H}}^* \in \arg \min_{f \in \mathcal{H}} E(f; p_{\text{data}}) \quad (\text{actual target})$$

#### Soluzione reale

Purtroppo però  $p_{\text{data}}$  è tipicamente sconosciuta quindi abbiamo bisogno di alcuni dati di allenamento (**training set**).

$$\mathcal{D}_n = \{z_1, \dots, z_n\}, z_i = (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}, z_i \sim p_{\text{data}} \quad (1.9)$$

Possiamo quindi basarci sul **training error**  $E(f; \mathcal{D}_n)$  per trovare la soluzione “migliore”.

$$f_{\mathcal{H}}^*(\mathcal{D}_n) \in \arg \min_{f \in \mathcal{H}} E(f; \mathcal{D}_n) \quad (\text{soluzione reale})$$

La soluzione reale è una **soluzione fattibile sui dati di allenamento**. Definiamo inoltre

$$f^*(\mathcal{D}_n) \in \arg \min_{f \in \mathcal{F}_{\text{task}}} E(f; \mathcal{D}_n) \quad (\text{soluzione ideale sui dati di allenamento})$$

**Definizione 1.15** (Funzione di errore). Determina l’esattezza della soluzione, il valore di  $f$  che la minimizza corrisponde alla soluzione “ottima”. È spesso scritta in termini di perdita puntuale  $l(f; z)$  (**pointwise loss**)

$$E(f; p_{\text{data}}) = \mathbb{E}_{z \sim p_{\text{data}}} [l(f; z)] \quad (1.10)$$

$$E(f; \mathcal{D}_n) = \frac{1}{n} \sum_{i=1}^n l(f; z_i) \quad (1.11)$$

## Calcolo della soluzione reale

Per ottimizzare il calcolo di  $f_{\mathcal{H}}^*(\mathcal{D}_n)$  gli algoritmi di apprendimento eseguono delle approssimazioni per velocizzare i calcoli. Ad esempio le reti neurali cercano un minimo locale di  $E(f; \mathcal{D}_n)$ . Questo approccio ottimizza la velocità ma potrebbe non trovare la soluzione corretta. Chiameremo  $\hat{f}_{\mathcal{H}}^*(\mathcal{D}_n)$  tale risultato.

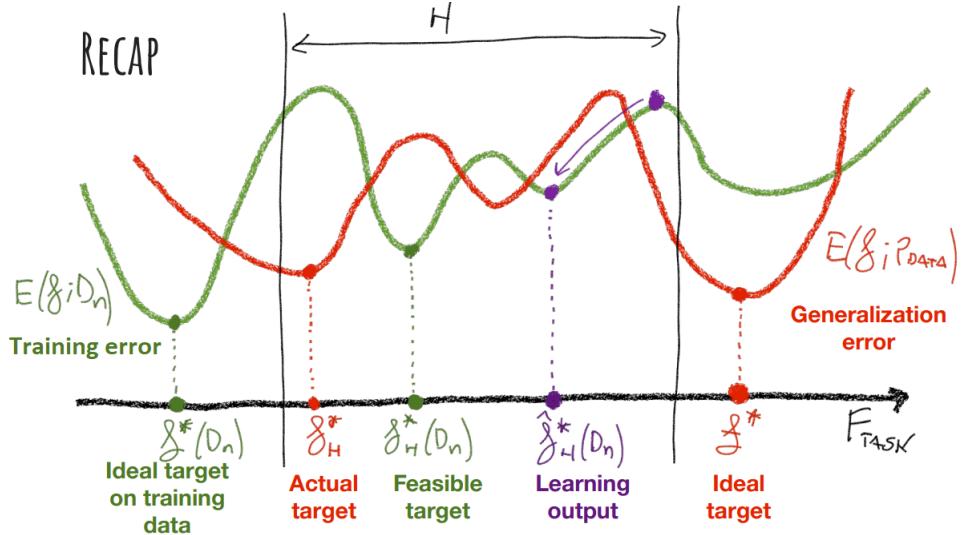


Figura 1.7: Riassunto

## Overfitting e Underfitting

**Definizione 1.16** (Overfitting). Overfitting si verifica quando il gap tra il training set e il validation set è troppo grande. Si avrà infatti un grosso errore nel momento in cui avverrà la generalizzazione.

**Definizione 1.17** (Underfitting). Underfitting si verifica quando si ha un training error troppo elevato.

**Regularization** Per trovare la soluzione migliore senza incappare in Underfitting o Overfitting ridefiniamo la funzione di errore penalizzando le soluzioni troppo complesse.

**Definizione 1.18** (Funzione di errore regolarizzata). Definiamo funzione di errore regolarizzata

$$E_{\text{reg}}(f; \mathcal{D}_n) = E(f; \mathcal{D}_n) + \lambda_n \Omega(f) \quad (1.12)$$

dove  $\lambda_n \in \mathbb{R}$  è detto **trade off parameter** e  $\Omega(f)$  (**funzione di regolarità**) è il termine che “giudica” la complessità delle soluzioni ( $f$ ).

### 1.4.5 Esempio: Polynomial curve fitting

L’obiettivo è quello di trovare una funzione polinomiale che descrivi una serie di dati iniziali. Definiamo i modelli (soluzioni)

$$f_w(x) \doteq \sum_{j=0}^M w_j x^j \quad (1.13)$$

dove  $M \in \mathbb{N}$  corrisponde al grado della funzione e  $w = \{w_0, \dots, w_M\}$ . Possiamo quindi ricavare uno spazio delle ipotesi

$$\mathcal{H}_M = \{f_w \text{ t.c. } w \in \mathbb{R}^M\} \quad (1.14)$$

Supponiamo di avere un insieme di dati iniziali

$$\mathcal{D}_n = \{z_i = (x_i, y_i) \text{ t.c. } i \in \{0, \dots, n\}\} \quad (1.15)$$

Definiamo dunque una funzione di errore puntuale sui dati di allenamento (training error)

$$E(f_w; \mathcal{D}_n) \doteq \frac{1}{n} \sum_{i=1}^n [f_w(x_i) - y_i]^2 \quad (1.16)$$

NB: questo corrisponde a definire  $l(f; z_i) \doteq [f(x_i) - y_i]^2$ .

Minimizzando questa funzione saremmo in grado di trovare un modello adatto ai nostri dati iniziali. Tuttavia

però potremmo incappare in problemi di Underfitting, nel caso in cui dovessimo approssimare troppo il risultato, o Overfitting causato da un'estremizzazione nella ricerca del minimo di  $E(f; \mathcal{D}_n)$ .

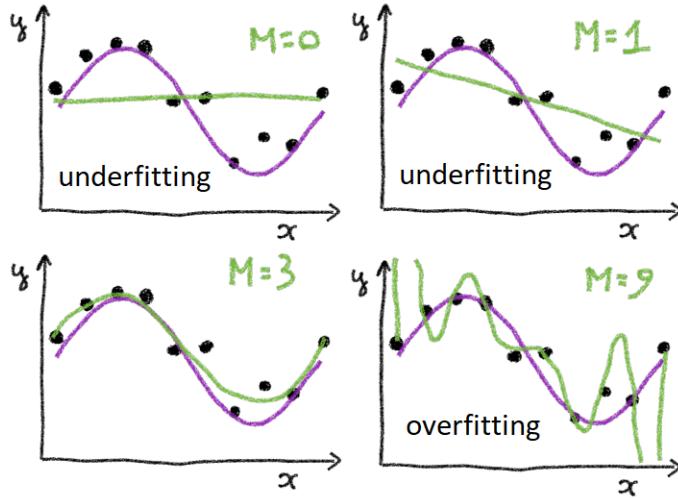


Figura 1.8: Underfitting e Overfitting al variare di  $M$

Per evitare l'Overfitting definiamo la funzione di regolarità penalizzando le soluzioni con coefficienti grandi:

$$\Omega(f) \doteq \|w\|^2 = \sum_i w_i^2 \text{ e } \lambda_n \doteq \frac{\lambda}{n}, \lambda \in \mathbb{R} \quad (1.17)$$

Otteniamo quindi

$$E_{\text{reg}}(f_w; \mathcal{D}_n) = \frac{1}{n} \sum_{i=1}^n n[f_w(x_i) - y_i]^2 + \frac{\lambda}{n} \|w\|^2 \quad (1.18)$$

NB: ovviamente un altro modo per minimizzare il generalization error è quello di aumentare il numero di dati iniziali, infatti

$$\lim_{n \rightarrow \infty} E(f; \mathcal{D}_n) = E(f; p_{\text{data}}) \quad (1.19)$$

# Capitolo 2

## Supervised learning models

### 2.1 K-Nearest neighbor

Algortimo basato su Supervised Learning utilizzato per la classificazione. L'idea è quella di classificare un elemento  $d$  in questo modo:

- si trovano i  $k$  elementi più vicini a  $d$ .
- si sceglie come label di  $d$  la label che compare più volte tra i suoi vicini.

NB: tecnica resistente a dati leggermente alterati (**noisy data**).

#### 2.1.1 Come misuriamo la distanza?

**Definizione 2.1** (Distanza euclidea).

$$d(P, Q) \doteq \|\overline{PQ}\|, \forall P, Q \in \mathbb{E}^n \quad (2.1)$$

La distanza euclidea funziona però solo quando le features che descrivono gli esempi sono comparabili. Tuttavia quando non lo sono possiamo standardizzarle, in questo modo avranno tutte la stessa importanza.

#### Standardization & Scaling

- **Standardization o Z-score normalization.** “Scaliamo” i dati in modo da avere media pari a 0 e deviazione standard 1.

$$x_{\text{norm}} = \frac{x - \mu}{\sigma} \quad (2.2)$$

dove  $\mu$  è la media e  $\sigma$  la deviazione standard.

- **Min-Max scaling.** “Scaliamo” i dati all'interno di un range contenuto in  $[0, 1]$ ,

$$x_{\text{morm}} = \frac{x - x_{\min}}{x_{\max} - x_{\min}} \quad (2.3)$$

#### Distanza e similiarità

**Definizione 2.2** (Similiarità). Misura numerica di quanto due entità si assomigliano. Più è alta più due oggetti si assomigliano.

**Definizione 2.3** (Distanza di Minkovski). La distanza di Minkovski è una generalizzazione della distanza euclidea.

$$D(a, b) = \sum_{k=1}^p |a_k - b_k|^{\frac{1}{r}} \quad (2.4)$$

dove  $r \in [1, +\infty]$ , e  $p$  è la dimensione dello spazio

- $r = 1$  Distanza di **Manhattan** (norma  $L^1$ )
- $r = 2$  Distanza euclidea
- $r = \infty$  Norma infinito  $L^\infty$

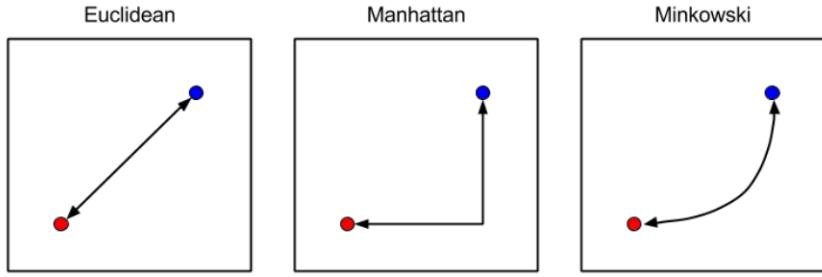


Figura 2.1: Distanze

**Definizione 2.4** (Cosine Similarity). Possiamo definire la similità tra due vettori  $d_1, d_2$

$$\text{sim}(d_1, d_2) = \frac{(d_1 \cdot d_2)}{\|d_1\| \|d_2\|} \quad (2.5)$$

### 2.1.2 Decision Boundaries

**Definizione 2.5** (Decision Boundaries). Luoghi dello spazio delle features dove la classificazione di un punto (esempio) cambia. k-NN genera dei decision boundaries definiti localmente.

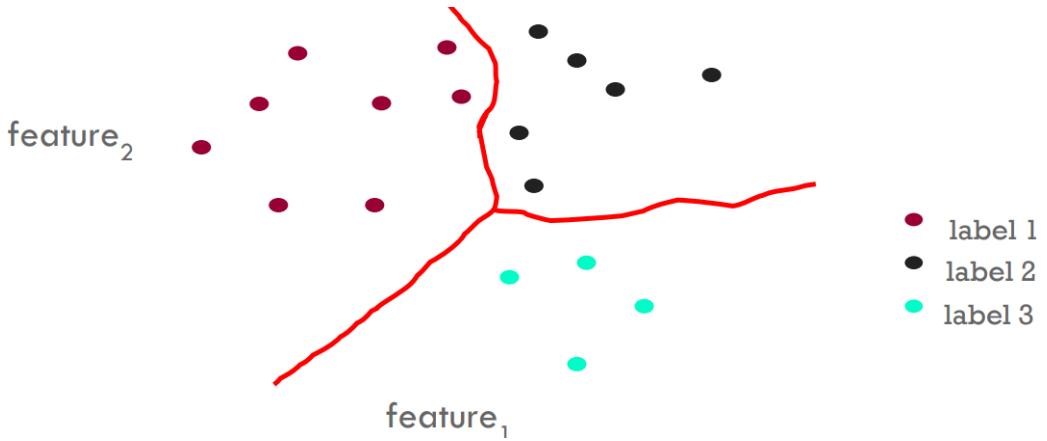


Figura 2.2: Esempio decision boundaries con k-NN

#### Come scegliamo il valore di k?

**Definizione 2.6** (Hyper Parameter). Parametro definito a priori usato per controllare il processo di apprendimento.

k è un hyper parameter, per scegliere il suo valore abbiamo varie opzioni:

- *k t.c. k* minimizza gli errori rispettivamente al validation set.
- scelta standard ( $k=3,5,7$ ). Scegliamo k dispari in modo da evitare pareggi.
- cross validation: divisione del training set in più sottogruppi.
- rule of thumb:  $k < \sqrt{n}$  dove  $n$  è il numero di esempi.

