

# Introduction to machine learning

Giacomo Fantoni

Telegram: @GiacomoFantoni

Github: <https://github.com/giacThePhantom/intro2ml>

13 luglio 2021

# Indice

<b>1</b>	<b>Introduzione</b>	<b>6</b>
1.1	Definizioni . . . . .	6
1.2	Processo . . . . .	6
1.2.1	Il processo di apprendimento . . . . .	6
1.3	Modello . . . . .	6
1.4	Deep learning . . . . .	7
<b>2</b>	<b>Machine learning basics</b>	<b>8</b>
2.1	Introduzione . . . . .	8
2.1.1	Processo di learning . . . . .	8
2.2	Dati . . . . .	8
2.2.1	Training e test set . . . . .	8
2.3	Task . . . . .	9
2.4	Modello . . . . .	9
2.4.1	Target ideale . . . . .	9
2.4.2	Target feasible . . . . .	9
2.4.3	Target attuale . . . . .	9
2.4.4	Funzione di errore . . . . .	10
2.4.5	Tipi di errore . . . . .	10
2.4.6	Stimare l'errore di generalizzazione . . . . .	10
2.5	Tipi di learning . . . . .	11
2.5.1	Supervised learning . . . . .	11
2.5.2	Unsupervised learning . . . . .	12
2.5.3	Reinforcement learning . . . . .	12
2.6	Polynomial curve fitting . . . . .	12
2.6.1	Dati . . . . .	12
2.6.2	Modello e spazio di ipotesi . . . . .	13
2.6.3	Error function . . . . .	13
2.6.4	Funzione obiettivo . . . . .	13
2.6.5	Regolarizzazione . . . . .	13
<b>3</b>	<b>KNN</b>	<b>14</b>
3.1	Introduzione . . . . .	14
3.2	Misurare la distanza . . . . .	14
3.3	Decision boundaries . . . . .	14
3.4	Il ruolo di $K$ . . . . .	14

3.4.1	Underfitting . . . . .	14
3.4.2	Overfitting . . . . .	15
3.5	Scelta di $K$ . . . . .	15
3.6	Variazioni di $K$ . . . . .	15
3.6.1	$K$ -NN pesata . . . . .	15
<b>4</b>	<b>Modelli lineari</b>	<b>16</b>
4.1	Introduzione . . . . .	16
4.1.1	Bias . . . . .	16
4.2	Linear separability . . . . .	16
4.2.1	Definire una linea . . . . .	16
4.3	Definizione di modello lineare . . . . .	17
4.3.1	Training . . . . .	17
4.3.2	Perceptron . . . . .	17
4.4	Perceptron e reti neurali . . . . .	18
4.4.1	Funzione di attivazione . . . . .	18
4.4.2	Storia del perceptron . . . . .	18
<b>5</b>	<b>Decision Trees</b>	<b>19</b>
5.1	Struttura . . . . .	19
5.2	Funzionamento . . . . .	19
5.2.1	Inferenza . . . . .	19
5.3	Decision trees learning algorithm . . . . .	19
5.3.1	Crescere una foglia . . . . .	20
5.3.2	Crescere un nodo . . . . .	20
5.3.3	Algoritmo . . . . .	21
5.3.4	Split selection . . . . .	21
5.3.5	Predizione delle foglie . . . . .	21
5.4	Misure di impurità per la classificazione . . . . .	21
5.5	Misure di impurità per la regressione . . . . .	22
5.6	Data features e attributi . . . . .	22
5.7	Funzioni di split o routing . . . . .	22
5.7.1	Features discrete e nominali . . . . .	22
5.7.2	Features ordinali . . . . .	23
5.7.3	Obliquo . . . . .	23
5.8	Decision trees e overfitting . . . . .	23
5.9	Random forest . . . . .	23
5.10	Confronto con KNN . . . . .	24
<b>6</b>	<b>Multi class classification</b>	<b>25</b>
6.1	Introduzione . . . . .	25
6.1.1	Classificazione binaria . . . . .	25
6.1.2	Classificazione multi classe . . . . .	25
6.1.3	Approccio black box alla multi class classification . . . . .	26
6.2	One versus all <i>OVA</i> . . . . .	26
6.2.1	Ambiguità . . . . .	26
6.2.2	Algoritmi . . . . .	26
6.3	All versus all <i>AVA</i> . . . . .	26

6.3.1	AVA training . . . . .	27
6.3.2	AVA classification . . . . .	27
6.3.3	Algoritmi . . . . .	27
6.4	Confronto tra <i>OVA</i> e <i>AVA</i> . . . . .	28
6.4.1	Tempo di training . . . . .	28
6.4.2	Tempo di test . . . . .	28
6.4.3	Errori . . . . .	28
6.5	Riassunto . . . . .	28
6.6	Multiclass evaluation . . . . .	28
6.6.1	Microaveraging . . . . .	28
6.6.2	Macroaveraging . . . . .	29
6.6.3	Confusion matrix . . . . .	29
<b>7</b>	<b>Ranking</b> . . . . .	<b>30</b>
7.1	Classificazione multiclasse e multilabel . . . . .	30
7.2	Problema del ranking . . . . .	30
7.2.1	Preference function . . . . .	30
7.3	Utilizzo del ranking e della preference function . . . . .	31
7.4	Naive Ranking . . . . .	31
7.4.1	Training . . . . .	31
7.4.2	Testing . . . . .	31
7.5	Bipartite ranking . . . . .	31
7.6	Ordinamento e $\omega$ -ranking . . . . .	32
7.6.1	Testing . . . . .	32
7.6.2	Implementazione . . . . .	33
<b>8</b>	<b>Gradient descent</b> . . . . .	<b>34</b>
8.1	Model based machine learning . . . . .	34
8.1.1	Modelli lineari . . . . .	34
8.2	Loss functions . . . . .	35
8.2.1	Loss 0/1 . . . . .	35
8.2.2	Funzioni convesse . . . . .	35
8.2.3	Surrogate loss function . . . . .	36
8.3	Gradient descent . . . . .	36
8.3.1	Spostamento in direzione della minimizzazione dell'errore . . . . .	36
8.3.2	Learning algorithm del perceptron . . . . .	37
8.3.3	Costante $\mathbf{c}$ . . . . .	37
8.3.4	Gradiente . . . . .	37
<b>9</b>	<b>Regularization</b> . . . . .	<b>39</b>
9.1	Introduzione . . . . .	39
9.2	Regolarizzatori . . . . .	39
9.2.1	Regolarizzatori comuni . . . . .	39
9.3	Gradient descent e regolarizzazione . . . . .	40
9.4	Regolarizzazione con le $p$ -norms . . . . .	40
9.4.1	L1 . . . . .	40
9.4.2	L2 . . . . .	40
9.4.3	Lp . . . . .	41

---

9.5	Metodi di machine learning con regolarizzazione . . . . .	41
<b>10</b>	<b>Support vector machines</b>	<b>42</b>
10.1	Introduzione . . . . .	42
10.1.1	Considerazioni su perceptron e gradient descent . . . . .	42
10.1.2	Idea delle support vector machines . . . . .	42
10.2	Margini . . . . .	42
10.2.1	Support vectors . . . . .	42
10.2.2	Calcolare il margine . . . . .	43
10.3	Problema di ottimizzazione . . . . .	43
10.3.1	Massimizzare il margine . . . . .	43
10.4	Soft margin classification . . . . .	43
10.4.1	Risolvere il problema delle SVM . . . . .	44
10.5	Data non linearmente separabile . . . . .	44
10.5.1	Soft margin classifier . . . . .	45
10.5.2	Identificare i support vector . . . . .	45
10.5.3	Utilizzo di SVN in maniera non lineare . . . . .	45
<b>11</b>	<b>Neural Networks</b>	<b>47</b>
11.1	Introduzione . . . . .	47
11.1.1	Perceptron . . . . .	47
11.1.2	Multi layer perceptron . . . . .	47
11.1.3	First AI winter . . . . .	48
11.1.4	Second AI winter . . . . .	48
11.1.5	Deep learning revolution . . . . .	49
11.1.6	Caratteristiche del deep learning . . . . .	49
11.2	Feedforward networks . . . . .	49
11.2.1	Training . . . . .	49
11.2.2	Scelte di modellamento . . . . .	49
11.2.3	Backpropagation . . . . .	51
11.2.4	Scelta di un ottimizzatore . . . . .	53
11.3	Convolutional Neural Networks (CNN) . . . . .	54
11.3.1	Origini . . . . .	55
11.3.2	Architettura . . . . .	55
11.3.3	Conclusione . . . . .	56
11.4	Altre reti neurali . . . . .	56
11.4.1	Recurrent neural networks . . . . .	56
11.4.2	Autoencoders . . . . .	57
<b>12</b>	<b>Unsupervised Learning</b>	<b>58</b>
12.1	Introduzione . . . . .	58
12.1.1	Task tipiche . . . . .	58
12.2	Dimensionality reduction . . . . .	58
12.2.1	Task . . . . .	58
12.2.2	Principal component analysis . . . . .	58
12.3	Clustering . . . . .	62
12.3.1	Task . . . . .	62
12.3.2	K-means clustering . . . . .	62

12.3.3	Problematiche del clustering . . . . .	63
12.3.4	Algoritmi di clustering . . . . .	63
12.3.5	EM clustering . . . . .	64
12.3.6	Altri algoritmi di clustering . . . . .	65
12.4	Density estimation . . . . .	65
12.4.1	Task . . . . .	65
12.4.2	Generative model . . . . .	65
12.4.3	Modelli discriminativi . . . . .	65
12.4.4	Tipi di density estimation . . . . .	66
12.4.5	Variational AutoEncode (VAE) . . . . .	66
12.4.6	Generative adversarial Networks (GAN) . . . . .	68
<b>13</b>	<b>Reinforcement learning</b>	<b>71</b>
13.1	Introduzione . . . . .	71
13.1.1	Policy . . . . .	71
13.2	Markov decision process (MDP) . . . . .	71
13.2.1	Componenti . . . . .	71
13.2.2	Ambienti stocastici . . . . .	72
13.2.3	MDP loop . . . . .	72
13.2.4	Policy . . . . .	72
13.3	Confronto tra reinforcement learning e supervised learning . . . . .	73
13.3.1	Supervised learning loop . . . . .	73
13.3.2	Reinforcement learning loop . . . . .	73
13.3.3	Differenze fondamentali . . . . .	73
13.4	Metodi di reinforcement learning . . . . .	73
13.4.1	Value based methods . . . . .	74
13.4.2	Policy gradient methods PGM . . . . .	76

# Capitolo 1

## Introduzione

### 1.1 Definizioni

Si intende per machine learning lo studio di algoritmi che migliorano autonomamente attraverso l'esperienza. È un campo dell'intelligenza artificiale. Stravolge il paradigma convenzionale della programmazione: un algoritmo di machine learning infatti prende come input un insieme di dati e risultati in modo da produrre un programma che fornisce un risultato appropriato. Coinvolge pertanto la scoperta automatica di regolarità nei dati attraverso algoritmi in modo da poter compiere azioni basate su di essi.

### 1.2 Processo

Il machine learning permette ai computer di acquisire conoscenza attraverso algoritmi che inferiscono e imparano da dati. Questa conoscenza viene rappresentata da un modello che può essere utilizzato su nuovi dati.

#### 1.2.1 Il processo di apprendimento

Il processo di apprendimento in particolare coinvolge diversi passaggi:

- Acquisizione dei dati dal mondo reale attraverso dispositivi di misurazione come sensori o database.
- Preprocessamento dei dati: filtraggio del rumore, estrazione delle feature e normalizzazione.
- Riduzione dimensionale: selezione e proiezione delle feature.
- Apprendimento del modello: classificazione, regressione, clustering e descrizione.
- Test del modello: cross-validation e bootstrap.
- Analisi dei risultati.