#### k-NN Pesato

Come k-NN tiene conto di k vicini ma assegna ad ognuno di essi un importanza (peso) diversa a seconda della distanza a cui si trovano.

NB: k-NN è un algoritmo di tipo **lazy learner**.

### 2.1.3 Curse of Dimensionality

In spazi di grandi dimensioni quasi tutti i punti sono lontani fra loro. Infatti la grandezza di uno spazio cresce esponenzialmente rispetto al numero di dimensioni. Questo implica che anche il data set dovrebbe crescere esponenzialmente per mantenere la stessa densità.

Siccome il successo di k-NN è dipendente dall'avere un data set denso, il fatto di avere spazi di grandi dimensioni (ovvero molteplici features) ci obbligherà ad avere un enorme data set, cosa non sempre possibile (costi, disponibilità).

### 2.1.4 Costo computazionale

Senza nessuna elaborazione dei dati (lazy learning) dovremmo ogni volta calcolare la distanza tra tutti gli  $N$  esempi. Avermo quindi un costo

$$O(aN) \quad (2.6)$$

dove  $a$  è il numero di features.

Per migliorare la complessità possiamo eseguire un elaborazione a priori generando delle strutture dati basate sugli alberi (**k-d trees**).

**Definizione 2.7** (k-d tree). Albero binario nel quale ogni nodo è un punto  $x \in \mathbb{R}^k$ .

### 2.1.5 Riassumendo

k-NN è un algoritmo basato su modello non parametrico. È adatto per risolvere problemi di classificazione quando:

- gli esempi sono mappati come punti in  $\mathbb{R}^n$
- per ogni esempio ci sono al massimo 20 features (Dimensionality Curse)
- sì è a disposizione di un grande training data

**Vantaggi:**

- facile da programmare
- non necessita di essere allenato
- può imparare funzioni di target complicate
- può essere molto accurato nella classificazione, in certi casi persino migliore di modelli più complessi

**Svantaggi:**

- difficoltà nello scegliere la misura e il valore di  $k$  adatti
- dimensionality curse
- alto costo computazionale: bisogna visionare tutti i dati per ogni classificazione
- facilmente ingannabile da features non rilevanti

## 2.2 Linear Model

Per migliorare l'efficienza degli algoritmi di apprendimento si possono fare assunzioni sui dati (**model assumptions**). Se le assunzioni sono vere si avrà una performance notevolmente migliorata ma nel caso in cui siano false il modello fallirà.

**Definizione 2.8** (Bias). Il Bias di un modello rappresenta quanto le assunzioni sui dati siano "forti". **Low-Bias** models fanno assunzioni minime sui dati (es. k-NN), quelli **High-Bias** fanno invece assunzioni forti.

NB: k-NN non fa nessuna assunzione sui dati. Si limita ad assumere che la vicinanza tra due esempi sia in relazione con l'appartenenza a delle classi.

**Definizione 2.9** (Linearmente separabile). Un insieme di dati si dice linearmente separabile se esistono iperpiani che possano dividere le classi (nel caso  $n=2$  rette).

I modelli lineari sono modelli che assumono che i dati siano linearmente separabili (High-Bias).

### 2.2.1 Definizione metodi lineari

Sia  $z = (f_1, \dots, f_n) \in \mathbb{R}^n$  un punto (esempio), definiamo peso  $w = (w_1, \dots, w_n)$ . Ogni peso  $w$  definisce un iperpiano (retta se  $n=2$ )

$$0 = w_1 f_1 + \dots + w_n f_n + b, \quad b \in \mathbb{R} \quad (2.7)$$

Un metodo per classificare i dati sfruttando questi iperpiani nel caso la label sia  $-1/1$  è verificare il segno di

$$b + \sum_{i=1}^n w_i f_i \begin{cases} > 0 \Rightarrow 1 \text{ (positive example)} \\ < 0 \Rightarrow -1 \text{ (negative example)} \end{cases} \quad (2.8)$$

## 2.2.2 Perceptron learning algorithm

Per scegliere il valore di  $w$  e  $b$  possiamo usare il seguente algoritmo

```

repeat until convergence (or for some # of iterations):
    for each training example ( $f_1, f_2, \dots, f_n$ , label):
        prediction =  $b + \sum_{i=1}^n w_i f_i$ 
        if prediction is different from label
            for each  $w_i$ :
                 $w_i = w_i + f_i * \text{label}$ 
             $b = b + \text{label}$ 

```

Consideriamo la **predizione diversa dalla label se hanno segno discorde**.

**Convergenza** l'algoritmo converge nel caso in cui l'assunzione che i dati siano linearmente separabili sia vera.

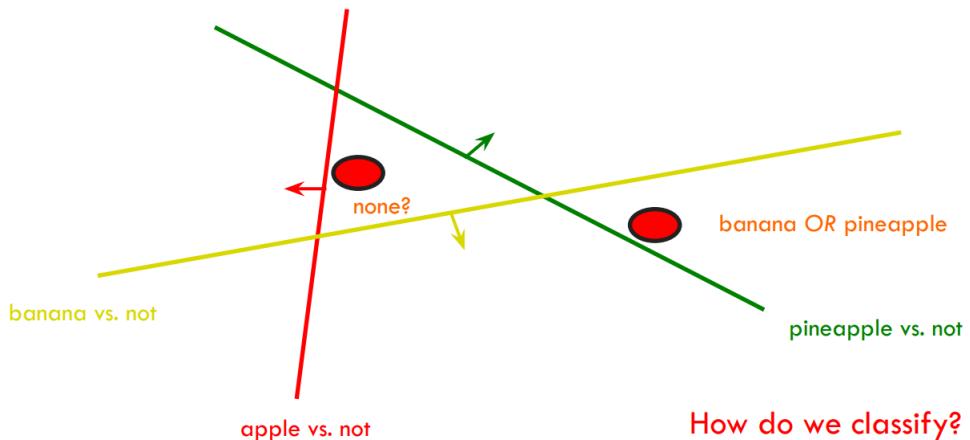
## 2.2.3 Multiclass classification

I modelli lineari possono essere utilizzati anche per classificazioni multi-classe. Esistono due metodi principali per fare ciò.

### One vs All (OVA)

Date  $L$  label definiamo per ognuna di esse un problema di classificazione binaria e escludiamo una categoria alla volta. In pratica apprendiamo  $L$  differenti modelli di classificazione binaria differenti.

Possono però esistere esempi che non rientrano nei "bordi" di più categorie o in nessuna di esse.



Per classificare tali esempi, se il classifier offre un valore per la confidenza:

- scegliamo la confidenza positiva massima se presente;
- altrimenti scegliamo la confidenza negativa minima.

### All vs All (AVA)

Date  $L$  label alleniamo  $L(L - 1)/2$  classifiers

$$F_{ij}, 1 \leq i < j \leq L \text{ è il classifier che descrimina la classe } i \text{ dalla classe } j \quad (2.9)$$

Dato un esempio in input,  $\forall F_{ij}$  se  $F_{ij}$  predice un risultato positivo la classe  $i$  prende un voto, altrimenti lo prende la classe  $j$ .

La classe che al termine del processo avrà più voti sarà quella "vincente".

## OVA vs AVA

AVA deve apprendere più classificatori ma è usualmente allenato su una mole di dati minore. Questo rende il training time di AVA inferiore a quello di OVA. Tuttavia il test time di OVA è più corto in quanto AVA deve calcolare più classifier. Inoltre il maggior numero di classificatori comporta una maggiore probabilità di errore. Per questi motivi usualmente la scelta più comune per la Multiclass classification è OVA.

### 2.2.4 Gradient Descent

Un altro metodo per determinare  $w$  nei modelli lineari è quello di cercare di minimizzare una funzione di perdita. Dato un modello lineare con  $n$  esempi  $z_i = (x_{i,1}, \dots, x_{i,m}) \in \mathbb{R}^m$ ,  $i = 1, \dots, n$  con label  $y_i$  definiamo:

**Definizione 2.10** (0/1 Loss function).

$$\sum_{i=1}^n 1[y_i(w \cdot x_i + b) \leq 0] \quad (2.10)$$

numero di esempi classificati erroneamente.

Per ottimizzare la classificazione sul training set basterà quindi trovare il minimo di tale funzione

$$\operatorname{argmin}_{w,b} \sum_{i=1}^n 1[y_i(w \cdot x_i + b) \leq 0] \quad (2.11)$$

Minimizzare tale funzione è un problema NP-Hard. Per questo motivo si cerca una funzione **convessa** che sostituisca la funzione di partenza e si minimizza quella. Tale funzione è chiamata **surrogate loss function**.

Un esempio di surrogate loss function è

$$\text{loss}(w) \doteq \sum_{i=1}^n e^{-y_i(w \cdot x_i + b)} \quad (2.12)$$

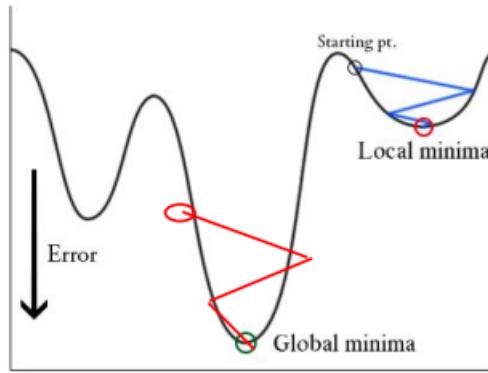
Una volta trovata la funzione convessa da minimizzare è possibile trovarne il minimo con il metodo del **Gradient Descent**.

**Definizione 2.11** (Gradient Descent). Scegliere dei pesi iniziali come starting point ( $w_0$ ) e ripetere fino a che la funzione non decresce in nessuna direzione:

- scegliere una dimensione ( $i$ );
- muoversi di una piccola quantità verso la direzione di decrescita per quella dimensione (usando la derivata).

$$w_{j+1} = w_j - \frac{d}{dw_{j,i}} \text{loss}(w_j) \quad (2.13)$$

Nel caso in cui la loss function non sia convessa troveremo un **minimo locale** anziché il minimo assoluto.



**Punti di sella** In certi punti il gradiente può risultare nullo nonostante non si tratti di un punto di minimo. Questo può portare a classificazioni errate.

## 2.2.5 Regularization

Con il gradient descent siamo in grado di trovare il minimo per la loss function ottimizzando così la classificazione. Dobbiamo però evitare l'overfitting. Per farlo aggiungiamo alla loss function un **regolarizzatore**. Questo perché non è detto che i parametri che minimizzano la loss function per il training test siano quelli che migliori per il test set (problema di generalizzazione).

$$\operatorname{argmin}_{w,b} \sum_{i=1}^n \text{loss}(w) + \lambda \text{regularizer}(w,b) \quad (2.14)$$

Solitamente il regularizer penalizza soluzioni troppo complesse. In questo caso penalizziamo pesi  $w$  troppo "grandi". Possiamo scegliere come regularizer le p-norme:

$$\text{regularizer}(w,b) = r(w,b) = \sqrt[p]{\sum_{i=1}^m |w_j|^p} = \|w\|_p \quad (2.15)$$

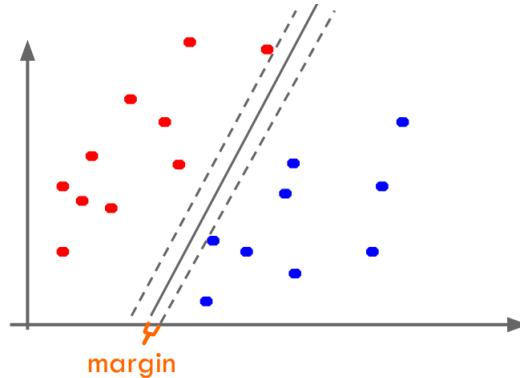
Notiamo che maggiori valori di  $p$  svantaggiano maggiormente pesi "grandi". Inoltre essendo tutte le p-norme **convesse** vale

$$\text{loss function convessa} \Rightarrow \text{loss function regolarizzata convessa} \quad (2.16)$$

## 2.3 Support Vector Machines

I modelli lineari visti in precedenza cercano un iperpiano che separa i dati ma nel caso di dati non linearmente separabili tali algoritmi non convergono ad una soluzione. Per ovviare a questo problema introduciamo le support vector machines.

**Definizione 2.12** (Margin). Il margine di un iperpiano classificatore è la distanza da esso al punto più vicino di qualsiasi classe. I **large margin classifier** puntano a massimizzare tale margine.



**Definizione 2.13** (Support vectors). Per ogni iperpiano separatore chiamiamo support vectors i punti più vicini ad esso.