### 1.3 Modello

Un algoritmo di machine learning impara dall'esperienza  $E$  in rispetto di una classe di compiti  $T$  e di misurazione delle performance  $P$ , se la  $P$  di  $T$  aumenta con  $E$ . Si nota pertanto come un compito

di machine learning ben definito possiede una tripla:

$$\langle T, P, E \rangle$$

### 1.4 Deep learning

Il deep learning è un sottoinsieme del machine learning che permette a modelli computazionali composti di multipli strati di imparare la rappresentazione di dati con multipli livelli di astrazione. Si utilizza pertanto una rete neurale con diversi strati di nodi tra input e output. Questa serie di strati tra input e output computa caratteristiche rilevanti automaticamente in una serie di passaggi. Questi algoritmi sono resi possibili da:

- Enorme mole di dati disponibili.
- Aumento del potere computazionale.
- Aumento del numero di algoritmi di machine learning e della teoria sviluppata dai ricercatori.
- Aumento del supporto dall'industria.



## Capitolo 2

# Machine learning basics

### 2.1 Introduzione

Il machine learning permette ai computer di acquisire conoscenza attraverso algoritmi che imparano e inferiscono dai dati. Tale conoscenza viene rappresentata da un modello che viene poi utilizzato su dati futuri.

#### 2.1.1 Processo di learning

Si individua un processo di learning:

- Acquisizione di dati dal mondo reale attraverso sensori.
- Preprocessamento dei dati: eliminazione del rumore, estrazione delle features e normalizzazione.
- Riduzione di dimensionalità attraverso selezione e proiezione di features.
- Learning del modello: classification, regression, clustering e description.
- Test del modello attraverso cross-validation e bootstrap.
- Analisi dei risultati.

### 2.2 Dati

I dati disponibili ad un algoritmo di machine learning sono tipicamente un insieme di esempi. Questi esempi sono tipicamente rappresentati come un array di features, caratteristiche dei dati di interesse per lo studio in atto.

#### 2.2.1 Training e test set

In particolare per questi algoritmi si assume sempre che il training e il test set siano distribuiti secondo variabili indipendenti e identicamente distribuite (*i.i.d*) La distribuzione  $P_{data}$  è tipicamente sconosciuta ma si può campionare, attraverso un modello probabilistico di learning. In particolare la distribuzione di probabilità di coppie di esempio e label viene detta data generating distribution e sia il training data che il test set sono generati basandosi su di essa.

## 2.3 Task

Si intende per task una rappresentazione del tipo di predizione che viene svolta per risolvere un problema su dei dati. Viene identificata con un insieme di funzioni che possono potenzialmente risolverla. In generale consiste di una funzione che assegna ogni input  $x \in \mathcal{X}$  a un output  $y \in \mathcal{Y}$ :

$$f : \mathcal{X} \rightarrow \mathcal{Y} \quad \mathcal{F}_{task} \subset \mathcal{Y}^{\mathcal{X}}$$

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

## 2.4 Modello

Un modello è un programma per risolvere un problema. È cioè l'implementazione di una funzione  $f \in \mathcal{F}_{task}$  che può essere computata. Un insieme di modelli formano uno spazio di ipotesi:

$$\mathcal{H} \subset \mathcal{F}_{task}$$

L'algoritmo cerca una soluzione nello spazio di ipotesi. Esistono diversi tipi di modello:

- Generativi e discriminativi.
- Parametrici e non parametrici.

### 2.4.1 Target ideale

Il target ideale del modello è quello di minimizzare una funzione di errore (generalizzazione)

$$E(f; P_{data})$$

Questa funzione determina quanto bene una soluzione  $f \in \mathcal{F}_{task}$  fitta dei dati. Guida pertanto la selezione della migliore soluzione in  $\mathcal{F}_{task}$ . Pertanto:

$$f^* \in \arg \min_{f \in \mathcal{F}_{task}} E(f; P_{data})$$

### 2.4.2 Target feasible

Si deve restringere il focus sul trovare funzioni che possono essere implementate e valutate in maniera trattabile. Si definisce pertanto uno spazio di ipotesi del modello  $\mathcal{H} \subset \mathcal{F}_{task}$  e si cerca la soluzione all'interno di quello spazio:

$$f_{\mathcal{H}}^* \in \arg \min_{f \in \mathcal{H}} E(f; P_{data})$$

Si noti come questa funzione non possa essere computata correttamente in quanto  $P_{data}$  è sconosciuta.

### 2.4.3 Target attuale

Per trovare il target attuale si deve lavorare su un campione di dati o il training set

$$\mathcal{D}_n = \{z_1, \dots, z_n\}$$

Dove

$$z_i = (x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$$

$$z_i \sim P_{data}$$

Pertanto in:

$$f_{\mathcal{H}}^* \in \arg \min_{f \in \mathcal{H}} E(f; P_{data})$$

$E(f; P_{data})$  è il training error.

#### 2.4.4 Funzione di errore

Le funzioni di generalizzazione e di training error possono essere scritte in termini di una pointwise loss  $l(f; z)$  che misura l'errore che avviene a  $f$  su un esempio di training  $z$ .

$$E(f; P_{data}) = \mathbb{E}_{z \sim P_{data}} [l(f; z)]$$

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

Si nota pertanto come l'algoritmo di learning risolve il problema di ottimizzazione con target:

$$f_{\mathcal{H}}^*(\mathcal{D}_n)$$

#### 2.4.5 Tipi di errore

- Underfitting il modello non fitta sui dati di training, avviene per mancanza di dati.
- Estimation error, indotto imparando da un campione di dati.
- Overfitting quando i training data sono rumorosi e il modello li fitta perfettamente, ma impara un modello che si adatta solo su di essi e non fitta dati dal mondo reale.
- Approximation error, indotto dallo spazio di ipotesi  $\mathcal{H}$ .
- Irreducible error a causa della variabilità intrinseca.

#### 2.4.6 Stimare l'errore di generalizzazione

L'errore di generalizzazione può essere stimato utilizzando diversi insiemi di training, validation e test. Si usa il training set per fare training di un modello, quello di validazione per valutarlo e sistemare i suoi iperparametri e dopo di quello si sceglie il modello migliore che si misura attraverso le performance sul test set.

##### 2.4.6.1 Migliorare la generalizzazione

La generalizzazione può essere migliorata:

- Evitando di ottenere il minimo sul training error.
- Iniettando rumore nell'algoritmo.
- Riducendo la capacità del modello.
- Fermando l'algoritmo prima che converga.
- Cambiando l'obiettivo con un termine di regolarizzazione.
- Aumentando la quantità di dati.
- Aggiungendo più campioni di training.

- Aumentando il training set con trasformazioni.
- Combinando predizioni da più modelli decorrelati o ensembling.

**2.4.6.1.1 Regolarizzazione** Si intende per regolarizzazione la modifica della funzione di training error con un termine  $\Omega(f)$  che penalizza soluzioni complesse:

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

## 2.5 Tipi di learning

### 2.5.1 Supervised learning

Nel supervised learning vengono dati in input a un modello o predittore un insieme di esempi che possiedono una label. Il modello poi impara a creare delle predizioni su un nuovo esempio.

#### 2.5.1.1 Dati

Nel caso del supervised learning i dati creano una distribuzione:

$$p_{data} \in \Delta(\mathcal{X} \times \mathcal{Y})$$

#### 2.5.1.2 Classificazione

In un problema di classificazione si trova un insieme finito di label discrete. In particolare dato un training set  $\mathcal{T} = \{(x_1, u_1), \dots, (x_m, y_m)\}$ , si deve imparare una funzione  $f$  per predire  $y$  dato  $x$ .  $f$  sarà pertanto:

$$f : \mathbb{R}^d \rightarrow \{1, 2, \dots, k\}$$

Dove  $d$  è la dimensionalità di  $x$  e  $k$  il numero di labels distinte.

**2.5.1.2.1 Task** Si deve pertanto trovare una funzione  $f \in \mathcal{Y}^{\mathcal{X}}$  che assegna ogni input  $x \in \mathcal{X}$  a una label discreta.

$$f(x) \in \mathcal{Y} = \{c_1, \dots, c_k\}$$

#### 2.5.1.3 Regression

Un problema di regressione presenta un insieme di label continue. Dato un training set  $\mathcal{T} = \{(x_1, y_1), \dots, (x_m, y_m)\}$ , si deve imparare una funzione  $f$  per predire  $y$  dato  $x$ .  $f$  sarà pertanto:

$$f : \mathbb{R}^d \rightarrow \mathbb{R}$$

Dove  $d$  è la dimensionalità di  $x$ .

**2.5.1.3.1 Task** Si deve trovare una funzione  $f(x) \in \mathcal{Y}$  che assegna ogni input a una label continua.

#### 2.5.1.4 Ranking

Il ranking è un tipo particolare di classificazione in cui una label è un ranking.

### 2.5.2 Unsupervised learning

Nel unsupervised learning vengono dati in input a un modello o predittore un insieme di esempi senza label. Il modello impara a creare delle predizioni su un nuovo esempio.

#### 2.5.2.1 Dati

Nel caso del supervised learning i dati creano una distribuzione:

$$p_{data} \in \Delta(\mathcal{X})$$

#### 2.5.2.2 Clustering

Nel clustering, data  $\mathcal{T} = \{x_1, \dots, x_m\}$  si deve trovare la struttura nascosta che intercorre tra le  $x$  o i clusters.

**2.5.2.2.1 Task** Si deve trovare una funzione  $f \in \mathbb{N}^{\mathcal{X}}$  che assegna ogni input  $x \in \mathcal{X}$  a un indice di cluster  $f(x) \in \mathbb{N}$ . Tutti i punti mappati sullo stesso indice formano un cluster.

#### 2.5.2.3 Dimensionality reduction

Nella dimensionality reduction si tenta di ridurre il numero di variabili sotto considerazione ottenendo un insieme di variabili principali.

**2.5.2.3.1 Task** Si deve trovare una funzione  $f \in \mathcal{Y}^{\mathcal{X}}$  che mappa ogni input di molte dimensioni  $x \in \mathcal{X}$  a un output a dimensione minore  $f(x) \in \mathcal{Y}$ , dove  $\dim(\mathcal{Y}) \ll \dim(\mathcal{X})$

### 2.5.3 Reinforcement learning

Nel reinforcement learning un agente impara dall'ambiente interagendo con esso e ricevendo premi per lo svolgimento di azioni particolari. In particolare, data una sequenza di esempi o stati e una reward dopo il completamento di tale sequenza si impara a predire l'azione da svolgere per uno stato o esempio individuale.

## 2.6 Polynomial curve fitting

### 2.6.1 Dati

L'insieme dei dati  $\mathcal{D}_n$  consiste di coppie:

$$\mathcal{D}_n = \{(x_1, y_1), \dots, (x_n, y_n)\}$$

I dati sono generati dalla funzione  $\sin(2\pi x)$  con del rumore. Il training set consiste di 10 punti:  $n = 10$ .

### 2.6.2 Modello e spazio di ipotesi

Il modello per il polynomial curve fitting è:

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

È parametrico, dove i parametri sono  $\{w_0, \dots, w_M\}$ . Lo spazio di ipotesi per  $M \in \mathbb{N}$  fissato è:

$$\mathcal{H}_M = \{f_w : w \in \mathbb{R}^M\}$$

### 2.6.3 Error function

La funzione di errore per il polynomial curve fitting consiste della media tra le distanze quadratiche tra la predizione effettuata e l'effettiva label. La pointwise loss:

$$l(f; (x_i, y_i)) = [f(x_i) - y_i]^2$$

La funzione di errore è pertanto:

$$E(f; \mathcal{D}_n) = \mathbb{E}(l(f; (x_i, y_i))) = \frac{1}{n} \sum_{i=1}^n [f(x_i) - y_i]^2$$

### 2.6.4 Funzione obiettivo

Si ricordi che la funzione obiettivo è:

$$f_{\mathcal{H}_M}^*(\mathcal{D}_n) \in \arg \min_{f \in \mathcal{H}_M} E(f; \mathcal{D}_n)$$

Equivalente rispetto a  $f_{w^*}$  dove:

$$w^* \in \arg \min_{w \in \mathbb{R}^M} \frac{1}{n} \sum_{i=1}^n [f_w(x_i) - y_i]^2$$

Si noti come questa richieda la risoluzione di un sistema lineare di equazioni. Il cambiamento di  $M$  influisce grandemente sull'accuratezza della predizione, andando a determinare casi di overfitting o underfitting. Con questo caso particolare si trova la soluzione migliore con  $M = 3$ .

### 2.6.5 Regularizzazione

Il polynomial curve fitting si regolarizza penalizzando i polinomi con grandi coefficienti:

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

Dove  $\|w\|^2 = \sum_{i=1}^n w_i^2$ .

## Capitolo 3

# KNN

### 3.1 Introduzione

Si possono considerare gli esempi come punti in uno spazio  $n$  dimensionale dove  $n$  è il numero di features. Per classificare un esempio  $d$  si può mettere a  $d$  una label uguale a quella dell'esempio più vicino a  $d$  nel training set. Questo concetto viene esteso nel  $K$ -nearest neighbour o  $K$ -NN in cui per classificare un esempio  $d$  si trovano i  $k$  esempi più vicini di  $d$  e si sceglie la label in maggior numero tra i  $k$  vicini più prossimi.

### 3.2 Misurare la distanza

Misurare la distanza tra due esempi è specifico al problema, ma un modo possibile è la distanza euclidea:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + \dots + (a_n - b_n)^2}$$

### 3.3 Decision boundaries

I decision boundaries sono posti nello spazio delle features dove la classificazione di un punto o un esempio cambia. In particolare  $K$ -NN definisce dei decision boundaries localmente tra le classi.

### 3.4 Il ruolo di $K$

I fattori che determinano la bontà di un algoritmo di machine learning sono la sua abilità di minimizzare il training error e minimizzare il gap tra il training error e il test error. Questi due fattori corrispondono a underfitting e overfitting.

#### 3.4.1 Underfitting

L'underfitting avviene quando il modello non è capace di ottenere un valore di errore abbastanza piccolo sul training set.

### 3.4.2 Overfitting

L'overfitting avviene quando il gap tra il training error e il test error è troppo grande.

## 3.5 Scelta di $K$

Il valore di  $K$  comprende euristiche comuni come 3, 5, 7 o un numero dispari per evitare pareggi. Può essere scelto utilizzando dati di sviluppo. Per classificare un esempio  $d$  si trovano i  $k$  vicini più prossimi di  $d$  e si sceglie la classe più presente nei  $k$  scelti.

## 3.6 Variazioni di $K$

Invece di scegliere i  $K$  vicini più prossimi si possono contare tutti gli esempi in una distanza fissata.

### 3.6.1 $K$ -NN pesata

Si può pesare il voto di tutti gli esempi in modo che esempi più vicini pesino di più. Si usa spesso qualche tipo di decadimento esponenziale.



## Capitolo 4

# Modelli lineari

### 4.1 Introduzione

Alcuni approcci del machine learning fanno delle forti assunzioni riguardo i dati. Questo avviene in quanto se le assunzioni sono vere si possono raggiungere performance migliori. In caso contrario l'approccio può fallire miseramente. Altri approcci che non fanno molte assunzioni riguardo i dati invece permettono di imparare da dati più vari ma sono più proni a overfitting e richiedono più dati di training.

#### 4.1.1 Bias

Il bias di un modello è quanto forte le assunzioni del modello sono. I classificatori a low-bias fanno delle assunzioni minime riguardo i dati come  $k$ -NN (che assume unicamente che la vicinanza è correlata alla classe) e  $DT$ . I classificatori a high-bias fanno assunzioni forti riguardo ai dati.

### 4.2 Linear separability

L'assunzione strong-bias dei modelli lineari è la linear separability, ovvero che in due dimensioni le classi si possano separare attraverso una linea, mentre in dimensioni maggiori da un hyperplane. Un modello lineare è pertanto un modello che assume che i dati sono linearmente separabili.

#### 4.2.1 Definire una linea

Ogni coppia di valori  $(w_1, w_2)$  definisce una linea attraverso l'origine:

$$0 = w_1 f_1 + w_2 f_2$$

Si può inoltre vedere il vettore  $\vec{w} = (w_1, w_2)$  come il vettore dei pesi perpendicolare alla linea. Per classificare i punti rispetto alla linea si considera il segno sostituendo i punti a  $f_1$  e  $f_2$ . La positività o negatività indica il lato della linea. Si può estendere l'equazione con:

$$a = w_1 f_1 + w_2 f_2$$

In questo modo la linea interseca l'asse delle  $y$  in  $a$ .

## 4.3 Definizione di modello lineare

Si definisce un modello lineare in uno spazio  $n$  dimensionale, dove  $n$  è il numero di features attraverso  $n + 1$  pesi.

$$0 = b + \sum_{i=1}^n w_i f_i$$

In un modello lineare si classifica un nuovo esempio moltiplicandolo con il vettore dei pesi, aggiungendo il bias e controllando il segno del risultato. Questo determina la classe dell'esempio.

### 4.3.1 Training

Il training di un modello lineare avviene online, ovvero a differenza del modo in batch in cui vengono dati i training data come  $\{(x_i, y_i) : 1 \leq i \leq n\}$ , i data points arrivano uno alla volta. L'algoritmo allora riceve un esempio  $x_i$  senza label, predice la classificazione di questo esempio e confronta la predizione con l'effettivo  $y_i$ . Infine aggiorna il proprio modello.

### 4.3.2 Perceptron

#### 4.3.2.1 Numero di iterazioni

Il numero di iterazioni del perceptron viene deciso in base alla convergenza. Inoltre può essere limitato in modo da ridurre l'overfitting. Si noti come in caso di dati non linearmente separabili la convergenza non avviene mai.

#### 4.3.2.2 Ordine dei campioni

I campioni da considerare nel perceptron sono considerati in ordine casuale. In questo modo si produce un modello a low bias.

#### 4.3.2.3 Linear separable sets

Le istanze di training sono linearmente separabili se esiste un hyperplane che separa le due classi.

#### 4.3.2.4 Algoritmo

---

```
: Perceptron()
repeat
  foreach training example  $(f_1, f_2, \dots, f_n, label)$  do
    %Label =  $\pm 1$ 
    check if it is correct based on the current label
    if not correct then
      %update all the weights
      foreach  $w_i$  do
         $w_i = w_i + f_i * label$ 
         $b = b + label$ 
until Convergence
%Or some number of iteration
```

---

#### 4.3.2.5 Calcolo della predizione

---

```
: Perceptron()
repeat
  foreach training example  $(f_1, f_2, \dots, f_n, label)$  do
    %Label =  $\pm 1$ 
    prediction =  $b + \sum_{i=1}^n w_i f_i$ 
    if prediction is not label then
      %update all the weights
      foreach  $w_i$  do
         $w_i = w_i + f_i * label$ 
         $b = b + label$ 
until Convergence
%Or some number of iteration
```

---

## 4.4 Perceptron e reti neurali

Si può immaginare il perceptron come un neurone artificiale o una funzione parametrizzata non lineare con un valore di attivazione soglia e un range di output ristretto. Le reti neurali sono reti di neuroni artificiali densamente connessi in modo da simulare la rete di neuroni del cervello.

### 4.4.1 Funzione di attivazione

Una funzione di attivazione può essere una soglia dura: se la somma di tutti gli input è maggiore di un valore allora il perceptron manda il segnale queste permettono di imparare solo modelli lineari. Funzioni di attivazione più interessanti sono le sigmoidi, tangenti iperboliche, ReLU o rectified linear unit o leaky ReLU.

### 4.4.2 Storia del perceptron

Il perceptron nasce nel 1958 da parte di Rosenblatt che lo crea con una soglia dura. Questo gli impedisce di imparare modelli non lineari come lo *xor*. Viene superato nel 1986 attraverso perceptrons multi layers e backpropagation e utilizzando una soglia meno dura.

## Capitolo 5

# Decision Trees

### 5.1 Struttura

Un decision tree è un modello di predizione con struttura ad albero. È composto da nodi terminali o foglie e nodi non terminali. I nodi non terminali hanno da due a più figli e implementano la funzione di routing. I nodi foglia non hanno figli e implementano la funzione di predizione. Non ci sono cicli e tutti i nodi hanno al massimo un genitore (con esclusione del nodo radice).

### 5.2 Funzionamento

Un decision tree prende un input  $x \in \mathcal{X}$  e lo ruta attraverso i nodi fino a che raggiunge un nodo foglia dove avviene la predizione. Ogni nodo non terminale

$$Node(\phi, t_L, t_R)$$

Contiene una funzione di routing  $\phi \in \{L, R\}^{\mathcal{X}}$ , un figlio destro  $t_L$  e un figlio sinistro  $t_R$ . Quando  $x$  raggiunge il nodo viene spostato sul figlio destro o sinistro in base al valore di  $\phi(x) \in \{L, R\}$ . Ogni nodo foglia

$$Leaf(h)$$

Contiene una funzione di predizione  $h \in \mathcal{F}_{task}$ , tipicamente una costante.

#### 5.2.1 Inferenza

Sia  $f_t$  la funzione che ritorna la predizione per l'input  $x \in \mathcal{X}$  secondo il decision tree  $t$ . Questa viene definita come:

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

### 5.3 Decision trees learning algorithm

Dato un training set  $\mathcal{D}_n = \{z_1, \dots, z_n\}$  si deve trovare  $f_{t^*}$  dove:

$$t^* \in \arg \min_{t \in \mathcal{T}} E(f_t; \mathcal{D}_n)$$

Dove  $\mathcal{T}$  è l'insieme dei decision trees. Il problema di ottimizzazione è facile se non si impongono constraints, altrimenti potrebbe diventare *NP-hard*. Una soluzione può essere trovata utilizzando una strategia greedy. Pertanto si assume:

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

Ora fissato un insieme di predizioni di foglie

$$\mathcal{H}_{leaf} \subset \mathcal{F}_{task}^*$$

E fissato un insieme di possibili funzioni di routing o split

$$\Phi \subset \{L, R\}^{\mathcal{X}}$$

La strategia di crescita dell'albero partiziona ricorsivamente il training set e decide se crescere foglie o nodi non terminali.

### 5.3.1 Crescere una foglia

Sia  $\mathcal{D} = \{z_1, \dots, z_m\}$  il training set che raggiunge un nodo. Il predittore di foglia ottimale viene computato come:

$$h_{\mathcal{D}}^* \in \arg \min_{h \in \mathcal{H}_{leaf}} E(h; \mathcal{D})$$

Il valore di errore ottimale o misura di impurità:

$$I(\mathcal{D}) = E(h_{\mathcal{D}}^*; \mathcal{D})$$

Dove  $h_{\mathcal{D}}$  è la predizione per il dataset  $\mathcal{D}$ . L'impurità è computata nel nodo in cui arriva il sottoinsieme del dataset originale  $\mathcal{D}$ . In quel nodo si calcola il numero di errori che si farebbero classificando tutti i dati del sottoinsieme in una singola classe  $c$ , ripetendo il calcolo per ogni classe. Il numero minimo di errori che fai con una certa classe  $c^*$  è la misura di impurità del classification error. Se si raggiungono dei criteri si cresce una foglia  $Leaf(h_{\mathcal{D}}^*)$ . Esempi di questi criteri sono la purezza  $I(\mathcal{D}) < \epsilon$ , la cardinalità minima  $|\mathcal{D}| < k$  o un'altezza massima dell'albero.

### 5.3.2 Crescere un nodo

Se non si raggiunge il criterio si deve trovare una funzione di split ottimale:

$$\phi_{\mathcal{D}}^* \in \arg \min_{\phi \in \Phi} I_{\phi}(\mathcal{D})$$

L'impurità  $I_{\phi}(\mathcal{D})$  di una funzione di split  $\phi$  dato il training set  $\mathcal{D}$  viene computata nei termini di impurità dei dati splittati:

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

Dove

$$\mathcal{D}_d^{\phi} = \{(x, y) \in \mathcal{D}; \phi(x) = d\}$$

L'impurità di una funzione di split è il più basso errore di training che può essere ottenuto da un albero che consiste di una radice e due figlie. Si cresce pertanto un nodo  $Node(\phi^*, t_L, t_R)$  dove  $\phi^*$  è lo split ottimale, mentre  $t_L$  e  $t_R$  sono ottenuti applicando ricorsivamente l'algoritmo di learning ai training set splits.