**Idea** per massimizzare i margini possiamo limitarci a considerare i support vectors.

### Misurare i margini

La distanza di un punto  $x$  da un iperpiano  $0 = w_1x_1 + \dots + w_mx_m + b$  risulta

$$d(x) = \frac{wx + b}{\|w\|_2} \quad (2.17)$$

Il margine viene quindi calcolato come  $\frac{c}{\|w\|}$  traslando l'iperpiano in modo che  $b = 0$ , inoltre essendo  $c, w$  strettamente collegati si può assumere  $c = 1$ . Definiamo dunque la funzione

$$\text{margin}(w,b) = \frac{1}{\|w\|} \quad (2.18)$$

### 2.3.1 Hard margin classification

Alla richiesta di avere la distanza massima dei support vectors dai margini dobbiamo aggiungerci il fatto che essi siano all'interno del margine, ovvero che la loro classificazione sia corretta. Cerchiamo quindi  $w, b$  con le seguenti equazioni:

$$\operatorname{argmax}_{w,b}(\text{margin}(w, b)) \quad (2.19)$$

$$y_i(w \cdot x_i + b) \geq 1, \forall i \quad (2.20)$$

Notiamo che la soluzione della (2.19) equivale a quella di

$$\operatorname{argmin}_{w,b} \|w\| \quad (2.21)$$

$$\operatorname{argmin}_{w,b} \|w\|^2 = \operatorname{argmin}_{w,b} \left( \sum_{i=1}^m |w_i|^2 \right) \quad (2.22)$$

Quindi ci rimane da risolvere un **problema di ottimizzazione quadrattico**.

### 2.3.2 Soft margin classification

La soft margin classification permette di usare SVM anche quando i punti non sono linearmente separabili. Per farlo si permettono predizioni sbagliate aggiungendo però una penalità per ognuna di esse (**slack penalties**). Le equazioni da soddisfare diventano pertanto:

$$\operatorname{argmin}_{w,b} \left( \|w\|^2 + C \sum_{i=1}^n \zeta_i \right) \quad (2.23)$$

$$y_i(w \cdot x_i + b) \geq 1 - \zeta_i, \forall i \quad (2.24)$$

dove  $\zeta_i \geq 0$  indica quanto sia "lontano"  $x_i$  dall'essere corretto,  $C > 0$  costante.

Il parametro  $C$  può essere adattato per evitare overfitting:

- $C$  piccolo permette di ignorare molti errori → margini ampi
- $C$  grande rende gli errori molto difficili da ignorare → margini ristretti
- $C = \infty$  equivale a richiedere le constraint per gli hard margin

Anche in questo caso dobbiamo risolvere un **problema di ottimizzazione quadrattico**.

### 2.3.3 Risolvere il problema di ottimizzazione

Per risolvere il problema di ottimizzazione quadrattico possiamo utilizzare la **surrogate function di Hinge** applicandola a  $C_i$

$$\text{loss}(y, y') = \max(0, 1 - yy') \quad (\text{funzione di Hinge})$$

$$C_i = 1 - y_i(w \cdot x_i + b) = \max(0, 1 - yy') \quad (2.25)$$

trasformando il problema di ottimizzazione nella nuova funzione obiettivo

$$\operatorname{argmin}_{w,b} \left( \|w\|^2 + C \sum_{i=1}^n \max\{0, 1 - y_i(w \cdot x_i + b)\} \right) \quad (2.26)$$

Questa funzione può essere considerata come problema di **Gradient Descent** con **Hinge loss function** e funzione regolarizzatrice  $\|w\|^2$ .

Una delle più importanti applicazioni di SVM è **Pedestrian detection** (Computer Vision).

## 2.4 Decision trees

Dato  $X$  spazio di input,  $Y$  spazio di output con le possibili labels. Vogliamo trovare una mappa

$$h : X \rightarrow Y, h \in Y^X \quad (2.27)$$

avendo a disposizione un training set  $\{\langle x_i, y_i \rangle\}_{i=1}^n$

**Definizione 2.14** (Decision tree). Modello di predizione ad albero composto da:

- **non-terminal nodes**: nodi aventi 2 o più figli che implementano funzione di routing

- **leaf nodes:** nodi foglia che implementano una funzione di predizione

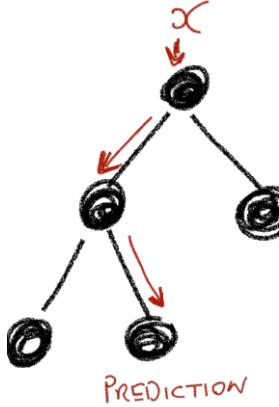


Figura 2.3: decision tree

Nel nostro caso assumeremo che gli alberi siano binari.

Ogni nodo non terminale  $\text{Node}(\phi, t_L, t_R)$  è composto da un figlio sinistro  $t_L$ , un figlio destro  $t_R$  e una **routing function**

$$\phi : X \ni x \mapsto t \in \{t_L, t_R\} \quad (2.28)$$

Ogni nodo foglia  $\text{Leaf}(h)$  contiene una funzione predizione  $h \in \mathcal{F}_{\text{task}}$  tipicamente costante. Quando  $x$  raggiunge una foglia la predizione sarà data da  $h(x)$ .

L'albero è definito tramite una funzione ricorsiva  $f_t : X \rightarrow Y$ , chiamata inizialmente sul nodo radice, definita:

$$f_t(x) = \begin{cases} h(x) & \text{se } t = \text{Leaf}(h) \\ f_{t_{\phi(x)}}(x) & \text{se } t = \text{Node}(\phi, t_L, t_R) \end{cases} \quad (2.29)$$

#### 2.4.1 Learning algorithm

Dato un training set  $\mathcal{D}_n = \{z_1, \dots, z_n\}$  cerchiamo  $f_{t^*}$  t.c.

$$t^* \in \operatorname{argmin}_{t \in \tau} E(f_t; \mathcal{D}_n) \quad (2.30)$$

dove  $\tau$  è l'insieme dei decision trees.

Il problema di ottimizzazione è semplice se non imponiamo vincoli altrimenti potrebbe diventare NP-Hard. Una possibile strada è quella di utilizzare una strategia **greedy** scegliendo

$$E(f_t; \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{z \in \mathcal{D}} l(f; z) \quad (2.31)$$

dove  $l(f; z)$  è la pointwise loss. Il training viene fatto in batch mode.

**Definizione 2.15** (Batch learning). Tipo di allenamento per ML incapace di apprendere incrementalmente. Se si vogliono aggiungere dati bisogna allenare l'algoritmo da capo.

Fissato un sett di predizioni  $\mathcal{H}_{\text{leaf}} \subset \mathcal{F}_{\text{task}}$  (**leaf predictors**). Fissato un set di possibili routing functions  $\phi \subset \{t_L, t_R\}^X$ . Dobbiamo stabilire dei criteri per decidere se crescere una foglia o un nodo non terminale.

**Definizione 2.16** (Impurità). Dato il training set  $\mathcal{D}_n$  il leaf predictor ottimale risulta

$$h_{\mathcal{D}}^* \in \operatorname{argmin}_{h \in \mathcal{H}_{\text{leaf}}} E(h; \mathcal{D}) \quad (2.32)$$

L'errore ottimale è chiamato **impurità**

$$I(\mathcal{D}) = E(h_{\mathcal{D}}^*; \mathcal{D}) \quad (\text{misura di impurità})$$

I criteri per crescere una foglia  $\text{Leaf}(h_{\mathcal{D}}^*)$  risultano quindi

$$I(\mathcal{D}) < \epsilon \in \mathbb{R}_+ \quad (\text{criterio di purezza})$$

$$|\mathcal{D}| < k \in \mathbb{N} \quad (\text{criterio di cardinalità})$$

Se nessuno di questi criteri viene soddisfatto si procede a crescere un nodo.

**Definizione 2.17** (Impurità di una split function). Definiamo **impurità della split function**  $\phi$  definita

$$I_\phi(\mathcal{D}) \doteq \sum_{d \in \{L, R\}} \frac{|\mathcal{D}_d^\phi|}{|\mathcal{D}|} I(\mathcal{D}_d^\phi) \quad (2.33)$$

dove  $\mathcal{D}_d^\phi \doteq \{(x, y) \in \mathcal{D} \text{ t.c. } \phi(x) = d\}$

**Definizione 2.18** (Split function ottimale). Definiamo split function ottimale

$$\phi_D^* \in \operatorname{argmin}_{\phi \in \Phi} I_\phi(\mathcal{D}) \quad (2.34)$$

L'**algoritmo di apprendimento** può essere quindi riassunto dalla formula

$$\operatorname{Grow}(\mathcal{D}) = \begin{cases} \operatorname{Leaf}(h_D^*) & \text{se è soddisfatto un criterio per crescere una foglia} \\ \operatorname{Node}(\phi_D^*, \operatorname{Grow}(\mathcal{D}_L^*), \operatorname{Grow}(\mathcal{D}_R^*)) & \text{altrimenti} \end{cases} \quad (2.35)$$

### Scelta della misura di impurità

Sia  $z = (x, y)$ , imponendo  $l(f; z) = -\log f_y(x)$ ,  $\mathcal{H}_{\text{leaf}} = \cup_{\pi \in Y} \pi^X$ . La misura di impurità sarà l'**entropia**

$$I(\mathcal{D}) = - \sum_{y \in Y} \frac{|\mathcal{D}^y|}{|\mathcal{D}|} \log \frac{|\mathcal{D}^y|}{|\mathcal{D}|} \quad (2.36)$$

notiamo infatti che  $\frac{|\mathcal{D}^y|}{|\mathcal{D}|} = p_y$  corrisponde alla probabilità della classe  $y$ .

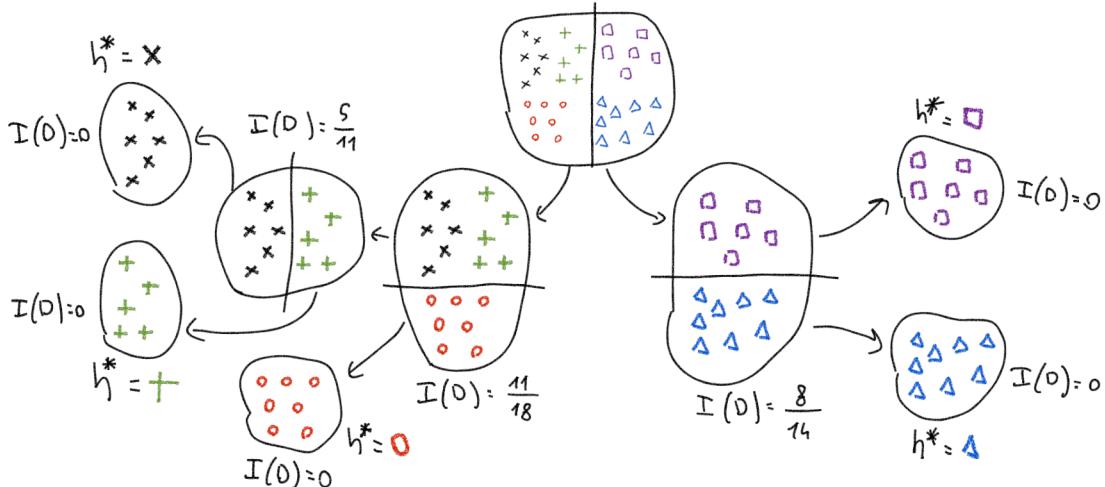


Figura 2.4: Esempio learning algorithm

### 2.4.2 Overfitting

I decision tree sono flessibili e facilmente adattabili al training set. Questo comporta un alto rischio di overfitting. Se infatti i training data fossero perturbati o nel caso di presenza di attributi non rilevanti si incapperebbe sicuramente in overfitting. Una soluzione ovvia è quella di eliminare manualmente tali dati, tuttavia non è sempre possibile.

Una tecnica usata per ridurre l'overfitting riducendo la complessità è il **pruning**, questa tecnica consiste nel rimpiazzare un intero sottoalbero con un singolo nodo foglia.

### 2.4.3 Riassunto

#### Punti di forza

- veloce e semplice da implementare
- può essere convertito in regole

#### punti deboli

- richiede vettori di features di lunghezza fissata
- non è incrementale (batch mode)

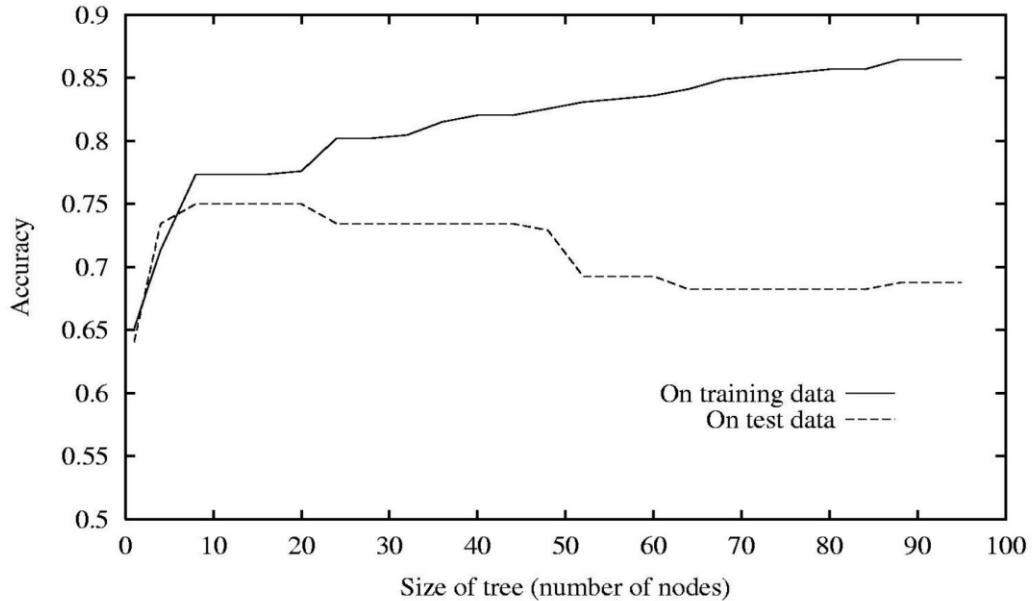


Figura 2.5: Grafico che mostra la tendenza dei decision tree a incappare in overfitting

#### 2.4.4 Random Forests

Una random forest è un insieme di decision trees:

- ogni albero è allenato partendo da una porzione differente del training set;
- le split function sono ottimizzate su un insieme di features randomico;
- la predizione finale è data dalla media delle predizioni ottenute dai vari alberi  $\mathcal{Q} = \{t_1, \dots, t_T\}$

$$f_{\mathcal{Q}}(x) = \frac{1}{T} \sum_{j=1}^T f_{t_j}(x) \quad (2.37)$$

Un esempio di applicazione delle random forest è il **kinect** della Xbox.