### 5.3.3 Algoritmo

$$Grow(\mathcal{D}) = \begin{cases} Leaf(h_{\mathcal{D}}^*) & \text{raggiunto criterio di stop} \\ Node(\phi_{\mathcal{D}}^*, Grow(\mathcal{D}_L^*), Grow(\mathcal{D}_R^*)) & \text{altrimenti} \end{cases}$$

Dove  $\mathcal{D}_d^* = \{(x, y) \in \mathcal{D}; \phi_{\mathcal{D}}^*(x) = d\}$ .

### 5.3.4 Split selection

Tipicamente la migliore funzione di split viene data in termini di minimizzazione dell'impurità dello split, ma altre volte nella massimizzazione del guadagno di informazioni o

$$\Delta_{\phi}(\mathcal{D}) = I(\mathcal{D}) - I_{\phi}(\mathcal{D})$$

Essendo  $\Delta_{\phi}(\mathcal{D}) \geq 0$  per ogni  $\phi \in \{L, R\}^{\mathcal{X}}$  e ogni training set  $\mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$ , l'impurità non aumenterà mai per ogni split scelto casualmente.

### 5.3.5 Predizione delle foglie

La predizione delle foglie fornisce una soluzione a un problema semplificato coinvolgendo solo dati che la raggiungono. Questa soluzione può essere una funzione arbitraria  $h \in \mathcal{F}_{task}$ , ma in pratica si restringe a un sottoinsieme di  $\mathcal{H}_{leaf}$ . Il predittore più semplice è una funzione che ritorna una costante (come una label). L'insieme di tutte le possibili funzioni costanti può essere scritto come:

$$\mathcal{H}_{leaf} = \bigcup_{y \in \mathcal{Y}} \{y\}^{\mathcal{X}}$$

## 5.4 Misure di impurità per la classificazione

Si consideri per la classificazione:

$$\mathcal{Y} = \{c_1, \dots, c_k\} \quad \mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$$

Sia  $\mathcal{D}^y = \{(x, y') \in \mathcal{D} : y = y'\}$ , che denota il sottoinsieme di training samples in  $\mathcal{D}$  con label  $y$ . Considerando la funzione di errore:

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

Se  $l(f; (x, y)) = 1_{f(x) \neq y}$  e  $\mathcal{H}_{leaf} = \bigcup_{y \in \mathcal{Y}} \{y\}^{\mathcal{X}}$  la misura di impurità è allora il classification error:

$$I(\mathcal{D}) = 1 - \max_{y \in \mathcal{Y}} \frac{|\mathcal{D}^y|}{|\mathcal{D}|}$$

Se invece  $l(f; (x, y)) = \sum_{c \in \mathcal{Y}} [f_c(x) - 1_{c=y}]^2$  e  $\mathcal{H}_{leaf} = \bigcup_{\pi \in \Delta(\mathcal{Y})} \{\pi\}^{\mathcal{X}}$  allora la misura di impurità è l'impurità di Gini:

$$I(\mathcal{D}) = 1 - \sum_{y \in \mathcal{Y}} \left( \frac{|\mathcal{D}^y|}{|\mathcal{D}|} \right)^2$$

Infine se  $l(f; (x, y)) = -\log f_y(x)$  e  $\mathcal{H}_{leaf} = \bigcup_{\pi \in \Delta(\mathcal{Y})} \{\pi\}^{\mathcal{X}}$ , con una distribuzione costante di label come predizione di foglie, allora la misura di impurità è l'entropia:

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

## 5.5 Misure di impurità per la regressione

Si consideri per la regressione:

$$\mathcal{Y} \subset \mathbb{R}^d \quad \mathcal{D} \subset \mathcal{X} \times \mathcal{Y}$$

Se  $l(f; (x, y)) = \|f(x) - y\|^2$  e  $\mathcal{H}_{leaf} = \bigcup_{y \in \mathcal{Y}} \{y\}^{\mathcal{X}}$  allora la misura di impurità è la varianza:

$$I(\mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{(x, y) \in \mathcal{D}} \|x - \mu_{\mathcal{D}}\|^2$$

Dove  $\mu_{\mathcal{D}} = \frac{1}{|\mathcal{D}|} \sum_{(x, y) \in \mathcal{D}} x$

## 5.6 Data features e attributi

Un data point  $x \in \mathcal{X}$  potrebbe essere  $d$  dimensionale con ogni dimensione con tipi di valori eterogenei come discreti o continui e avere un ordinamento o no, rispettivamente ordinali o nominali.

## 5.7 Funzioni di split o routing

Il routing o split  $\phi \in \{L, R\}^{\mathcal{X}}$  determina se un data point  $x \in \mathcal{X}$  deve muoversi a destra o sinistra. La possibile funzione di split è ristretta in un insieme predefinito  $\Phi \subset \{L, R\}^{\mathcal{X}}$  in base alla natura dello spazio di features. La funzione di split prototipica per un input  $d$  dimensionale prima seleziona una dimensione e poi applica un criterio di split 1 dimensionale.

### 5.7.1 Features discrete e nominali

Si assumano features discrete e nominali con valori in  $\mathcal{K}$ . La funzione di split può essere implementata data una partizione di  $\mathcal{K}$  in  $\mathcal{K}_R$  e  $\mathcal{K}_L$ :

$$\phi(x) = \begin{cases} L & \text{if } x \in \mathcal{K}_L \\ R & \text{if } x \in \mathcal{K}_R \end{cases}$$

Trovare lo split ottimale richiede testare  $2^{|\mathcal{K}|-1} - 1$  bi-partizioni.

### 5.7.2 Features ordinali

Si assumano features ordinali con valori in  $\mathcal{K}$ . La funzione di split può essere implementata dando una soglia  $r \in \mathcal{K}$ :

$$\phi(x) = \begin{cases} L & \text{if } x \leq r \\ R & \text{if } x > r \end{cases}$$

Se  $|\mathcal{K}| \leq |\mathcal{D}|$  trovare lo split ottimale richiede il test di  $|\mathcal{K}| - 1$  soglie. Se  $|\mathcal{K}| > |\mathcal{D}|$  si deve ordinare i valori di input in  $\mathcal{D}$  dove  $\mathcal{D}$  è il training set che raggiunge il nodo e testare  $|\mathcal{D}| - 1$  soglie.

### 5.7.3 Obliquo

A volte è conveniente fare split considerando più features alla volta. Tali funzioni lavorano con features continue e sono dette oblique in quanto generano decision boundaries obliqui. Se  $x \in \mathbb{R}^d$  allora la funzione di split può essere implementata dato  $w \in \mathbb{R}^d$  e  $r \in \mathbb{R}$ :

$$\phi(x) = \begin{cases} L & \text{if } w^T x \leq r \\ R & \text{altrimenti} \end{cases}$$

Si nota come questa funzione sia più difficile da ottimizzare.

## 5.8 Decision trees e overfitting

I decision trees sono modelli non parametrici con una struttura determinata dai dati. Per questo sono flessibili e possono facilmente fare fit sul training set, con un alto rischio di overfitting. Tecniche standard per migliorare la generalizzazione si applicano ai decision trees:

- Early stopping.
- Regularization.
- Data augmentation.
- Complexity reduction.
- Ensembling.

Una tecnica per ridurre la complessità a posteriori è detta pruning.

## 5.9 Random forest

Le random forest sono ensembles di decision trees. Ogni albero è tipicamente trained con una versione bootstrapped del training set campionata con sostituzione. Le funzioni di split sono ottimizzate su features campionate a caso o completamente a caso (extremely randomized trees). Questo aiuta ad ottenere decision trees decorrelati. La predizione finale della foresta è ottenuta facendo la media delle predizioni per ogni albero nell'ensemble  $\mathcal{Q} = \{t_1, \dots, t_T\}$ .

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



## 5.10 Confronto con KNN

Si nota come a differenza dei decision trees KNN non richiede nessun training, ma la classificazione è più veloce per i decision trees in quanto KNN per ogni esempio deve calcolare  $k$  distanze. A differenza di KNN che tratta tutte le features in maniera uguale i decision trees permettono una selezione delle features più importanti facendo scelte pesate.

## Capitolo 6

# Multi class classification

### 6.1 Introduzione

#### 6.1.1 Classificazione binaria

Si è definita la classificazione binaria come la task in cui dati:

- Uno spazio di input  $\mathcal{X}$ .  $\{-1, +1\}$ .
- Una distribuzione sconosciuta  $\mathcal{D}$  su  $\mathcal{X} \times \{-1, +1\}$ . • Un training set  $D$  campionato da  $\mathcal{D}$

Si deve computare una funzione  $f$  che minimizza  $\mathbb{E}_{(x,y) \sim \mathcal{D}}[f(x) \neq y]$

#### 6.1.2 Classificazione multi classe

La classificazione multiclasse è l'estensione naturale della classificazione binaria. L'obiettivo è quello di assegnare una label discreta a degli esempi. La differenza è che ora ci sono  $k > 2$  classi da cui scegliere. Dati pertanto:

- Uno spazio di input  $\mathcal{X}$  e un numero di classi  $K$ .  $[K]$ .
- Una distribuzione sconosciuta di  $\mathcal{D}$  su  $\mathcal{X} \times [K]$ . • Un training set  $D$  campionato da  $\mathcal{D}$ .

Si deve computare una funzione  $f$  che minimizza  $\mathbb{E}_{(x,y) \sim \mathcal{D}}[f(x) \neq y]$ .

##### 6.1.2.1 K nearest neighbours

Si noti come una  $K$ -NN per classificare un esempio  $d$  trova i  $k$  vicini di  $d$  e sceglie la label in presenza maggiore tra i  $k$  vicini più prossimi. Non necessita pertanto di cambi algoritmici nel caso della multi class classification.

##### 6.1.2.2 Decision tree

I decision tree non richiedono cambi algoritmici per la multi class classification.

### 6.1.3 Approccio black box alla multi class classification

Dato un classificatore binario questo si può usare per risolvere il problema della multiclass classification. Si noti come un perceptron oltre al risultato può anche dare un punteggio di confidenza. Inoltre siccome una linea non è sufficiente per dividere le classi se ne possono usare diverse.

## 6.2 One versus all OVA

Nell'approccio OVA nel training si definisce per ogni label  $L$  un problema binario in cui:

- Tutti gli esempi con la label  $L$  sono positivi.
- Tutti gli altri esempi sono negativi.

In pratica si imparano  $L$  diversi modelli di classificazione. Si ricordi come il classificatore divide il piano in due semipiani.

### 6.2.1 Ambiguità

In questo caso si formano pertanto delle zone in cui si creano delle ambiguità. Se il classificatore non fornisce confidence e c'è ambiguità si sceglie una delle label in conflitto. Nella maggior parte dei casi i classificatori forniscono confidence, allora in questo caso si:

- Si sceglie il positivo con confidence maggiore.
- Se nessuno è positivo si sceglie il negativo con confidence minore.

La confidence nel perceptron si calcola come distanza dall'iperpiano stabilito dalla prediction.