# Capitolo 3

## Neural network

**Definizione 3.1** (Neurone artificiale). Un neurone artificiale è una funzione non lineare parametrizzata con un output range ristretto

Le reti neurali si basano su un algoritmo che abbiamo già incontrato ovvero **perceptron** (2.2.2). Il perceptron può essere considerato come un **neurone artificiale**. Sia  $x = x_0, \dots, x_n$  l'esempio testato

$$\hat{y} = h \left( \sum_i w_i x_i + w_0 \right) = h(w^T x + w_0) \quad (3.1)$$

dove  $h(x)$  è definita:

$$h(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (3.2)$$

$\hat{y} \in \{0, 1\}$  sarà quindi il risultato calcolato da perceptron.

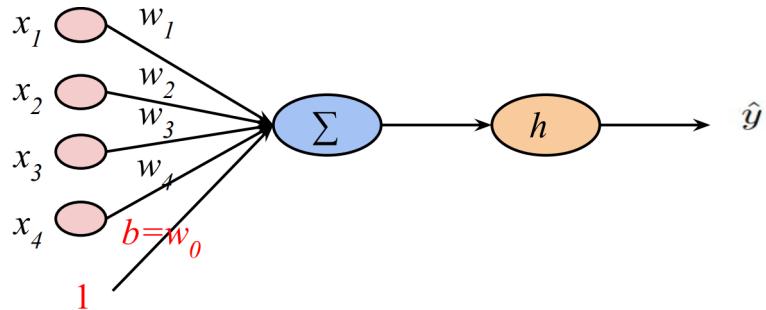


Figura 3.1: Schema funzionamento perceptron

Come sappiamo però perceptron è in grado di risolvere **solo problemi linearmente separabili** (2.9). Ad esempio le tavole di verità di AND e OR sono problemi linearmente separabili, al contrario di quella di XOR.

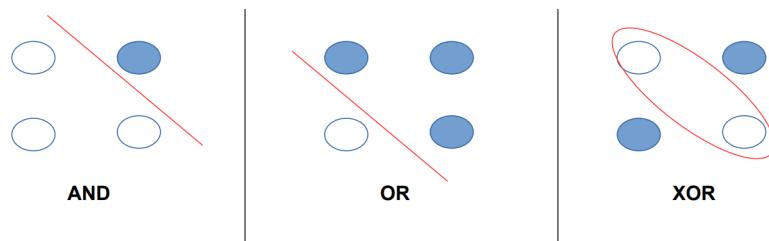


Figura 3.2: problemi linearmente separabili e non

Minsky e Papert nel 1969 capiscono che si può risolvere XOR con un **Multi-Layer Perceptron (MLP)**. L'idea è usare dei neuroni artificiali densamente connessi per realizzare composizioni di funzioni non lineari. Questa tecnica può essere impiegata in qualsiasi task di supervised learning (es. classification, regression, ecc.).

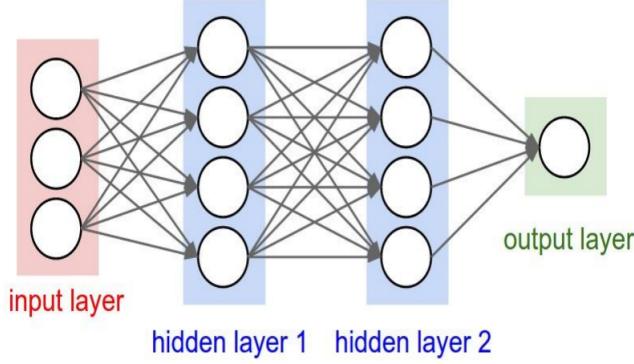


Figura 3.3: multi-layer perceptron

### 3.1 Feed-Forward neural network

In una Feed-Forward neural network le informazioni sono propagate dagli input agli output, si tratta quindi di un grafico diretto aciclico (DAG). Gli **hidden layer** calcolano una rappresentazione intermedia degli esempi in input. Ad esempio in un **modello semplificato con un singolo layer nascosto** si computeranno

$$z_j = \sum_{i=0}^n w_{i,j}^{(1)} x_i + w_{0,j}^{(1)} \quad (3.3)$$

$$\hat{y}_k = f \left( \sum_{i=0}^n w_{i,k}^{(2)} h(z_i) + w_{0,k}^{(2)} \right) \quad (3.4)$$

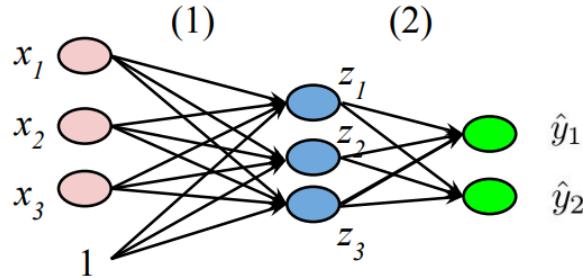


Figura 3.4: esempio feed-forward con un singolo layer nascosto

#### 3.1.1 Definizione

L'obiettivo di Feed-Forward è approssimare una funzione ideale  $f^* : X \rightarrow Y$  minimizzando gli errori. Definiamo quindi una funzione parametrizzata  $f(x_i; \Theta)$  e apprendiamo i parametri in modo da ottenere una buona approssimazione di  $f^*$ . L'informazione viaggia attraverso gli hidden-layer che computano di volta in volta l'output,  $f$  sarà quindi composizione di più funzioni

$$f(x) = f^{(k)}(f^{(k-1)}(\dots f^{(1)}(x))) \quad (3.5)$$

dove  $f^{(i)}$  rappresenta l' $i$ -esimo layer. Il layer finale è chiamato **output-layer**

#### 3.1.2 Training

Ottimizzare  $\Theta$  in modo da rendere  $f(x; \Theta)$  il più simile possibile a  $f^*(x)$ .

Specifichiamo solo l'output dell'output layer, e eseguiamo il training con **gradient descent** in modo simile a quello visto nel capitolo precedente. Essendo però il problema **non convesso** la **convergenza non è garantita**. Per applicare gradient descent dobbiamo quindi specificare:

- una funzione di costo (cost function)
- la forma dell'output
- una funzione di attivazione  $h(z)$  (activation function)
- l'architettura (numero di layers)
- un ottimizzatore

## Cost function

Definiamo la funzione

$$\mathcal{L} \doteq \text{dist}(f_\theta(x), y) \quad (3.6)$$

dove  $y$  sono le label degli esempi  $x$ . A questa funzione possiamo applicare le surrogate loss function viste in precedenza (es. square loss).

Per classification inoltre usualmente si convertono gli output in probabilità normalizzando la loss function (**softmax**). Un esempio di loss function è la **cross-entropy**

$$\mathcal{L}_i \doteq \sum_k y_k \log(S(l_k)) \quad (3.7)$$

dove  $l_i$  sono gli score in output e  $S(l_i)$  definita

$$S(l_i) \doteq \frac{e^{l_i}}{\sum_k e^{l_k}} \quad (3.8)$$

è la funzione che trasforma gli score in probabilità (softmax).

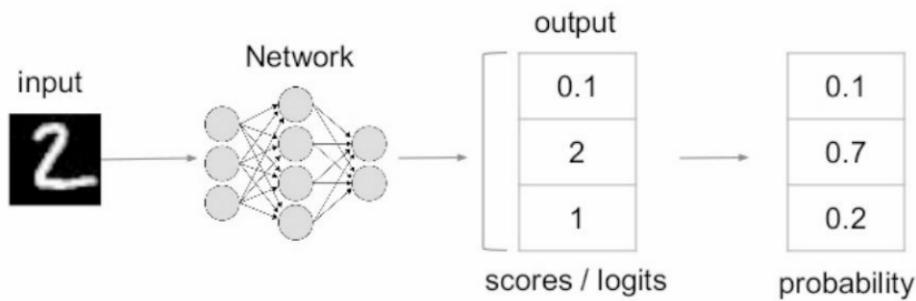


Figura 3.5: esempio feed-forward

## Forma dell'output

Date delle features  $h$  un layer di output **lineare** restituisce

$$\hat{y} = W^T h + b \quad (3.9)$$

Le unità lineari **non saturano il gradiente**, rendendo gradient descent più semplice da applicare. Per le **hidden units** non conosciamo gli output, all'interno di ognuna di esse:

- accettiamo un input  $x$
- calcoliamo una trasformazione affine  $z = W^T x + b$
- applichiamo elemento per elemento una funzione non lineare  $h(z)$  (**activation function**)
- ricaviamo l'output  $h(z)$

Resta quindi da determinare la scelta di  $h$ . *Il design delle hidden units è tuttora un area attiva della ricerca.*

## Architettura

**Teorema 3.1** (Cybenko (1989)). Network a 2 layers con output lineare e non linearità nelle hidden units possono approssimare qualsiasi funzione continua su un dominio compatto con accuratezza arbitraria.

Questo implica che indipendentemente dalla funzione che stiamo cercando di apprendere una MLP sufficientemente ampia può rappresentarla. Tuttavia non siamo certi che il nostro algoritmo di apprendimento sia in grado di impararla.

### 3.1.3 First AI winter

Purtroppo perceptron necessita di **conoscere i target desiderati (label)** per funzionare, ma non possiamo conoscerli per gli hidden layer, questo problema è conosciuto come **first AI winter**. Nel 1986 però grazie a Backpropagation si riesce a trovare una soluzione.

## 3.2 Backpropagation (1986)

Il problema di learning di MLP viene risolto con l'avvento di Backpropagation. Esso permette di processare grandi training sets sviluppando architetture complesse di neural networks. Backpropagation **fornisce infatti un metodo alternativo per calcolare il gradiente**

Il training di Backpropagation consiste in 3 passaggi:

- **forward propagation:** propagare i training data attraverso il modello computando i successivi hidden layers
- **stima dell'errore:** comparare gli outputs con le risposte corrette (label del training set) e ottenere l'**error signal**
- **Backpropagate:** propagare l'error signal e usarlo per aggiustare i pesi

Questi passaggi vengono ripetuti finché il modello non converge o fino a quando non si supera un numero massimo di iterazioni.

Dati dei training data  $\mathcal{D} = \{(x_1, y_1), \dots, (x_N, y_N)\}$  aggiustiamo i pesi della rete cercando di minimizzare la loss function.

$$\operatorname{argmin}_w \sum_i L(y_i, f(x_i; \Theta)) \quad (3.10)$$

Per fare ciò è necessario:

- scegliere una funzione di costo, solitamente la square loss;
- aggiornare i pesi di ogni layer con **gradient descent**;
- usare Backpropagation dell'errore per calcolare efficientemente il gradiente.

### 3.2.1 Training

Dai training data non sappiamo cosa dovrebbero computare le hidden units. L'idea è quella di **computare quanto velocemente cambia l'errore rispetto alla variazione di una hidden unit**. Per farlo usiamo la derivata della loss function. Ogni hidden unit può agire su molteplici output units e avere differenti effetti sull'errore, dovremmo quindi combinare tali risultati. Analizziamo il **caso specifico di un network avente un singolo layer nascosto**.

#### Step 1: feed-forward operation

Calcoliamo con feed-forward l'output (3.4)

$$\hat{y}(x; w) = f \left( \sum_{j=1}^m w_j^{(2)} h \left( \sum_{i=1}^d w_{ij}^{(1)} x_i + w_{0j}^{(1)} \right) + w_0^{(2)} \right) \quad (3.11)$$

#### Step 2: compute error and train

Definiamo la funzione di errore sul training set come

$$L(x, w) \doteq \sum_{i=1}^N \frac{1}{2} (y_i - \hat{y}(x_i, w))^2 \quad (3.12)$$

Ora è necessario calcolare la derivata di  $L$  su un singolo esempio.

Per semplicità consideriamo come modello lineare per l'output

$$\hat{y} = \sum_j w_j x_{ij} \quad (3.13)$$

Varrà quindi

$$\frac{\partial L(x_i)}{\partial w_j} = (\hat{y}_i - y_i) x_{ij} \quad (3.14)$$

dove  $y_i$  è la label attesa e  $\hat{y}_i - y_i$  è l'errore puntuale.

### Step 3: backpropagation

Definiamo  $a_t \doteq \sum_j w_{jt} z_j$

$$z_t \doteq h \left( \sum_j w_{jt} z_j \right) = h(a_t) \quad (3.15)$$

memo:  $z_j$  è il j-esimo valore in output del hidden layer.

Notiamo che la variazione di  $L$  dipende da  $w_{jt}$  solo attraverso  $a_t$  quindi, definendo  $\sigma_t \doteq \frac{\partial L}{\partial a_t}$ :

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} \frac{\partial a_t}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j = \sigma_t z_j \quad (3.16)$$

Calcoliamo quindi  $\sigma_t$ :

- per le output units lineari

$$\sigma_t = \hat{y} - y \quad (3.17)$$

- per le hidden units t che mandano il loro output alle output units nell'insieme  $S$

$$\sigma_t \doteq \frac{\partial L}{\partial a_t} = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t} = h'(a_t) \sum_{s \in S} w_{ts} \sigma_s \quad (3.18)$$

### 3.2.2 Second AI winter

Le neural network si dovettero però imbattere in altri problemi:

- impossibilità di sfruttare un grande numero di layer a causa dell'overfitting e del **vanishing gradient** (la moltiplicazione di piccoli numeri nel training causa l'annullamento del gradiente);
- mancanza di potenza di calcolo, non erano ancora state inventate le GPUs;
- mancanza di dati, non si era a disposizione di grandi dataset.

## 3.3 Scelta di un ottimizzatore

Ricordiamo la definizione di gradiente della loss function

$$\nabla_w L = \left[ \frac{\partial L}{\partial w_1}, \dots, \frac{\partial L}{\partial w_N} \right] \quad (3.19)$$

dove ogni derivata  $\frac{\partial L}{\partial w_i}$  misura quanto velocemente varia la loss function nella direzione  $i$ .

Precedentemente abbiamo visto backpropagation come metodo per computare il gradiente, analizziamo ora possibili miglioramenti (**stochastic gradient descent (SGD)**).

### 3.3.1 Batch gradient descent (BGD)

In BGD i gradienti sono computati per ogni update su tutto il training set. Questo causa un **elevato costo computazionale** ma garantisce una **grande stabilità** nella stima del gradiente. Il learning rate  $\epsilon_k$  può variare.

Batch gradient descent algorithm

**Require:** initial weights  $\theta$

**Require:** learning rate  $\epsilon_k$

**while** stopping criteria not met **do**

$$\hat{g} \leftarrow \frac{1}{N} \nabla_\theta \sum_i L(f(x^{(i)}, \theta), y^{(i)})$$

$$\theta \leftarrow \theta - \epsilon_k \hat{g}$$

▷ Calcolo il gradiente usando gli N esempi in input

▷ Applico gli aggiornamenti

**end while**

### 3.3.2 Stochastic gradient descent (SGD)

In SGD si computa il gradiente solo su un campione scelto casualmente in modo da ottenere performance migliori. Il learning rate cambia ad ogni passo, tipicamente **decade linearmente**.

**Require:** initial weights  $\theta$

**Require:** learning rate  $\epsilon_k$

**while** stopping criteria not met **do**

▷ Calcolo il gradiente su un sottoinsieme  $\{(x^{(i)}, y^{(i)})\}_{i \in I}$  del training set

$$\hat{g} \leftarrow \frac{1}{N} \nabla_{\theta} \sum_{i \in I} L(f(x^{(i)}, \theta), y^{(i)})$$

▷ Applico gli aggiornamenti

$$\theta \leftarrow \theta - \epsilon_k \hat{g}$$

**end while**

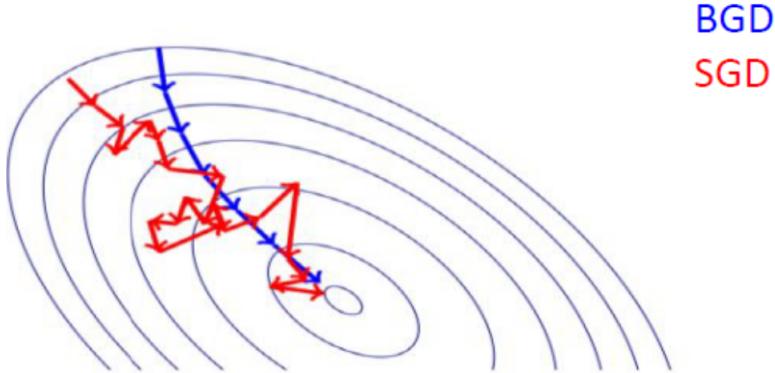


Figura 3.6: BGD vs. SGD

### 3.3.3 MiniBatches

La stima del gradiente può essere influenzata dai “rumori”. Una soluzione ovvia è quella di utilizzare piccoli set di esempio (**mini-batches**). I vantaggi sono:

- il computational time non dipende dal numero di esempi  $N$ ;
- permette il calcolo su dataset estremamente ampi;
- implementazione in parallelo;
- grazie alle GPUs si ha un ottimo runtime usando potenze di 2 come dimensione del batch.

## 3.4 Convolutional neural network (CNN)

Le convolutional neural network sono un tipo di feedforward neural network. Esse vengono impiegate quando lo **spazio di input è localmente strutturato** (es. spazialmente o temporalmente).

In particolare ogni layer organizza le sue unità in griglie 2D chiamate **feature maps**, ogni feature map è il risultato di una convoluzione.

**Definizione 3.2** (Convolutional neural network). Le CNN sono reti neurali che utilizzano la **convoluzione** al posto di una moltiplicazione tra due matrici generiche in almeno uno dei loro layers.

**Definizione 3.3** (Convoluzione). Definiamo convoluzione di  $f, g : \mathbb{R} \rightarrow \Omega$ ,  $f, g$  integrabili secondo Lebesgue, la funzione definita

$$(f * g)(t) \doteq \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau) d\tau \quad (3.20)$$

Per calcolarla all'interno dei layer useremo la formula

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (3.21)$$

NB: la convoluzione è commutativa.

Questo tipo di reti è ispirato alla corteccia visiva dei mammiferi. La convoluzione può essere infatti vista come un operazione di filtraggio per le immagini.

Per permettere tale operazione si applica una **matrice kernel** alle immagini e si determina il valore di un **pixel centrale** aggiungendo i valori pesati dei suoi vicini. Esistono kernel specializzati in **estrazione dei bordi**, **estrazione linee verticali**, ecc.

### 3.4.1 Architettura

Le CNN sono feedforward network con struttura di connettività specializzata. Tipicamente i CNN layers trasformano una matrice in input in una classe di output (predizione). Nei CNN layers si eseguono 3 tipi di operazioni:

- **convolution:**
- **non-linearity**
- **pooling**



Figura 3.7: funzionamento di un convolutional layer

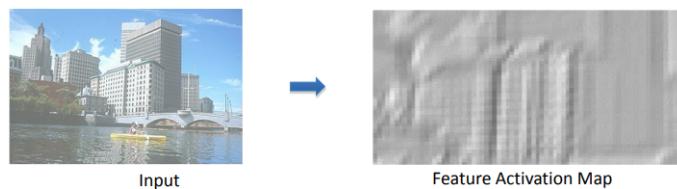


Figura 3.8: input e output di un convolutional layer

#### convolution

I convolutional layers sono i layer principali di CNN. Essi consistono in insiemi di **learned filters**, si esegue la convoluzione tra ogni filtro e i dati in input ottenendo così una **feature map multi dimensionale**.

In questo modo la rete apprenderà filtri che si attiveranno quando comparirà uno specifico pattern nei dati.

#### non-linearity

Le operazioni di non-linearity servono ad aumentare la non linearità della architettura senza avere effetto sui campi del convolution layer.

#### pooling

Il pooling (processo deterministico) consiste nel ridurre la dimensione dei dati **riducendo il numero di parametri**, questo aiuta anche a **controllare l'overfitting**. Inoltre, riducendo progressivamente la dimensione spaziale, otteniamo l'invarianza per traslazioni.

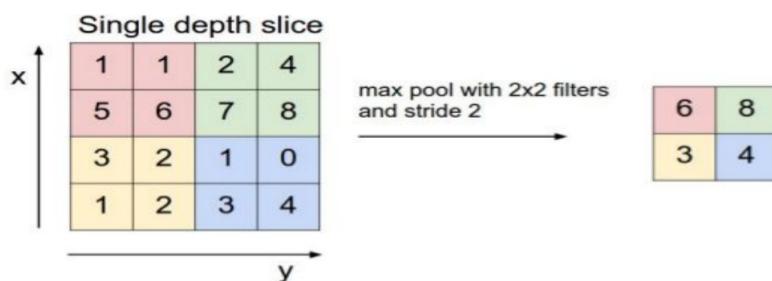


Figura 3.9: esempio max pooling

### 3.4.2 Applicazioni

Le CNN oltre che per classification possono essere utilizzate anche in:

- **semantic segmentation:** segmentazione di un immagine nei suoi elementi principali
- **structured regression:** utilizzata ad esempio nel kinect

### 3.5 Recurrent neural network (RNN)

Fino ad ora ci siamo concentrati su reti neurali con input e output di lunghezza prefissata. Consideriamo ora casi in cui l'input o l'output abbiano **lunghezza variabile**. Per risolvere problemi di questo tipo usiamo recurrent neural networks.

Per farlo introduciamo a feedforward cicli di ricorrenza, questo corrisponde a definire

$$h_t = f_W(x_t, h_{t-1}) \quad (3.22)$$

dove  $h_t$  corrisponde all'activation function all'istante  $t$ . In questo modo possiamo analizzare i dati “parola per parola” trovando così una soluzione al problema di non conoscerne la lunghezza.

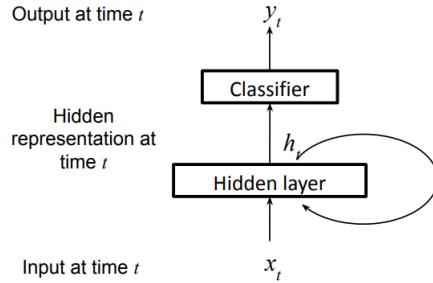


Figura 3.10: schema di una recurrent neural network

# Capitolo 4

## Unsupervised learning

Come visto nella sezione (1.3.2) unsupervised learning è usato per:

### 4.1 Dimensionality reduction

Trovare una funzione  $f \in Y^X$  che mappi ogni input  $x \in X$  in un punto  $f(x) \in Y$  e  $\dim(Y) \ll \dim(X)$ . Tale operazione è utile per:

- comprimere i dati e ridurre la loro complessità spaziale;
- velocizzare il tempo di computazione di eventuali operazioni future;
- avere una migliore visualizzazione dei dati;
- ridurre “curse of dimensionality”.

#### 4.1.1 Principal Component Analysis (PCA)

Tecnica per dimensionality reduction basata sull’idea che una **maggior varianza** nei dati comporta una **maggior ricchezza di informazioni**. Si procede quindi nel modo seguente:

- si calcolano le varianze nelle varie direzioni;
- si cambia il sistema di coordinate;
- si esclude la dimensione di varianza inferiore.

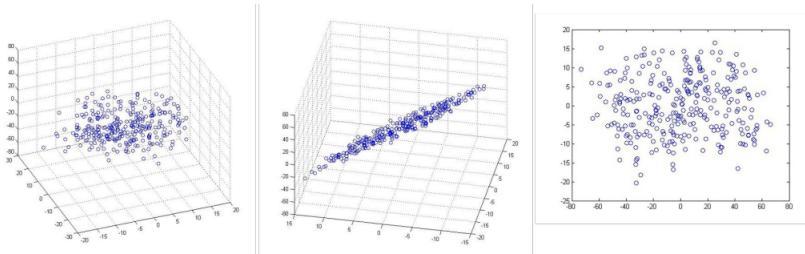


Figura 4.1: esempio da 3D a 2D

#### Calcolo della varianza lungo la direzione $w$

Consideriamo un versore  $w$ , ovvero  $ww^T = 1$  e calcoliamo

$$t_i = (x_i - c)^T w \quad (4.1)$$

dove  $t_i$  è la distanza tra  $x_i$  e un “centro” arbitrario  $c$  lungo la direzione  $w$ . Calcoliamo quindi

$$\mathbb{E}[t] = \frac{1}{n} \sum_{i=1}^n t_i = \frac{1}{n} \sum_{i=1}^n (x_i - c)^T w \quad (4.2)$$

$$\text{Var}[t] = \frac{1}{n} \sum_{i=1}^n (t_i - \mathbb{E}[t])^2 = \dots = w^T \left[ \frac{1}{n} \bar{X} \bar{X}^T \right] w = w^T C w \quad (4.3)$$

dove  $C \doteq \frac{1}{n} \bar{X} \bar{X}^T$  è la **matrice di covarianza** e  $\bar{X} \doteq [\bar{x}_1, \dots, \bar{x}_n]$ .

Per migliorare l'efficienza dei calcoli si ricava la **matrice diagonale** (quando possibile) o la forma canonica di Jordan della matrice di covarianza.

#### 4.1.2 Altre tecniche per dimensionality reduction

- **multidimensional scaling**: trovare la proiezione che preserva meglio le distanze tra i punti;
- **linear discriminant analysis (LDA)**: proietta i dati in un sottospazio cercando di massimizzare la separazione tra le classi (*anche clustering*).

## 4.2 Clustering

Trovare una funzione  $f \in \mathbb{N}^X$  che assegna ad ogni input  $x \in X$  un indice  $f(x) \in \mathbb{N}$ , l'insieme dei punti mappati nello stesso indice forma un **cluster**.

Tale operazione è utile per:

- analizzare i dati raggruppandoli tra loro;
- comprimere i dati riducendo il numero di essi, scegliendone solo alcuni per ogni gruppo.