### 6.2.2 Algoritmi

---

```
: OneVersusAllTrain( $D^{multiclass}$ , BinaryTrain())
```

---

```
for  $i = 1$  to  $K$  do  
     $D^{bin} = \text{relabel } D^{multiclass}$  in modo che  $i$  è positivo e  $\neq i$  è negativo  
     $f_i = \text{BinaryTrain}(D^{bin})$   
Return  $f_1, \dots, f_K$ 
```

---

```
: OneVersusAllTest( $f_1, \dots, f_K, \hat{x}$ )
```

---

```
score =  $\langle 0, \dots, 0 \rangle$  %Inizializza  $K$  score a 0  
for  $i = 1$  to  $K$  do  
     $y = f_i(\hat{x})$   
     $score_i = score_i + y$   
Return  $\max(score)$ 
```

---

## 6.3 All versus all AVA

Un approccio alternativo consiste nel gestire il problema della classificazione multi classe componendolo in problemi di classificazione binaria. Questo approccio viene detto anche all pairs. Si

### 6.3. ALL VERSUS ALL AVA

---

classificano  $\frac{K(K-1)}{2}$  classificatori in modo che

$$F_{ij}, 1 \leq i < j \leq K$$

Sia il classificatore che discrimina la classe  $i$  contro la classe  $j$ . Questo classificatore riceve tutti gli esempi della classe  $i$  come positivi e tutti gli esempi della classe  $j$  come negativi. Quando arriva un punto di test si valuta su tutti i classificatori  $F_{ij}$ . Ogni volta che  $F_{ij}$  predice positivo la classe  $i$  prende un voto, altrimenti lo prende  $j$ . Dopo aver eseguito tutti i  $\frac{K(K-1)}{2}$  classificatori la classe con più voti decide la label.

#### 6.3.1 AVA training

Per ogni coppia di label si addestra un classificatore che le distingue:

---

```
: AllVersusAllTrain()
```

---

```
  for  $i = 1$  to number of labels do
```

```
    for  $j = i + 1$  to number of labels do
```

```
      train a classifier  $F_{ij}$  to distinguish between  $label_j$  and  $label_i$ 
```

```
      create a dataset with all examples with  $label_j$  labeled positive and with  $label_i$  negative
```

```
      Train classifier  $F_{ij}$  su questo sottoinsieme di dati
```

---

#### 6.3.2 AVA classification

Per classificare un esempio  $x$  lo si classifica per ogni classificatore  $F_{ij}$ . Per scegliere la classe finale si può:

- Considerare la maggioranza.  $score_i = y$ .
- Considerare un voto pesato basato sulla confidence:
  - $y = F_{ij}(x)$ .
  - $score_j = y$ .

Lo score viene cambiato in quanto se  $y$  è positivo il classificatore lo pensa di tipo  $j$ , se negativo lo pensa di tipo  $i$  e pertanto lo score viene aggiornato di conseguenza.

#### 6.3.3 Algoritmi

---

```
: AneVersusAllTrain( $D^{multiclass}$ , BinaryTrain())
```

---

```
 $f_{ij} = \emptyset, \forall 1 \leq i < j \leq K$ 
```

```
for  $i = 1$  to  $K - 1$  do
```

```
   $D^{pos} =$  all  $x \in D^{multiclass}$  labeled  $i$ 
```

```
  for  $j = i + 1$  to  $K$  do
```

```
     $D^{neg} =$  all  $x \in D^{multiclass}$  labeled  $j$ 
```

```
     $D^{bin} = \{(x, +1) : x \in D^{pos}\} \cup \{(x, -1) : x \in D^{neg}\}$ 
```

```
     $f_{ij} =$  BinaryTrain( $D^{bin}$ )
```

```
Return all  $f_{ij}$ 
```

---

---

```
: AllVersusAllTest(all  $f_{ij}$ ,  $\hat{x}$ )  
score =  $\langle 0, \dots, 0 \rangle$  %Inizializza  $K$  score a 0  
for  $i = 1$  to  $K - 1$  do  
    for  $j = i + 1$  to  $K$  do  
         $y = f_{ij}(\hat{x})$   
         $score_i = score_i + y$   
         $score_j = score_j - y$   
Return  $\max(score)$ 
```

---

## 6.4 Confronto tra OVA e AVA

### 6.4.1 Tempo di training

AVA impara più classificatori ma il training set è molto più piccolo pertanto tende ad essere più veloce se le label sono equamente bilanciate.

### 6.4.2 Tempo di test

Avendo AVA più classificatori è tipicamente più lenta.

### 6.4.3 Errori

AVA fa training con data sets più bilanciata, ma avendo più classificatori i test tendono ad avere più possibilità di errori.

## 6.5 Riassunto

Se vengono usati classificatori binari viene tipicamente utilizzata OVA, altrimenti si usa un classificatore che permette label multiple come DT o K-NN, nonostante altri metodi più sofisticati siano meglio.

## 6.6 Multiclass evaluation

Per computare l'accuratezza nella predizione di una classe  $c$  si può usare la misura di predizione  $Pr = \frac{TP}{TP+FP}$  dove  $TP$  sono le predizioni corrette e  $FP$  le predizioni scorrette.

### 6.6.1 Microaveraging

Nel microaveraging si fa la media sugli esempi. Considerando  $n$  classi si calcola come:

$$Pr = \frac{\sum_{i=1}^n TP_{c_i}}{\sum_{i=1}^n (TP_{c_i} + FP_{c_i})}$$

### 6.6.2 Macroaveraging

Nel macroaveraging si calcola lo score di valutazione o accuratezza per ogni label e poi si fa la media tra le label. Questo in quanto dà più enfasi a label più rare e permette un'altra dimensione di analisi.

$$Pr = \frac{\sum_{c \in C} Pr_c}{|C|}$$

### 6.6.3 Confusion matrix

La confusion matrix è una matrice in cui  $(i, j)$  rappresenta il numero di esempi con label  $i$  predetti avere label  $j$ . Viene spesso espressa come percentuale. Nel caso di una classificazione a  $k$  classi sarà di dimensione  $k \times k$ . La performance è buona nel caso in cui la diagonale presenti le percentuali più alte.

## Capitolo 7

# Ranking

### 7.1 Classificazione multiclasse e multilabel

Nella classificazione multi classe ogni esempio ha esattamente una label che sono pertanto mutualmente esclusive. Nella classificazione multi label ogni esempio ha zero o più labels, dette anche annotazioni. Per svolgere una classificazione multi label basterebbe fare training su un modello per ogni label e applicarli a ogni nuovo esempio, ma ci sono altri metodi più sofisticati ed efficaci come il joint learning.

### 7.2 Problema del ranking

I dati di training sono divisi in  $K$  categorie ognuna delle quali corrispondente a un ranking. Si deve insegnare a un modello che quando riceve un insieme di esempi li fitti nel ranking corretto o ritorni un ordinamento per questo nuovo esempio.

#### 7.2.1 Preference function

La preference function o binary classifier è un'implementazione di ranking. Data una query  $q$  e due campioni  $x_i$  e  $x_j$  il classificatore predice se  $x_i$  deve essere preferito a  $x_j$  rispetto alla query  $q$ . Il classificatore prende pertanto due campioni e dà in output 1 se il primo è più alto o  $-1$  se è più basso. In questo modo si ottiene una funzione di ordinamento atomica che si può estendere a diversi esempi.

##### 7.2.1.1 Perceptron

Per implementare questo algoritmo si può utilizzare il perceptron creando il vettore delle feature combinate dei due esempi:

$$f'_i = a_i - b_i$$
$$f'_i = \begin{cases} 1 & \text{if } a_i > b_i \\ 0 & \text{altrimenti} \end{cases}$$

Questa funzione viene sviluppata in maniera dipendente dall'applicazione. Un modo per computare il ranking numerico per i campioni può essere risolto utilizzando la preference function sommando i valori di ritorno di ogni classificatore e ottenendo uno score per ognuno di essi.

## 7.3 Utilizzo del ranking e della preference function

Con gli algoritmi visti precedentemente si potrebbe pesare il ranking di un esempio utilizzando la distanza dagli altri. Si possono usare diversi metodi di distanza dati che sono consistenti. La distanza a tempo di testing è calcolata come la confidenza della predizione del perceptron e dà un ordinamento degli esempi. In questo modo si ottiene una forma di ranking più precisa rispetto a quella vista prima. Per un algoritmo di ranking sofisticato si devono incorporare queste osservazioni a tempo di training: se un problema ritorna un'alta differenza in preferenza tra due esempi dovrebbe avere un peso più alto.

## 7.4 Naive Ranking

### 7.4.1 Training

---

```
: NaiveRankTrain(RankingData, BinaryTrain)
```

---

```

 $D = []$ 
for  $n = 1$  to  $N$  do
  for all  $i, j = 1$  to  $M$  and  $i \neq j$  do
    if  $i$  is preferred to  $j$  on query  $n$  then
       $D = D \oplus (x_{nij}, -1)$ 
return BinaryTrain( $D$ )
```

---

### 7.4.2 Testing

---

```
: NaiveRankTest( $f, \hat{x}$ )
```

---

```

 $score = \langle 0, \dots, 0 \rangle$ 
for all  $i, j = 1$  to  $M$  and  $i \neq j$  do
   $y = f(\hat{x}_{ij})$ 
   $score_i = score_i + y$ 
   $score_j = score_j - y$ 
return ArgSort( $score$ )  $D = []$ 
for  $n = 1$  to  $N$  do
  for all  $i, j = 1$  to  $M$  and  $i \neq j$  do
    if  $i$  is preferred to  $j$  on query  $n$  then
       $D = D \oplus (x_{nij}, -1)$ 
return BinaryTrain( $D$ )
```

---

## 7.5 Bipartite ranking

Gli algoritmi di ranking bipartito risalgono problemi in cui si tenta di predire una risposta binaria. L'obiettivo è di assicurare che tutti gli esempi rilevanti si trovano prima di quelli irrilevanti. Non si trova un ordinamento tra esempi rilevanti.



## 7.6 Ordinamento e $\omega$ -ranking

Nonostante la veloce soluzione dell'ordinamento contro la funzione di preference si può utilizzarla come funzione di sorting. Si definisce un ranking come una funzione  $\sigma$  che mappa gli oggetti alla posizione nella lista di ranking desiderata  $(1, \dots, M)$ . Se  $\sigma_u < \sigma_v$  allora  $u$  è preferito a  $v$ . Dati i dati con ranking osservati  $\sigma$  l'obiettivo è di imparare a predire i ranking per nuovi oggetti  $\sigma^*$ . Si definisce  $\sum_M$  l'insieme di tutti gli ordinamenti di ranking in  $M$ . Si vuole modellare il fatto che uno sbaglio su alcune coppie è peggiore rispetto ad altre, pertanto si implementa una nuova funzione di errore per questo scopo. Si definisce una funzione di costo  $\omega$  dove  $\omega(i, j)$  è il costo di mettere qualcosa nella posizione  $j$  quando dovrebbe essere in  $i$ . Tale funzione deve essere:

- Simmetrica:  $w(i, j) = w(j, i)$ .
- Monotona:  $i < j < l \vee i > j > k \Rightarrow \omega(i, j) \leq \omega(i, k)$ .
- Soddisfa la disuguaglianza triangolare:  $\omega(i, j) + \omega(j, k) \leq \omega(i, k)$ .

Un esempio potrebbe essere:

$$\omega(i, j) = \begin{cases} 1 & \text{if } \{i, j\} \leq K \wedge i \neq j \\ 0 & \text{altrimenti} \end{cases}$$

Questa viene detta task di  $\omega$  ranking. Dati:

- Uno spazio di input  $\mathcal{X}$ .
- Una distribuzione sconosciuta  $\mathcal{D}$  su  $\mathcal{X} \times \sum_M$ .
- Un training set  $D$  campionato da  $\mathcal{D}$ .

Si deve computare una funzione  $f : \mathcal{X} \rightarrow \sum_M$  che minimizzi:

$$\mathbb{E}_{(\mathcal{X}, \sigma) \sim \mathcal{D}} \left[ \sum_{u \neq v} [\sigma_u < \sigma_v] [\hat{\sigma}_v < \hat{\sigma}_u] \omega(\sigma_u, \sigma_v) \right]$$

Dove  $\hat{\sigma} = f(x)$ .

### 7.6.1 Testing

A tempo di testing invece di predire score e poi ordinare la lista come negli algoritmi prima si utilizza un algoritmo di ordinamento utilizzando la funzione imparata come funzione di ordinamento. In pratica ad ogni passo si sceglie un pivot  $p$  e ogni oggetto  $u$  viene confrontato con  $p$  utilizzando la funzione imparata e ordinato a destra o a sinistra. La differenza è che qua la funzione di comparazione è probabilistica.

## 7.6.2 Implementazione

### 7.6.2.1 Algoritmo di train

---

```
: OmegaRankTest( $D^{rank}$ ,  $\omega$ , BinaryTrain)
 $D^{bin} = []$  for all  $(x, \sigma) \in D^{rank}$  do
    for all  $u \neq v$  do
         $y = \text{Sign}(\sigma^v, \sigma_u)$ 
         $w = w(\sigma_u, \sigma_v)$ 
         $D^{bin} = D^{bin} \oplus (y, w, x_{uv})$ 
return BinaryTrain( $D^{bin}$ )
```

---

### 7.6.2.2 Algoritmo di test

---

```
: OmegaRankTest( $f$ ,  $\hat{x}$ ,  $obj$ )
if  $obj$  contains 0 or 1 elements then
    return  $obj$ 
else
     $p$  =randomly chosen object in  $obj$ 
     $left = []$ 
     $right = []$ 
    for all  $u \in obj \setminus \{p\}$  do
         $\hat{y} = f(x_{up})$ 
        if uniform random variable  $< \hat{y}$  then
             $left = left \oplus u$ 
        else
             $right = right \oplus u$ 
     $left = \text{OmegaRankTest}(f, \hat{x}, left)$ 
     $right = \text{OmegaRankTest}(f, \hat{x}, right)$ 
    return  $left \oplus \langle p \rangle \oplus right$ 
```

---

## Capitolo 8

# Gradient descent

### 8.1 Model based machine learning

Nel model based machine learning si sceglie un modello definito da un insieme di parametri. In particolare si nota come:

- Per i decision trees i parametri sono la struttura dell'albero, quali features ogni nodo divide e le predizioni delle foglie.
- Per il perceptron i parametri sono i pesi e il valore di  $b$ .

Dopo aver scelto il modello si deve scegliere un criterio da ottimizzare o la funzione obiettivo come per esempio il training error. Infine si sviluppa un algoritmo di learning che deve cercare di minimizzare il criterio, spesso in maniera euristica.

#### 8.1.1 Modelli lineari

Nei modelli lineari il modello è:

$$0 = b + \sum_{j=1}^m w_j f_j$$

Si deve scegliere il criterio da ottimizzare.

##### 8.1.1.1 Notazioni

**8.1.1.1.1 Funzione indicatrice** Una funzione indicatrice trasforma valori di *Vero* e *Falso* in numeri e conte.

$$1[x] = \begin{cases} 1 & \text{if } x = \text{True} \\ 0 & \text{if } x = \text{False} \end{cases}$$

**8.1.1.1.2 Dot-product** Utilizzando una notazione vettoriale si rappresenta un esempio  $f_1, \dots, f_m$  come un vettore singolo  $\vec{x}$  in cui  $j$  indicizza la feature e  $i$  indicizza un dataset di esempi. Si possono rappresentare anche i pesi  $w_1, \dots, w_m$  come un vettore  $\vec{w}$ . Il dot-product tra due vettori  $a$  e  $b$  viene definito come:

$$a \cdot b = \sum_{j=1}^m a_j b_j$$

### 8.1.1.2 Funzione obiettivo

Il criterio da ottimizzare o funzione obiettivo può essere:

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

Si devono pertanto trovare  $w$  e  $b$  tali che minimizzano questa funzione, ovvero:

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

## 8.2 Loss functions

### 8.2.1 Loss 0/1

Una funzione di loss 0/1 è una funzione nella forma:

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

Dove tra le quadre si trova se la predizione e la label sono d'accordo, con vero se non lo fanno e tra le tonde la distanza dall'iperpiano, di cui il segno è la predizione. Questa funzione ritorna il numero di sbagli.

#### 8.2.1.1 Minimizzare la loss 0/1

Per minimizzare una funzione 0/1 si deve, ogni volta cambiare un valore di  $w$  in modo che l'esempio è corretto o scorretto la perdita aumenta o diminuisce. Si nota come a ogni feature aggiunta si aggiunge una nuova dimensione allo spazio. Il minimo si trova trovando  $w$  e  $b$  che minimizzano la perdita. Questo è un problema *NP-hard*. Sue difficoltà comprendono il fatto che piccoli cambi in ogni  $w$  possono portare a grandi cambi nella perdita in quanto il cambio non è continuo. Ci possono essere molti minimi locali. Ad ogni punto non si hanno informazioni che direzionano verso il minimo. Pertanto si nota come una loss function ideale sia continua e differenziabile in modo da avere un'indicazione verso la direzione di minimizzazione e un unico minimo.

**8.2.1.1.1 Loss function ideale** Una loss function ideale dovrebbe essere:

- Continua in modo da ottenere informazioni riguardo la direzione della minimizzazione.
- Avere un solo minimo.
- Misurare la distanza tra la predizione reale e quella predetta.

### 8.2.2 Funzioni convesse

In una funzione convessa il segmento tra qualsiasi due punti della funzione si trova al di sopra della funzione.

### 8.2.3 Surrogate loss function

Per molte applicazioni si vuole minimizzare la loss 0/1. Una surrogate loss function è una loss function che fornisce un limite superiore alla loss function attuale. Si vuole identificare un surrogato convesso della loss function in modo da facilitarne la minimizzazione. Chiave a una loss function è come verifica la differenza tra la label  $y$  effettiva e la predizione  $y'$ .

#### 8.2.3.1 Alcune surrogate loss function

- 01 loss:  $l(y, y') = 1[y y' \leq 0]$ .
- Exponential:  $l(y, y') = \exp(-y y')$
- Hinge  $l(y, y') = \max(0, 1 - y y')$ .
- Squared loss:  $l(y, y') = (y - y')^2$ .

## 8.3 Gradient descent

Il gradient descent è un modo per trovare il minimo di una funzione: le derivate parziali danno un slope o direzione dove muoversi in tale dimensione. Questo approccio consiste di scegliere un punto di partenza e a ripetizione di: scegliere una dimensione e muoversi di una piccola quantità verso il minimo utilizzando la derivata. Pertanto si:

- Sceglie un punto di inizio  $w$ .
- Sceglie una dimensione.
- si muove di una piccola quantità verso la diminuzione della loss utilizzando la derivata.

Questo ciclo si ripete fino a che la loss non diminuisce in nessuna dimensione.

### 8.3.1 Spostamento in direzione della minimizzazione dell'errore

Il movimento in direzione della minimizzazione dell'errore è pertanto:

$$w_j = w_j - \eta \frac{d}{dw_j} \text{loss}(w)$$

Dove  $\eta$  è il learning rate.

#### 8.3.1.1 Calcolo dello spostamento per la loss function esponenziale

Si deve pertanto calcolare:

$$\begin{aligned} \frac{d}{dw_j} \text{loss} &= \frac{d}{dw_j} \sum_{i=1}^n \exp(-y_i(w \cdot x_i + b)) = \\ &= \sum_{i=1}^n \frac{d}{dw_j} [-y_i(w \cdot x_i + b)] \exp(-y_i(w \cdot x_i + b)) \end{aligned}$$

Si consideri pertanto ora:

$$\begin{aligned}\frac{d}{dw_j}[-y_i(w \cdot x_i + b)] &= -\frac{d}{dw_j}[-y_i(w \cdot x_i + b)] = \\ &= -\frac{d}{dw_j}y_i(w_1x_{i1} + \dots + w_mx_{im} + b) = \\ &= -y_ix_{ji}\end{aligned}$$

Si nota pertanto come:

$$\frac{d}{dw_j}loss = \sum_{i=1}^n -y_ix_{ij} \exp(-y_i(w \cdot x_i + b))$$

Si aggiorna pertanto  $w_j$ :

$$w_j = w_j - \eta \sum_{i=1}^n -y_ix_{ij} \exp(-y_i(w \cdot x_i + b))$$

Questo viene fatto per ogni esempio  $x_i$ .

### 8.3.2 Learning algorithm del perceptron

Si nota pertanto come considerando il perceptron nell'ambito del gradient descent si aggiorna sempre il vettore dei pesi.

---

**: Perceptron()**

---

```
repeat
  foreach training example  $(f_1, f_2, \dots, f_n, label)$  do
    %Label =  $\pm 1$ 
    prediction =  $b + \sum_{i=1}^n w_i f_i$ 
    foreach  $w_i$  do
       $w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b))$   $b = b + label$ 
until Convergence
```

---

Si noti come in questo caso  $\eta$  rappresenta il learning rate,  $y_i$  la label e  $(w \cdot x_i + b)$  la predizione. Questi generano una costante  $c$

### 8.3.3 Costante c

Nella costante  $c$  se label e predizione hanno lo stesso segno quando gli elementi predetti aumentano gli aggiornamenti diventano minori. Se invece sono diversi più diversi lo sono, maggiore l'aggiornamento.

### 8.3.4 Gradiente

Il gradiente è il vettore delle derivate parziali rispetto a tutte le coordinate dei pesi:

$$\nabla L = \left[ \frac{\partial L}{\partial w_1} \dots \frac{\partial L}{\partial w_N} \right]$$

### 8.3. GRADIENT DESCENT

---

Ogni derivata parziale misura quanto veloce la perdita cambia in una direzione. Quando il gradiente è zero la perdita non sta cambiando in nessuna direzione.

---

**: GradientDescent**( $\mathcal{F}, K, \eta_1, \dots$ )

---

$z^{(0)} \rightarrow \langle 0, 0, \dots, 0 \rangle$  %Inizializza la variabile da ottimizzare  
**for**  $k = 1$  **to**  $K$  **do**  
     $g^{(k)} \rightarrow \nabla_z \mathcal{F}|_{z^{(k-1)}}$  %Computa il gradiente nella posizione corrente  
     $z^{(k)} \rightarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$  %scendi il gradiente  
**return**  $z^{(k)}$

---

Nei problemi in cui il problema di ottimizzazione è non convesso si trovano dei minimi locali, questi non permettono all'algoritmo di proseguire in quanto non distingue tra minimi locali e minimi globali. Un altro punto è un punto a sella, in cui certe direzioni curvano verso l'alto e altre verso il basso. In tali punti il gradiente è 0 e l'algoritmo si blocca. Un modo per uscire da un punto a sella è spostarsi a lato un po' in modo da uscirne. Si nota come i punti a sella sono molti comuni in alte dimensioni. Il learning rate è molto importante in quanto permette di decidere la distanza coperta da uno spostamento determinando velocità di avvicinamento al minimo e precisione dell'algoritmo.

## Capitolo 9

# Regularization

### 9.1 Introduzione

Si noti come con il gradient descent si calcola il valore minimo della loss function sul training set. Ci si deve pertanto preoccupare del overfitting. Il minimo  $w$  e  $b$  sul training set infatti non sono tipicamente il minimo per il test set. Questo problema viene risolto attraverso la regolarizzazione.

### 9.2 Regolarizzatori

I regularizzatori sono criteri addizionali alla loss function in modo da evitare overfitting. Prova a mantenere i parametri regolari o normali. È un bias sul modello che forza che il learning preferisca certi tipi di pesi rispetto agli altri.

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

Tipicamente non si vogliono pesi molto grandi in quanto un piccolo cambio in una feature potrebbe causare un cambio nella predizione. Si potrebbe anche preferire pesi di 0 per features che non sono utili.

#### 9.2.1 Regolarizzatori comuni

##### 9.2.1.1 Somma dei pesi

Il regularizzatore somma dei pesi penalizza di più valori piccoli e si calcola come:

$$r(w, b) = \sum_{w_j} |w_j|$$

Si dice anche 1-norm.

##### 9.2.1.2 Somma quadratica dei pesi

La somma quadratica dei pesi penalizza di più valori grandi e si calcola come:

$$r(w, b) = \sqrt{\sum_{w_j} |w_j|^2}$$



Si dice anche 2-norm.