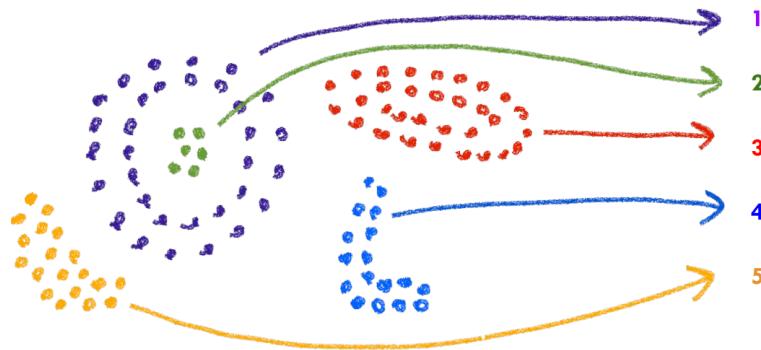


Figura 4.2: clustering

**Definizione 4.1** (Hard-Soft clustering). In hard clustering ogni esempio viene assegnato ad un singolo cluster. Viceversa in soft clustering viene assegnata agli esempi una **probabilità** di appartenenza ai clusters.

#### 4.2.1 K-Means clustering

Per eseguire il clustering di elementi non labeled è necessario scegliere il criterio di raggruppamento. Uno dei più conosciuti è K-Means (**hard clustering**).

**Definizione 4.2** (K-Means). Dati  $n$  dati in input  $X = [x_1, \dots, x_n] \in \mathbb{R}^{d \times n}$ , dove  $d$  è la dimensione dello spazio di input, fissato un **numero di cluster**  $k$ , cerchiamo la partizione di dati che **minimizza la varianza** all'interno di ogni set.

$$\operatorname{argmin}_{\mathcal{C}_1, \dots, \mathcal{C}_k} \sum_{j=1}^k V(\mathcal{C}_j) \quad (4.4)$$

$$V(\mathcal{C}_j) = \sum_{i \in \mathcal{C}_j} \|x_i - \mu_j\|^2 \quad (4.5)$$

$$\mu_j = \frac{1}{|\mathcal{C}_j|} \sum_{i \in \mathcal{C}_j} x_i \quad (4.6)$$

I  $\mu_j$  sono detti **centroidi dei clusters**.

k-means algorithm

**Require:** usare una strategia di inizializzazione per calcolare i valori iniziali di  $\mu_1, \dots, \mu_k$

**while** clusters subiscono variazioni (convergenza) **do**

▷ assegno i datapoints ai clusters con il centroide più vicino ad essi

$\mathcal{C}_j \leftarrow \{i \in \{1, \dots, n\} \text{ t.c. } j = \operatorname{argmin}_l \|x_i - \mu_l\|\}$

▷ computo i cluster centroids  $\mu_1, \dots, \mu_k$

$\forall j = 1, \dots, k \mu_j = \frac{1}{|\mathcal{C}_j|} \sum_{i \in \mathcal{C}_j} x_i$

**end while**

Questo algoritmo ha **convergenza garantita** se l'insieme delle possibili partizioni è finito. Potrebbe però terminare in un **minimo locale** non corrispondente a quello globale.

### Running time

L'assegnamento dei punti ai clusters ha costo  $O(kn)$  mentre il calcolo dei centroidi viene eseguito in tempo  $O(n)$ .

### Scelta della distanza

Un problema nella realizzazione di questi algoritmi è la scelta della distanza da usare per valutare  $\|x_i - \mu_i\|$ . Una possibile scelta è la distanza euclidea

$$d(x, y) \doteq \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (4.7)$$

Questa però non corrisponde sempre alla soluzione migliore.

Supponiamo di dover eseguire il **clustering di documenti**. Per farlo dobbiamo trasformare i documenti in punti di uno spazio vettoriale: prendiamo una feature per ogni parola e come valore assegniamole il numero di volte che tale parola appare in un documento.

In questo caso **documenti che contengono esattamente le stesse parole ma in “quantità” diverse risulterebbero “lontani”**. Prendendo l'esempio in figura, impiegando la distanza euclidea avremmo  $q$  e  $d_2$  molto lontani nonostante essi siano molto simili.

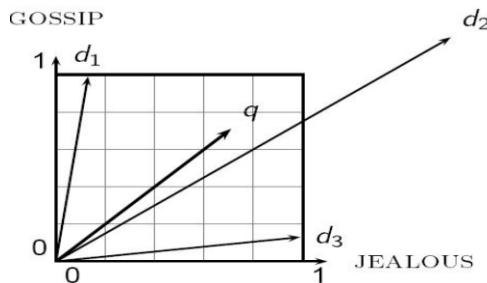


Figura 4.3: esempio documenti nello spazio vettoriale

Per ovviare a questo problema usiamo come misura di lontananza **cosine similarity**

$$\text{sim}(x, y) = \frac{x \cdot y}{|x||y|} = \frac{x}{|x|} \cdot \frac{y}{|y|} = \frac{\sum_{i=1}^n x_i y_i}{\sqrt{\sum_{i=1}^n x_i^2} \sqrt{\sum_{i=1}^n y_i^2}} \quad (4.8)$$

questa distanza è correlata all'angolo compreso tra due vettori e non tiene in considerazione il loro modulo.

### Clusters sferici

K-means, avendo come criterio di appartenenza ai clusters la distanza dai centri, restituisce esclusivamente clusters sferici. A causa di ciò talvolta il clustering può avvenire in modo errato.

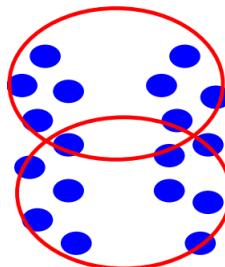


Figura 4.4: esempio fallimento di k-means

vedremo ora un algoritmo migliore che evita questa problematica.

## 4.2.2 Expectation Maximization (EM) clustering

Assumiamo che i dati arrivino da un insieme di Gaussiane (**mixture of gaussians**) e che quindi i clusters abbiano **forma ellittica**. Assegneremo ad ogni esempio una probabilità di appartenere ai clusters (**soft clustering**).

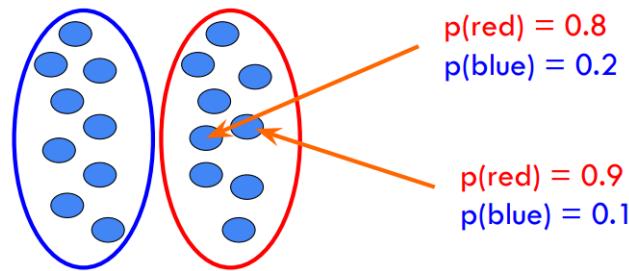


Figura 4.5: esempio EM clustering

**Definizione 4.3** (Mixture of gaussians). Definendo la gaussiana unidimensionale

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \quad (4.9)$$

Data la matrice di coovarianza  $\Sigma$  definiamo insieme di gaussiane

$$N[x, \mu, \Sigma] = \frac{1}{(2\pi)^{d/2} \sqrt{\det(\Sigma)} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)}} \quad (4.10)$$

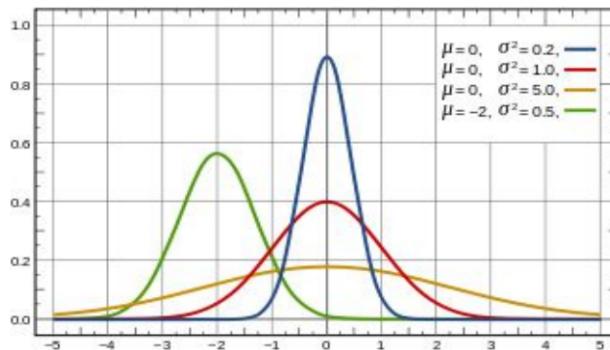


Figura 4.6: gaussiane unidimensionali

### Algoritmo

L'algoritmo per EM è ad alto livello molto simile a quello di K-means.

---

#### EM algorithm

---

**Require:** usare una strategia di inizializzazione per calcolare i valori iniziali di  $\theta_1, \dots, \theta_k$

**while** clusters subiscono variazioni (convergenza) **do**

|                                     |  |
|-------------------------------------|--|
| $p(\theta_c x_i), \forall x_i$      | ▷ assegno softpoints ad ogni cluster calcolandone la probabilità di appartenenza |
| $\theta_j, \forall j = 1, \dots, k$ | ▷ calcolo i nuovi centri scegliendo il "più probabile" per ogni cluster          |

**end while**

---

anche per EM clustering la convergenza è garantita ad un minimo locale.

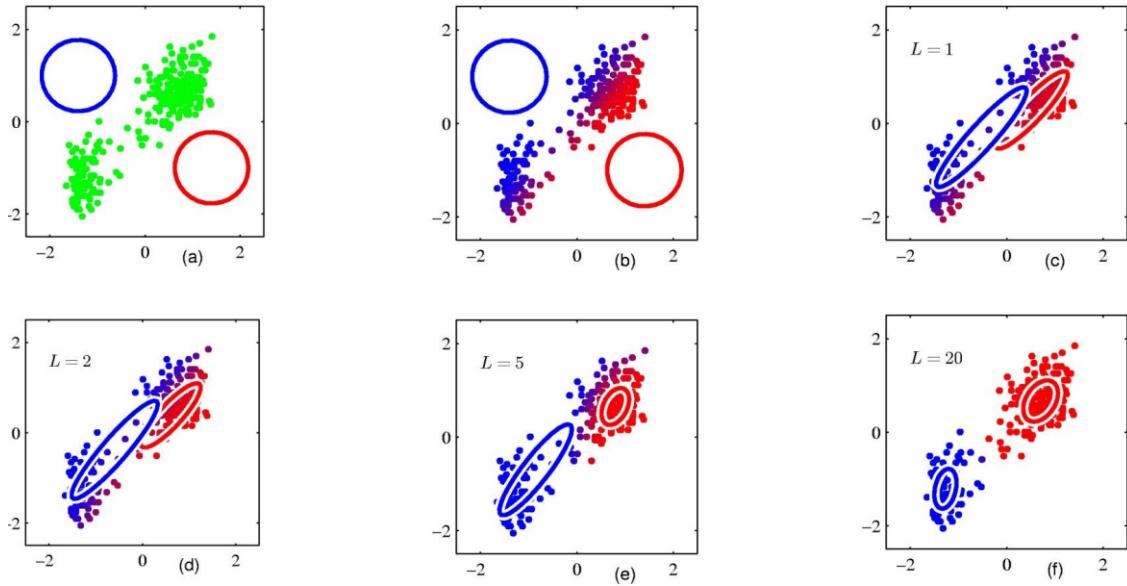


Figura 4.7: EM clustering dopo L iterazioni

#### 4.2.3 Spectral clustering

K-mean ed EM-clustering sono gli algoritmi di clustering più usati. Tuttavia non possono risolvere tutte le clustering task, se i dati seguono una **distribuzione non gaussiana** essi falliscono.

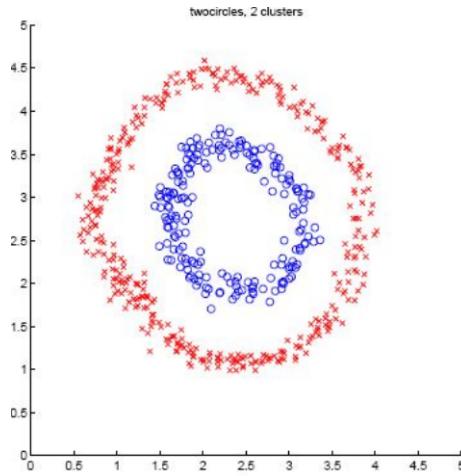


Figura 4.8: distribuzione non gaussiana

Spectral clustering raggruppa i punti basandosi su un grafo che connette i K-nearest neighbor. Definiamo una nozione di **similarità** tra i nodi, usualmente si sceglie il nucleo gaussiano

$$W(i, j) = e^{-\frac{|x_i - x_j|^2}{\sigma^2}} \quad (4.11)$$

Se  $x_i, x_j$  simili  $W(i, j) \rightarrow 1$ , se molto diversi  $W(i, j) \rightarrow 0$ . Connettiamo tutte le coppie di nodi  $(x_i, x_j)$  tra loro con un arco di peso  $W(i, j)$ .

Useremo inoltre le seguenti definizioni:

**Definizione 4.4** (Grado di un nodo). Il grado di un nodo è dato dal numero di nodi adiacenti ad esso

$$d_i \doteq \sum_j w_{i,j} \quad (4.12)$$

**Definizione 4.5** (Volume di un insieme). Il volume di un insiem  $A \subset V$  è dato dalla somma dei gradi dei nodi presenti in  $A$

$$\text{vol}(A) \doteq \sum_{i \in A} d_i \quad (4.13)$$

**Definizione 4.6** (Taglio di un grafo). Consideriamo una divisione del grafo  $V$  in 2 parti  $A, B$ . Definiamo  $\text{Cut}(A, B)$  la somma dei pesi degli archi che connettono i due gruppi

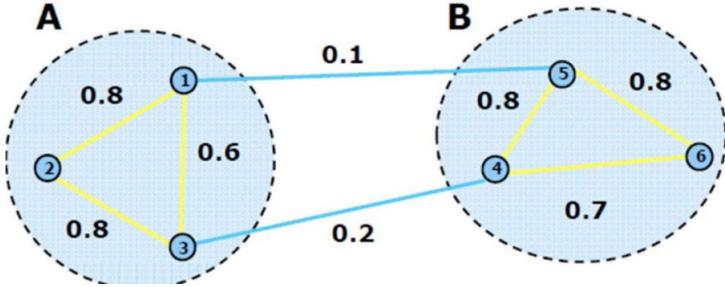


Figura 4.9: esempio taglio ( $\text{cut}(A, B) = 0.3$ )

L'obiettivo di spectral clustering sarà quello di **trovare la partizione che minimizza il taglio**, ovvero quello di trovare

$$\operatorname{argmin}_{A,B} \text{NCut}(A, B) = \frac{\text{Cut}(A)}{\text{vol}(A)} + \frac{\text{Cut}(B)}{\text{vol}(B)} \quad (4.14)$$

### 4.3 Density estimation

Trovare una distribuzione di probabilità  $f \in \Delta(\mathcal{F})$  che soddisfi i dati  $z \in \mathcal{F}$ . Memo:  $f \in \Delta(\mathcal{F}) \Leftrightarrow f : \mathcal{F} \rightarrow [0, 1]$ . Tale operazione è utile per:

- stimare la distribuzione di un training set;
- generare nuovi dati partendo dalla distribuzione di quelli già presenti;
- rilevare anomalie verificando la probabilità dei data points.

**Definizione 4.7** (Generative models). modelli statistici che descrivono la distribuzione  $p_X$  o  $p_{XY}$  a seconda della disponibilità dei target data.

**Definizione 4.8** (Discriminative models). modelli statistici che descrivono la distribuzione condizionata  $p_{Y|X}$ . Un modello discriminativo può essere ricavato da uno generativo applicando la formula di Bayes (non viceversa)

$$p_{Y|X}(y|x) = \frac{p_{XY}(x, y)}{\sum_{\hat{y}} p_{XY}(x, \hat{y})}, \text{ ricordando } \sum_{\hat{y}} p_{XY}(x, \hat{y}) = p_X(x) \quad (4.15)$$

La density estimation può essere eseguita sia con supervised che con unsupervised learning:

- supervised:  $\mathcal{F} = X \times Y$ ;
- unsupervised:  $\mathcal{F} = X$ .

Può inoltre essere esplicita o implicita:

- **esplicita**: cerchiamo  $f \in \Delta(\mathcal{F})$  che soddisfi dati  $z \in \mathcal{F}$  che derivano da una distribuzione sconosciuta  $p_{\text{data}} \in \Delta(\mathcal{F})$ ;
- **implicita**: cerchiamo  $f : \Omega \rightarrow \mathcal{F}$  che **generi** dati  $f(\omega) \in \mathcal{F}$ , partendo da un input  $\omega$  derivato da una distribuzione predefinita  $p_\omega \in \Delta(\Omega)$ , facendo in modo che i dati generati rispettino una distribuzione sconosciuta  $p_{\text{data}} \in \Delta(\mathcal{F})$ .

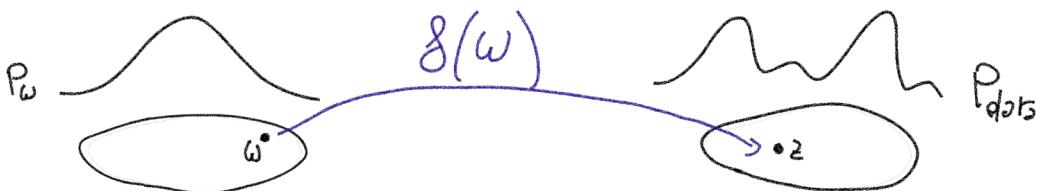


Figura 4.10: esempio density estimation implicita

Per il corretto funzionamento di density estimation dobbiamo definire:

- uno spazio di ipotesi (hypothesis space)  $\mathcal{H} \subset \Delta(\mathcal{F})$  dei modelli che possono rappresentare la distribuzione di probabilità;
- una misura di divergenza (divergence measure)  $d \in \mathbb{R}^{\Delta(\mathcal{F}) \times \Delta(\mathcal{F})}$  tra le distribuzioni di probabilità in  $\Delta(\mathcal{F})$ ;
- una ipotesi  $q^* \in \mathcal{H}$  che soddisfi al meglio la distribuzione in accordo con  $p_{\text{data}}$

$$q^* \in \operatorname{argmin}_{q \in \mathcal{H}} d(p_{\text{data}}, q) \quad (4.16)$$

### 4.3.1 Variational AutoEncoders (VAE)

Metodo di density estimation esplicito introdotto nel 2014 da Kingma e Welling. Al concetto di AutoEncoders si è aggiunta l'idea che lo **spazio latente**  $\omega$  segua una distribuzione (tipicamente gaussiana).

**Definizione 4.9** (AutoEncoders). Gli AutoEncoders sono metodi per comprimere high-dimensional data  $x$  in una loro rappresentazione di dimensione inferiore  $\omega$ . La compressione preserverà i fattori più significativi dei dati.

Gli encoders sono allenati sfruttando dei decoder, essi rimappano la rappresentazione fornita in output dagli encoder in uno spazio di dimensione superiore  $\hat{x}$  (**reconstruction**). L'obiettivo sarà quello di minimizzare la divergenza tra  $x$  e  $\hat{x}$ .

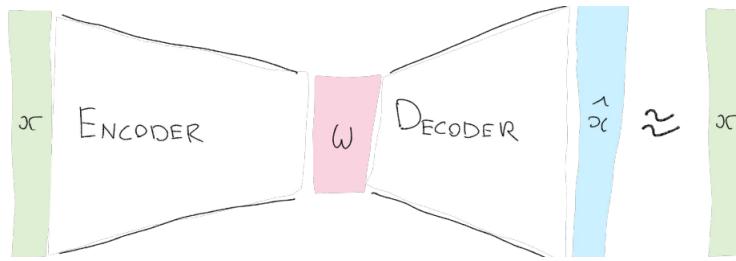


Figura 4.11: allenamento AutoEncoders

Dopo l'allenamento il decoder non viene più utilizzato e il classificatore andrà ad agire sui dati compressi  $\omega$  restituendo l'output finale  $y$ .

#### AutoEncoders come modello generativo

Il decoder potrebbe essere utilizzato per generare nuovi dati partendo dai dati compressi, non è garantito però che essi soddisfino la distribuzione  $p_{\text{data}}$ .

Vorremmo in particolare che il decoder mappasse i punti  $a \in \omega$  in punti  $b \in \hat{x}$  soddisfacendo

$$p(a) \approx p_{\text{data}}(b), \quad a \in \omega, b \in \hat{x} \quad (4.17)$$

dove  $p(\cdot)$  è la distribuzione della rappresentazione compressa.

Per ottenere tale risultato cerchiamo  $q^*$  tale che soddisfi la formula (4.16). Per farlo definiamo una funzione parametrizzata da  $\theta$

$$q_\theta(x) = \mathbb{E}_{\omega \sim p^\omega}[q_\theta(x|\omega)] \quad (4.18)$$

dove  $p^\omega = N(0, 1)$  è assunta a priori mentre come gaussiana standard e  $q_\theta(x|\omega)$  è il decoder.

Cerchiamo quindi i parametri  $\theta$  tali che:

$$\theta^* \in \operatorname{argmin}_{\theta \in \Theta} d(q_\theta, p_{\text{data}}) \quad (4.19)$$

dove come misura di divergenza scegliamo

$$d_{KL}(q_\theta, p_{\text{data}}) = \mathbb{E}_{x \sim p_{\text{data}}} \left[ \log \frac{p_{\text{data}}(x)}{q_\theta(x)} \right] \quad (4.20)$$

$$= \mathbb{E}_{x \sim p_{\text{data}}} [\log q_\theta(x)] + \text{const} \quad (4.21)$$

$$= \mathbb{E}_{x \sim p_{\text{data}}} [\log \mathbb{E}_{\omega \sim p^\omega}[q_\theta(x|\omega)]] + \text{const} \quad (4.22)$$

Purtroppo però  $\log \mathbb{E}_{\omega \sim p^\omega}[q_\theta(x|\omega)]$  è intrattabile ma sappiamo

$$d_{KL}(q_\theta, p_{\text{data}}) \leq \mathbb{E}_{x \sim p_{\text{data}}} [-\mathbb{E}_{\omega \sim q_\psi(\cdot|x)}[\log q_\theta(x|\omega)] + d_{KL}(q_\psi(\cdot|x), p_\omega)] + \text{const} \quad (4.23)$$

dove  $q_\psi(\omega|x) \in \Delta(\Omega)$  denota una distribuzione di encoding. Notiamo è ancora intrattabile  $\mathbb{E}_{\omega \sim q_\psi(\cdot|x)}[\log q_\theta(x|\omega)]$ , possiamo però **stimare il gradiente**, mentre  $d_{KL}(q_\psi(\cdot|x), p_\omega)$  detto **regolarizzatore** potrebbe avere una formula chiusa come soluzione (es. distribuzione gaussiana).

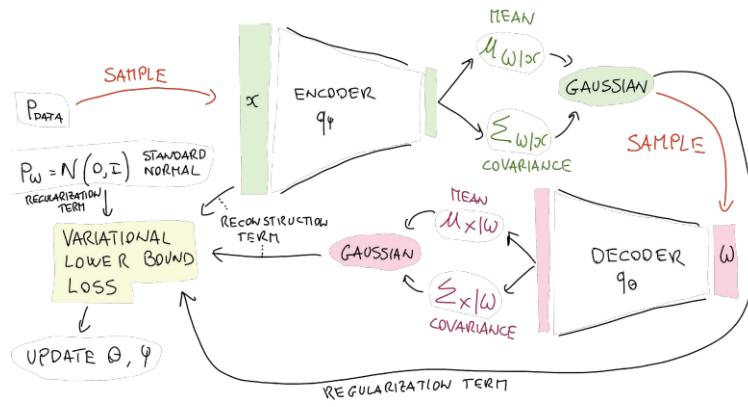


Figura 4.12: allenamento AutoEncoders

### Problemi con VAE

- **Underfitting:** inizialmente il regolarizzatore ha troppa importanza e questo tende a diminuire le capacità del modello;
- **Blurry:** tendenza a generare dati “blurrati”.

### 4.3.2 Generative adversarial networks (GANs)

Le GANs introducono la possibilità di stimare densità implicite. Assumiamo di avere a priori una densità  $p_\omega \in \Delta(\Omega)$  (per l'insieme  $\Omega$ ) e un generatore (o decoder)  $g_\theta \in X^\Omega$  che genera punti in  $X$  partendo da elementi randomici di  $\Omega$ . La densità indotta da  $p_\omega$  e dal generatore  $g_\theta$  sarà data da

$$q_\theta(x) = \mathbb{E}_{\omega \sim p^\omega} \delta[g_\theta(\omega) - x] \quad (4.24)$$

dove  $\delta$  è la funzione delta di Dirac

$$\delta(x - \alpha) \doteq \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{ip(x-\alpha)} dp \quad (4.25)$$

L'obiettivo di GAN è trovare  $\theta^*$  tale che  $q_{\theta^*}$  sia più simile possibile alla distribuzione  $p_{\text{data}}$  basandosi sulla divergenza di Jensen-Shannon

$$\theta^* \in \operatorname{argmin}_{\theta \in \Theta} d_{JS}(p_{\text{data}}, q_\theta) \quad (4.26)$$

$$d_{JS}(p, q) \doteq \frac{1}{2} d_{KL}\left(p, \frac{p+q}{2}\right) + \frac{1}{2} d_{KL}\left(q, \frac{p+q}{2}\right) \quad (4.27)$$

Come visto in VAE  $d_{KL}$  può essere approssimato a problemi trattabili.

### Funzionamento

GAN è suddiviso in due componenti che operano uno “contro” l’altro (Adversarial si traduce infatti in “avversario”):

- **generatore:** tenta di generare dati non distinguibili dagli originali, cercando di soddisfare (4.26);
- **discriminatore:** tenta di distinguere i dati originali da quelli generati.

Il problema è poi risolto con gradient descent e approcci specializzati per gestire il min-max problem.

### Problemi con GAN

- **training stability:** i parametri potrebbero oscillare senza mai convergere;
- **mode collapse:** il generatore potrebbe limitarsi ad apprendere come ricopiare elementi del training set;
- **vanishing gradient:** se il discriminatore è molto potente lascia il generatore con un gradiente minuscolo sul quale operare.

## GAN vs VAE

VAE è usato principalmente per:

- generazioni di immagini;
- processamento di linguaggi naturali;
- anomaly detection.

Mentre GAN è impiegato principalmente per la generazione di immagini identiche alle originali ma aventi una risoluzione maggiore.

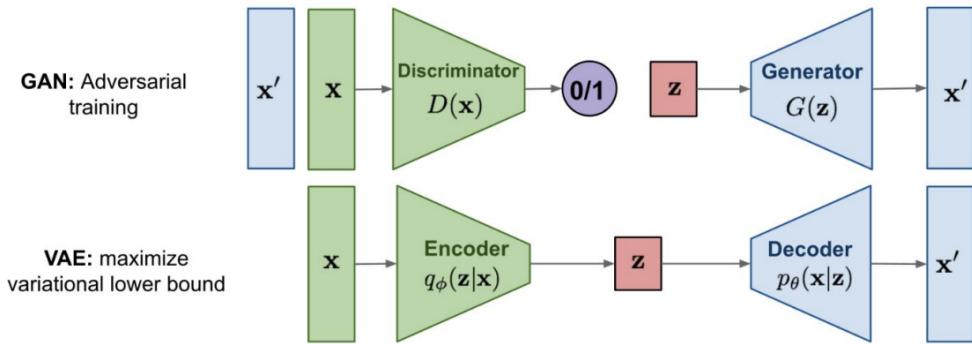


Figura 4.13: comparazione del funzionamento di GAN e VAE

### 4.3.3 Denoising diffusion models

I **denoising diffusion models** consistono in due processi:

- **forward**: aggiunge “rumore” applicando una **convoluzione gaussiana**;
- **generation**: rimuove il “rumore” dai dati generando nuovi dati simili a quelli di partenza.