### 9.2.1.3 $P$ -norm

Si intende per  $p$ -norm:

$$r(w, b) = \sqrt[p]{\sum_{w_j} |w_j|^p} = ||w||^p$$

Valori più piccoli di  $p < 2$  incoraggiano vettori più sparsi, mentre valori più grandi scoraggiano pesi più grandi creando pertanto vettori con pesi più simili.

## 9.3 Gradient descent e regolarizzazione

Si nota come se scelto un modello e dimostrato che  $loss + regularizer$  è una funzione convessa si può ancora utilizzare il gradient descent. Si nota come per costruzione le  $p$ -norms sono convesse per  $p \geq 1$  e pertanto:

$$\begin{aligned} \frac{d}{dw_j} objective &= \frac{d}{dw_j} \sum_{i=1}^n \exp(-y_i(w \cdot x_i + b)) + \frac{\lambda}{2} ||w||^2 = \\ &\quad \cdot \\ &\quad \cdot \\ &\quad \cdot \\ &= - \sum_{i=1}^n y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) + \lambda w_j \end{aligned}$$

Pertanto con il regolarizzatore l'aggiornamento è

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \cdot x_i + b)) - \eta \lambda w_j$$

Si noti pertanto come la regolarizzazione, se  $w_i$  è positiva la riduce, mentre se è negativa lo aumenta muovendo  $w_i$  verso lo 0.

## 9.4 Regolarizzazione con le $p$ -norms

### 9.4.1 L1

$$w_j = w_j + \eta(lossCorrection - \lambda sign(w_j))$$

Popolare in quanto tende a dare soluzioni sparse, ma non è differenziabile e lavora unicamente per risolutori a gradient descent.

### 9.4.2 L2

$$w_j = w_j + \eta(lossCorrection - \lambda w_j)$$

Popolare in quanto per qualche loss function può essere risolta direttamente.

### 9.4.3 Lp

$$w_j = w_j + \eta(lossCorrection - \lambda c w_j^{p-1})$$

Meno popolare in quanto non riduce i pesi abbastanza.

## 9.5 Metodi di machine learning con regolarizzazione

- Ordinario: least squares: squared loss. regularization
- Ridge regression: squared loss with L2 regularization
- Elastic regression: squared loss with L2 and L1 regularization
- Lasso regression: squared loss with L1
- Logistic regression: logistic loss.

# Capitolo 10

## Support vector machines

### 10.1 Introduzione

Le support vector machines permettono di trovare l'iperpiano ottimo che separa i training data. Si nota come fino ad ora c'era una grande variabilità nell'iperpiano che il classificatore lineare trova cominciando da diversi punti nell'iperspazio. Inoltre quando i dati non sono linearmente separabili questa variabilità aumenta grandemente.

#### 10.1.1 Considerazioni su perceptron e gradient descent

Come visto il perceptron se i dati sono linearmente separabili trova un qualche iperpiano che li separa, altrimenti continuerà ad aggiustarsi iterando attraverso gli esempi e l'iperpiano dipenderà dall'ultimo esempio visto. Il gradient descent invece trova l'iperpiano che minimizza  $loss + regularization$  in entrambi i casi.

#### 10.1.2 Idea delle support vector machines

Le support vector machines cercano di trovare il migliore iperpiano che lo fa. Per definirlo vengono introdotti i margini di un iperpiano.

### 10.2 Margini

I margini di un iperpiano sono la distanza dal punto più vicino. Maggiore il margine meglio l'iperpiano separa le classi. Il fatto che il modello trovato da una SVM ha il maggior margine possibile aumenta l'abilità di generalizzazione del modello.

#### 10.2.1 Support vectors

I support vectors sono i data points più vicini al margine. Per  $n$  dimensioni ci saranno almeno  $n + 1$  support vectors.

### 10.2.2 Calcolare il margine

Il margine può essere pertanto calcolato come la distanza dal support vector. La distanza di un punto  $x$  da un'iperpiano viene calcolata come  $d(x) = \frac{wx+b}{\|w\|}$ . Il margine viene calcolato pertanto come  $\frac{c}{\|w\|}$ , dove  $c$  è la traslazione dell'iperpiano in modo che la sua distanza dal support vector sia 0. Si nota come scalando il vettore dei pesi  $w$  la distanza dall'iperpiano rimane la stessa. Inoltre essendo  $c$  e  $w$  strettamente correlati si può assumere  $c = 1$  sempre.

## 10.3 Problema di ottimizzazione

Si vuole pertanto massimizzare il margine ma classificando correttamente ogni esempio.

$$\operatorname{argmax}(\operatorname{margin}(w, b)) \quad \wedge \quad y_i(wx_i + b) \geq 1 \forall i$$

L'errore viene tenuto basso dal fatto di avere tutti i data point al di fuori dal margine come stabilito dalla seconda equazione.

### 10.3.1 Massimizzare il margine

Per massimizzare il margine si vuole massimizzare

$$\frac{1}{\|w\|}$$

Tenendo d'occhio l'errore. Si vuole pertanto avere il vettore di pesi minore possibile  $w$ .  $c = 1$  è un'assunzione che si fa in quanto altrimenti si starebbe imparando una versione scalata dello stesso problema in quanto per ogni support vector  $c = w \cdot x_i + b = 1$ . In realtà si tenta di minimizzare:

$$\|w\|^2 = \sqrt{\sum_{j \in I} |w_j|^2}$$

Soggetto a  $y_i(wx_i + b) \geq 1$ . Questo è lo stesso problema con il vantaggio che è una funzione convessa con lo stesso minimo della norma di  $w$ . In questo modo diventa un problema di ottimizzazione quadratica, di risoluzione semplice.

## 10.4 Soft margin classification

La soft margin classification permette di usare SVM quando i punti non sono linearmente separabili. Quando qualche dato non è linearmente separabile non si riescono a soddisfare i due constraint necessari per train le SVM e pertanto si necessita di modificare la funzione obiettivo. Per farlo si permette al modello di fare delle predizioni sbagliate, ma si aggiunge alla funzione obiettivo una penalità per ogni esempio classificato erroneamente. Queste sono dette slack penalties e sono usate per ottenere un modello che accetta con una certa tolleranza errori di classificazione. La funzione da ottimizzare diventa pertanto:

$$\|w\|^2 + C \sum_i \zeta_i \quad \text{subject to} \quad y_i(wx_i + b) \geq 1 - \zeta_i \forall i$$

Con il constraint  $\zeta_i \geq 0 \forall i$ . Si nota come  $\zeta_i$  sono le slack variables e se ne trova una per ogni esempio nel dataset e sono utilizzate per correggere classificazioni sbagliate, ma poi il loro peso è aggiunto alla

funzione obiettivo. Si vuole pertanto trovare un trade-off tra minimizzare il quadrato della norma di  $w$  e il valore di penalità dato dalla slack variable. La somma di  $\zeta_i$  misura la tolleranza agli errori. Si nota come:

- Piccoli valori di  $\mathcal{C}$  permettono più errori che consistono in un margine più grande.
- Grandi valori di  $\mathcal{C}$  danno agli errori di classificazione più peso restringendo il margine.
- $\mathcal{C} = \infty$  impone tutti i constraint e riduce a un hard margin.

I valori di  $\zeta_i$  sono imparati insieme a  $w$  e  $b$ . Si nota come questo è ancora un problema di ottimizzazione quadratica con constraint lineari, ma il numero di calcoli con un training set medio grande è molto elevato.

### 10.4.1 Risolvere il problema delle SVM

Data la soluzione ottimale  $(w, b)$  si può calcolare la slack penalty per ogni punto. Per tutti gli esempi correttamente classificati al di fuori del margine il valore dovrà essere 0. Per gli esempi correttamente classificati all'interno del margine sarà la distanza dal punto e la linea del margine, che può essere calcolata come  $1 - (\text{valore della funzione di decision})$ , un valore compreso tra 0 e 1, in particolare:

$$\zeta_i = 1 - y_i(wx_i + b)$$

I dati classificati con errore il valore è dato dalla somma della distanza dall'iperpiano più la distanza dal margine, con la stessa formula come nel caso precedente. Si riassume la formula in:

$$\zeta_i = \max(0, 1 - yy')$$

Che si nota che è la hinge loss function. Trasformando la funzione obiettivo di un SVM in una funzione di hinge loss si trasforma il problema in unconstrained in quanto entrambi i constraint sono tenuti in considerazione dalla funzione. La nuova funzione obiettivo pertanto sarà:

$$\min_{w,b} ||w||^2 + \mathcal{C} \sum_i \max(0, 1 - y_i * wx_i + b)$$

Che può essere considerata come un problema di gradient descent con Hinge loss function e regolarizzatore  $||w||^2$ . La soluzione a questo problema trova pertanto l'iperpiano con il margine più grande possibile permettendo un soft margin.

## 10.5 Data non lineramente separabile

Per separare i dati non linearmente separabili si possono utilizzare spazi con dimensioni maggiori. Prima si tentava di risolvere un problema di ottimizzazione quadratica soggetto a un'insieme di constraint lineari. Questo è il problema primario delle SVM. Si può riscrivere questo problema nella forma di un dual problem. Si noti come i problemi di ottimizzazione quadratica sono una classe ben conosciuta di problemi di programmazione per cui esistono diversi algoritmi non banali. Una possibile soluzione coinvolge costruire un dual problem dove un moltiplicatore di Lagrange  $a_i$  è associato con ogni constraint di ineguaglianza nel problema originario. Si deve pertanto trovare  $a_1, \dots, a_n$  tale che:

$$Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j x_i^T x_j$$

è massimizzato e  $\sum a_i y_i = 0$  e  $a_i \geq 0 \forall a_i$ . Il problema è pertanto massimizzare  $Q(a)$ , un problema di ottimizzazione quadratico con un paio di constraint lineari. Un buon risolutore scalabile è  $SM$ ). Una volta risolto il dual problem e ottenuto tutti i valori di  $a_i$  posso computare  $w$  e  $b$ :

$$w = \sum a_i y_i x_i$$

$$b = y_k - \sum a_i y_i x_i^T x_k$$

Per ogni  $a_k > 0$ . Pertanto la funzione classificatrice è:

$$f(x) = \sum (a_i y_i x_i^T x) + b$$

Si nota come tutte le  $a_i$  degli esempi che non sono support vector hanno valore 0, pertanto non si necessita di esplicitare  $w$  se si conosce  $a$ . In quanto  $x$  compare solo nel dot product nella funzione di predizione e di ottimizzazione e  $a$  è nulla per ogni esempio non support vector il loro impatto è nullo: una volta trainata la funzione di predizione può mantenere solo i support vector risparmiando memoria.

### 10.5.1 Soft margin classifier

Il dual problem è simile nel caso del soft margin classifier: si deve trovare  $a_i, \dots, a_n$  tali che:

- Massimizzano  $Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j x_i^T x_j$ .
- $\sum a_i y_i = 0$
- $0 \leq a_i \leq C \forall a_i$

Si nota come  $x_i$  con non zero  $a_i$  sono support vectors. Pertanto la predizione dipende solo dai support vectors. Con entrambi gli approcci si impara un iperpiano che separa i datapoints e in entrambi solo i data points rilevanti per la prediction function sono i support vector.

### 10.5.2 Identificare i support vector

I support vector sono identificati dagli algoritmi di ottimizzazione quadratica come i training point con non zero lagrangian multipliers  $a_i$  in quanto i training point appaiono unicamente all'interno prodotti interni, pertanto non li necessito ma solo i loro dot-product.