#### Forward process

Formalmente il processo di aggiunta del rumore è definito dalla formula

$$q(x_t|x_{t-1}) = N(x_t; \sqrt{1 - \beta_t}x_{t-1}, \beta_t I) \quad (4.28)$$

e quindi

$$q(x_j|x_0) = \prod_{t=1}^j q(x_t|x_{t-1}), \forall j = 1, \dots, T \quad (4.29)$$

Formalmente stiamo applicando ai dati una **convoluzione gaussiana**, forzando la distribuzione di partenza in una distribuzione gaussiana.

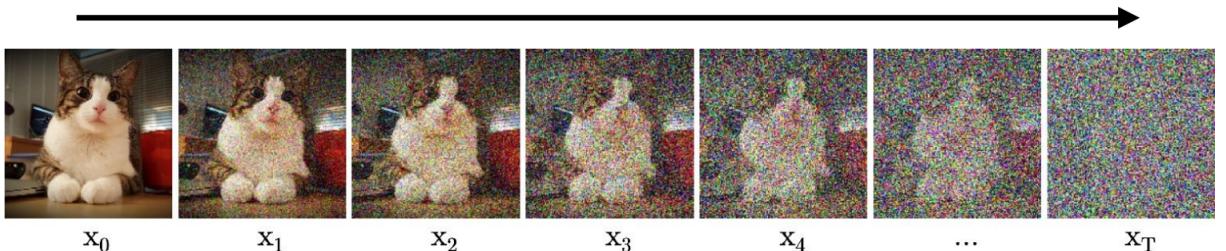


Figura 4.14: processo di aggiunta del rumore

#### Generation process

Data la scelta della convoluzione gaussiana come metodo di aggiunta del rumore sappiamo che

$$q(x_T) \approx N(0, 1) \quad (4.30)$$

Per rimuovere il rumore definiamo quindi la funzione

$$p_\theta(x_{t-1}|x_t) = N(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I) \quad (4.31)$$

e quindi

$$p_{\theta}(x_j) = p(x_T) \prod_{t=1}^T p_{\theta}(x_{t-1}|x_t) \quad (4.32)$$

**Diffusion models:**  
Gradually add Gaussian noise and then reverse

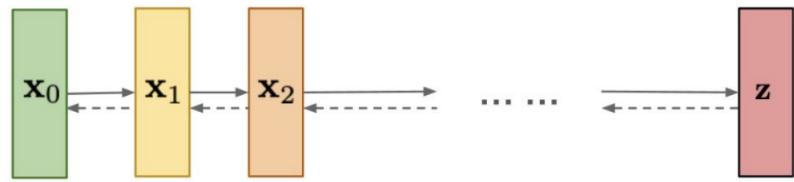


Figura 4.15: funzionamento dei diffusion models

# Capitolo 5

## Reinforcement Learning

Inspirato a ricerche in ambito psicologico e di apprendimento degli animali. Il problema coinvolge un agente che interagisce in un environment e riceve ricompense basate sulle azioni che compie. L'obiettivo dell'agente è quello di **massimizzare le ricompense**.

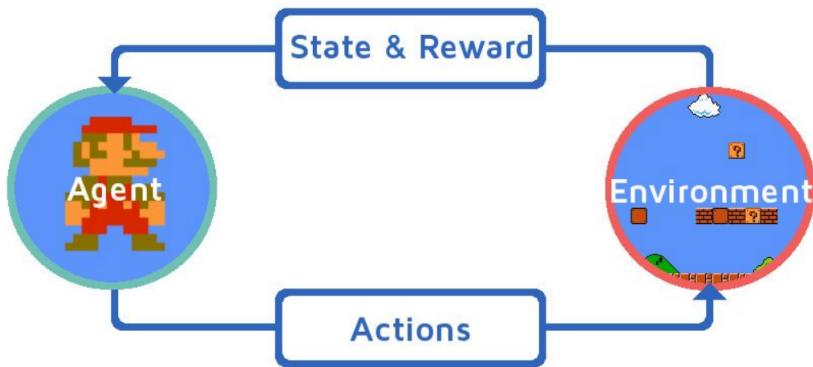


Figura 5.1: reinforcement learning

### 5.1 Markov decision process (MDP)

MDP è una framework usato per prendere decisioni in un environment stocastico. L'obiettivo è quello di trovare delle **policy** che danno le migliori azioni da compiere per ogni stato del nostro environment. Per risolvere MDP useremo **programmazione dinamica**.

**Definizione 5.1** (Markov assumption). La probabilità di passare da uno stato  $s$  a uno stato  $s'$  dipende solo da  $s$  e non dalle decisioni prese in precedenza.

#### Componenti

- **stato**  $s$ , inizializzato a  $s_0$ ;
- **azioni**  $a$ ;
- **transition model**  $P(s'|s, a)$ , qui sfruttiamo Markov assumption;
- **reward function**  $r(s) : \mathcal{S} \rightarrow \mathbb{R}$ ;
- **policy**  $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$ , le azioni che un agente può compiere in uno stato  $s$ .

Possiamo quindi definire MDP come:

**Definizione 5.2** (Markov decision process).

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma) \quad (5.1)$$

dove  $\mathcal{S}$  è l'insieme dei possibili stati,  $\mathcal{A}$  insieme delle possibili azioni,  $\mathcal{R}$  distribuzione delle ricompense rispetto a una coppia (state, action),  $\mathbb{P}$  transition probability ovvero la distribuzione sullo stato successivo rispetto a (state,action),  $\gamma$  discount factor.

### 5.1.1 MDP loop

Al tempo iniziale  $t = 0$ , inizializziamo uno stato iniziale  $s_0 \sim p(s_0)$ . Ripetiamo poi le seguenti operazioni:

- l'agente sceglie un'azione  $a_t \in \mathcal{A}$ ;
- l'environment campiona una ricompensa  $r_t \sim R(\cdot | s_t, a_t)$ ;
- l'environment campiona lo stato successivo  $s_{t+1} \sim P(\cdot | s_t, a_t)$ ;
- l'agente riceve la ricompensa  $r_t$  e lo stato  $s_{t+1}$ .

L'obiettivo è trovare una *policy*  $\pi^*$  che massimizzi *cumulative discounted reward*.

#### Cumulative discounted reward

Supponiamo che la policy  $\pi$  partendo dallo stato  $s_0$  porti a una sequenza di stati  $s_0, s_1, \dots$ , definiamo

**Definizione 5.3** (Cumulative reward).

$$\sum_{t \geq 0} r(s_t) \tag{5.2}$$

dove  $r$  è la reward function,  $s_0, s_1, \dots$  sequenza di stati (anche infinita).

Tipicamente definiamo la

cumulative reward come somma di rewards scontate da un fattore  $\gamma \in (0, 1]$

$$\sum_{t \geq 0} \gamma^t r(s_t) \tag{5.3}$$

Il **discounting factor** controlla l'importanza delle future ricompense rispetto a quelle immediate e aiuta la convergenza dell'algoritmo.

### 5.1.2 Reinforcement learning vs Supervised learning

#### Supervised learning loop

- prendere un input  $x_i$  campionato dalla data distribution;
- usare un modello parametrico (di parametri  $w$ ) per predire l'output  $y$ ;
- osservare il target output  $y_i$  e calcolare la loss function  $l(w, x_i, y_i)$ ;
- aggiornare i parametri  $w$  per ridurre la loss function utilizzando *Stochastic Gradient Descent*

$$w \leftarrow w - \eta \nabla l(w, x_i, y_i) \tag{5.4}$$

#### Reinforcement learning loop

- in uno stato  $s$  prendere una azione  $a$  determinata dalla policy  $\pi(s)$ ;
- selezionare il nuovo stato  $s'$  (**goal state**) basandosi su un modello  $P(s'|s, a)$ ;
- usare  $s'$  e la reward  $r(s)$  per aggiornare la policy.

## 5.2 Reinforcement learning methods

Esistono due principali approcci a RL:

- **value-based methods**, l'obiettivo dell'agente è ottimizzare una funzione  $V(s)$  corrispondente alla somma delle rewards  $r(s)$  in un determinato stato;
- **policy-based methods**, definiamo una policy che ottimizzeremo direttamente.

### 5.2.1 Value based methods

**Definizione 5.4** (Value function). Restituisce la somma totale delle rewards che un agente può aspettarsi in un determinato stato. Partendo da uno stato  $s$  con una policy  $\pi$  definiamo

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) \mid S_0 = s, \pi \right] \quad (5.5)$$

scegliendo  $a_t = \pi(s_t)$ ,  $s_{t+1} \sim P(\cdot, s_t, a_t)$ .

Definiamo inoltre **optimal value of a state**

$$V^*(s) \doteq \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) \mid S_0 = s, \pi \right] \quad (5.6)$$

Ai fini di reinforcement learning conviene però maggiormente la definizione

**Definizione 5.5** (Q-value function). Definizione della value function sulle coppie (state,action)

$$Q^\pi(s, a) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) \mid s_0 = s, a_0 = a, \pi \right] \quad (5.7)$$

Definiamo quindi **optimal Q-value of a state-action pair**

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) \mid s_0 = s, a_0 = a, \pi \right] \quad (5.8)$$

Chiamiamo inoltre **optimal policy**

$$\pi^*(s) \doteq \operatorname{argmax}_a Q^*(s, a) \quad (5.9)$$

### Q-learning

Q-learning è basato sul fatto che la matrice  $R = \text{State} \times \text{Action}$  delle ricompense non è conosciuta in partenza dall'agente. Esso avrà a disposizione una matrice  $Q$  che mette in relazione (stato,azione) restituendo la ricompensa,  $Q$  sarà inizializzata a 0 e diventerà come  $R$  attraverso l'esperienza.

Q-learning algorithm

```

 $Q_{ij} \leftarrow 0, \forall i, j$                                      ▷ Inizializzo  $Q$  a 0
Seleziono uno stato iniziale  $s_{\text{next}} = s_0$  random
for all episode do
    while  $s_{\text{next}} \neq s'$                                          ▷ finchè lo stato non è lo stato obiettivo ( $s'$ ) do
        Seleziona una possibile azione  $a$  dallo stato attuale  $s$ 
        ▷ Consideriamo se usare  $a$  per andare al prossimo stato  $s_{\text{next}}$  trovando il valore massimo di  $Q$  per  $s_{\text{next}}$ 
         $Q^*(s, a) = r(s, a) + \gamma \max_{a'} [Q^*(s_{\text{next}}, a')]$ 
    end while
end for

```

---

**Definizione 5.6** (Episode). insieme di azioni che iniziano nello stato iniziale e terminano nel goal state  $s'$ .

$$\begin{array}{c}
 \begin{array}{ccccccc}
 & \color{red}{0} & \color{red}{1} & \color{red}{2} & \color{red}{3} & \color{red}{4} & \color{red}{5} \\
 \mathcal{Q} = & \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} & \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \text{Action} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \\
 R = & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} & \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \text{Action} & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \\
 \end{array} & \mathcal{Q} = & \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 0 & 0 & 80 & 0 \\ 2 & 0 & 0 & 0 & 64 & 0 \\ 3 & 0 & 80 & 51 & 0 & 80 \\ 4 & 64 & 0 & 0 & 64 & 0 \\ 5 & 0 & 80 & 0 & 80 & 100 \end{bmatrix}
 \end{array}$$

Figura 5.2: training dopo svariati episodi

## Deep Q-learning

Per calcolare  $Q^*(s, a)$  abbiamo applicato l'**equazione di Bellman**.

$$Q^*(a, b) = r(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^*(s', a') = \mathbb{E}_{s' \sim P(\cdot|s, a)} [r(s) + \gamma \max_{a'} Q^*(s', a')|s, a] \quad (5.10)$$

Il problema però è che lo **spazio degli stati può essere enorme** (es. Atari game) e questo rende il calcolo di  $Q^*(a, b)$  **tropppo oneroso**. Sceglieremo quindi di **approssimare**  $Q^*$  usando funzioni parametriche

$$Q^*(s, a) \approx Q_w(s, a) \quad (5.11)$$

Trovaremo poi il massimo di questa funzione sfruttando *Stochastic Gradient Descent*.

### 5.2.2 Policy gradient methods

Invece che rappresentare una policy usando Q-values potrebbe essere più efficiente parametrizzare  $\pi$  e apprenderla direttamente. Questo è vero specialmente in spazio di azioni grandi e continui nei quali la Q-value function risulterebbe estremamente complicata.

Si imparerà quindi una funzione basandosi sulla distribuzione di probabilità delle azioni nello stato corrente

$$\pi_\theta(s, a) \approx P(a|s) \quad (5.12)$$

L'idea di base è quella di imparare una buona policy attraverso l'esperienza e le ricompense ricevute. Cerchiamo quindi i parametri  $\theta$  che massimizzano l'**expected reward**

$$J(\theta) \doteq \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t \mid \pi_\theta \right] = \mathbb{E}[r(\tau)] \quad (5.13)$$

anche in questo caso useremo gradient descent.