### 10.5.3 Utilizzo di SVN in maniera non lineare

Se i dati non sono linearmente separabili si possono portare a uno spazio a dimensioni maggiori rendendoli lineramente separabili. Lo spazio delle features originali può essere sempre mappato a uno spazio di features con dimensione maggiore dove il training set è separabile. Il problema di ottimizzazione e la funzione di predizione dipendono solo dal prodotto dei vettori di features degli esempi. Il classificatore lineare dipende sul prodotto interno di questi vettori:

$$K_{linear}(x_i, x_j) = x_i^T x_j$$

Se ogni data point è mappato in uno spazio con maggiore dimensione attraverso una qualche trasformazione  $\phi : x \rightarrow \phi(x)$ , il prodotto interno diventa:

$$K_{for \ function \ \phi}(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

**10.5.3.1 Kernel function**

Una kernel function è una funzione equivalente a un prodotto interno in uno spazio di features. In generale non si è interessati a  $\phi$  ma alla funzione kernel in quanto mappa i dati a uno spazio a maggiore dimensione implicitamente. Ogni kernel ha un iperparametro a parte la lineare.

- Linear:  $K(x_i, x_j) = x_i^T x_j$
- Polinomial of power  $p$ :  $K(x_i, x_j) = (1 + x_i^T x_j)^p$ .
- Gaussian o radial-basis:  $K(x_i, x_j) = e^{-\frac{\|x_i - x_j\|^2}{2\sigma^2}}$ . In questo caso ogni punto è mappato a una funzione gaussiana e  $\phi(x)$  ha infinite dimensioni. La combinazione delle funzioni per i support vectors è il separatore.

Lo spazio a maggiore dimensioni ha una dimensionalità  $d$  intrinseca ma il separatore lineare in esso corrisponde a un separatore non lineare nello spazio originale.

**10.5.3.1.1 Teorema di Mercer**

- Ogni funzione simmetrica semi positiva è un kernel.
- Le funzioni semi-positive simmetriche corrispondono a una matrice di Gram semi-positiva definita

**10.5.3.2 Soluzione del problema scelto il kernel**

Una volta scelto il kernel il problema diventa trovare  $a_1, \dots, a_n$  tali che:

- Massimizzano  $Q(a) = \sum a_i - \frac{1}{2} \sum \sum a_i a_j y_i y_j K(x_i, x_j)$ .
- $\sum a_i y_i = 0$
- $a_i \geq 0 \forall a_i$

La soluzione è:

$$f(x) = \sum a_i y_i K(x_i, x) + b$$

La tecnica di ottimizzazione per  $a_i$  rimane la stessa. Si nota come risolvendo il dual problem con il kernel giusto si risolve il problema SVM portando i dati in uno spazio a dimensione maggiore in cui è linearmente separabile.

# Capitolo 11

## Neural Networks

### 11.1 Introduzione

È stato visto come il perceptron può imparare un modello lineare utilizzando la funzione attivatrice e il peso degli input. Funzioni attivatrici tradizionali sono non lineari come la sigmoide  $h(x) = \frac{1}{1+e^{-x}}$  e la tangente iperbolica  $h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ , mentre funzioni attivatrici più moderne sono la rectified linear unit *ReLU*:  $h(x) = \max(0, x)$  e la Leaky ReLU  $h(x) = \max(\alpha x, x)$  con  $\alpha$  costante piccola.

#### 11.1.1 Perceptron

Il perceptron si può considerare come un neurone artificiale, una funzione non lineare parametrizzata con un range di output ristretto:

$$\hat{y} = h\left(\sum_i \theta_i x_i\right) = h(\theta^T x)$$

##### 11.1.1.1 Roseblatt

Nel 1958 Roseblatt considera il perceptron come una macchina per la classificazione lineare: si imparano i pesi e si considera il bias: un peso per input, si moltiplicano i pesi con gli input rispettivi e si aggiunge il bias. Se il risultato è più grande di una threshold si ritorna 1, altrimenti 0.

#### 11.1.2 Multi layer perceptron

Il perceptron presenta diverse limitazioni: non è in grado di risolvere problemi non linearmente separabili come quello dello *XOR*. Per superare questo problema Minsky e Papert nel 1969 sviluppano il multi-layer perceptron *MLP*. Questi sono neuroni artificiali densamente connessi che realizzano composizioni di funzioni non lineari utilizzati per la classificazione e regressioni. Sono composti da un input layer, diversi hidden layer e un output layer. L'informazione viene propagata dall'input all'output senza cicli: l'*MLP* è un directed acyclic graph *DAG*. La computazione viene svolta dalla composizione di un numero di funzioni algebriche implementate dalle connessioni, pesi e biases dei layer di output e nascosti. Gli hidden layer computano delle rappresentazioni intermedie.



### 11.1.2.1 Single layer neural network

In una single layer neural network il primo layer (1) riceve i dati dall'input e li passa a un layer nascosto (2), che a sua volta li passa all'output:  $\hat{y}_k$ :

$$(1) \quad z_j = \sum_i \theta_{i,j}^{(1)} x_i + \theta_{0,j}^{(1)}.$$

$$(2) \quad \hat{y}_k = h\left(\sum_i \theta_{i,j}^{(2)} h(z_i) + \theta_{0,j}^{(2)}\right).$$

### 11.1.3 First AI winter

Il first AI winter nasce in quanto si rende difficile fare del training su un *MLP*: non ci si può applicare la regola del perceptron in quanto si aspetta di conoscere il target desiderato. Per gli hidden layer infatti è impossibile conoscere il target desiderato.

#### 11.1.3.1 Backpropagation

Il problema del training del *MLP* viene risolto attraverso la backpropagation nel 1986, che permette pertanto il learning di *MLP* per funzioni complicate. Algoritmi efficienti permettono di processare grandi training sets e permettono architetture complesse di neural networks. Tutt'oggi si trova al nucleo del training delle reti neurali. Il training pertanto consiste di tre passaggi:

- Forward propagation: si sommano gli input, si produce attivazione e si fa feed-forward.
- Si stima l'errore.
- Si propaga all'indietro il segnale di errore e lo si usa per aggiornare i pesi.

Il learning diventa un problema di ottimizzazione, in cui dati i training samples  $T = \{(x_1, y_1), \dots, (x_N, y_N)\}$  si devono aggiustare tutti i pesi della rete  $\Theta$  in modo che una funzione di costo sia minimizzata:

$$\min_{\Theta} \sum_i L(y_i, f(x_i, \Theta))$$

Si deve pertanto scegliere la loss function, aggiornare i pesi di ogni layer con il gradient descent e usare la backpropagation del segnale di errore per computare il gradiente efficientemente.

#### 11.1.3.2 CNN e LSTM

La backpropagation permette importanti sviluppi nel campo come le convolutional neural networks e le recurrent long-short term memory networks negli anni 90.

### 11.1.4 Second AI winter

Si nota come queste reti neurali non possono sfruttare molti layer a causa di overfitting e del vanishing gradient: la moltiplicazione di piccoli numeri nel training causa a questi di diventare sempre più piccoli fino a renderli non significativi. Inoltre non si disponeva della capacità computazionale necessaria e mancavano grandi dataset annotati.

#### 11.1.4.1 SVM e metodi di kernel

A causa di questi problemi diventano popolari le Kernel machines come gli SVM in quanto riuscivano ad ottenere accuratèzze simili alle reti neurali, possedevano meno euristiche e parametri e una prova di generalizzazione.

#### 11.1.5 Deep learning revolution

Nel 2006 si utilizza un nuovo modo per inizializzare i pesi: ogni layer viene trainato singolarmente attraverso unsupervised training come contrastive divergence e i pesi si sistemano con un round di supervised learning. Nasce così alexnet, una rete CNN simile a LeNet ma trainata con due GPU e con migliori tecniche come ReLU, dropout e data augmentation.

#### 11.1.6 Caratteristiche del deep learning

Si nota come il deep learning, a differenza del machine learning tradizionale non richiede la creazione di feature a tavolino: la loro estrazione avviene infatti negli hidden layers. Si nota pertanto come una rete neurale è una composizione di moduli, funzioni connesse gerarchicamente con parametri  $\Theta$ .

### 11.2 Feedforward networks

Nelle reti feed forward la funzione  $f$  è la composizione di multiple funzioni:

$$f(x) = f^{(n)}(\dots(f^{(1)}(x)\dots))$$

L'obiettivo è quello di approssimare una funzione sconosciuta ideale  $f^* : \mathcal{X} \rightarrow \mathcal{Y}$ , in cui il modello ideale è  $y = f^*(x)$ . Per queste reti si definisce una mappatura parametrica  $y = f(x, \theta)$  e si imparano i parametri per trovare una buona approssimazione di  $f^*$  dai sample disponibili. L'informazione fluisce dall'input, verso computazioni intermedie e si produce l'output. La composizione delle funzioni può essere descritta come un *DAG*, in cui la profondità è il massimo  $i$  nella catena di composizione delle funzioni. Il layer finale viene chiamato layer di output.

#### 11.2.1 Training

Per il training si deve ottimizzare  $\theta$  per portare  $f(x, \theta)$  il più vicino possibile a  $f^*(x)$ . Questa viene valutata a diverse istanze di  $x$  di training data, che specifica solamente l'output dei layer finali. L'output dei layer intermedi non viene specificato, da cui il nome hidden layers. Questo è simile a progettare una macchina basata su gradient descent: il problema diventa non convesso e non c'è garanzia di convergenza. In particolare per applicare il gradient descent si deve specificare un modello, una funzione di costo e la rappresentazione dell'output.

#### 11.2.2 Scelte di modellamento

Per modellare una feedforward network si devono scegliere:

- Funzione di costo.
- Architettura.
- Forma dell'output.
- Ottimizzatore.
- Funzione di attivazione.

### 11.2.2.1 Funzione di costo

La funzione di costo dice quanto la rete si comporta bene rispetto ai training data:

$$\mathcal{L}(w) = \text{distance}(f_{\theta}(x), y)$$

Calcola ovvero la discrepanza tra la predizione e la label effettiva, Si possono applicare loss già viste e tipicamente si converte l'output in probabilità come attraverso softmax.

**11.2.2.1.1 Cross-entropy** La cross-entropy loss è la loss function più comune applicata con softmax:

$$\mathcal{L}_i = - \sum_k y_k \log(S(l_k)) = \log(S(l))$$

### 11.2.2.2 Output layers

Si nota come la scelta della loss function sia correlata alla scelta dell'unità di output.

**11.2.2.2.1 Linear** Data una feature  $h$  un unità di un layer di output lineari dà:

$$\hat{y} = W^T h + b$$

Questi unità non si saturano (mantiene il gradiente lontano da 0), offrono poca difficoltà agli algoritmi basati sull'ottimizzazione del gradiente. Si nota come un output del modello vicino a 0 è problematico.

**11.2.2.2.2 Softmax** Softmax permette di produrre probabilità normalizzate nell'output layer, pertanto produce:

$$S(l_i) = \frac{e^{l_i}}{\sum_k e^{l_k}}$$

### 11.2.2.3 Hidden units

All'interno di una hidden unit si accetta un input  $x$ , si computa una trasformazione affine:  $z = w^T x + b$ , si applica element-wise una funzione non lineare  $h(z)$  e si ottiene l'output  $h(z)$ . La scelta da compiere si basa su quale  $h$  utilizzare.

**11.2.2.3.1 Rectified linear units (RELU)** Una RELU presenta un gradiente di 0 o 1, facile da ottimizzare simile alle unità linari. Dà gradienti grandi e consistenti quando attiva, ma non è sempre differenziabile, risolvibile attraverso una derivata da un lato a  $z = 0$ . Ne esistono diverse varianti che risolvono il problema che l'unità muore quando il gradiente è 0:

- ReLu:  $y_i = 0$  per  $x < 0$  o  $y_i = x_i$  per  $x > 0$ .
- Randomized Leaky Relu:  $y_{ji} = a_{ji} x_{ji}$  per  $x < 0$  o  $y_{ji} = x_{ji}$  per  $x > 0$ .
- Leaky Relu:  $y_i = a_i x_i$  per  $x < 0$  o  $y_i = x_i$  per  $x > 0$ .

**11.2.2.3.2 Sigmoid e Tanh** Questi due tipi di hidden unit riducono il tipo di non linearità restringendo l'output ai range  $[0, 1]$  e  $[-1, 1]$ . Si saturano lungo la maggior parte del dominio e sono fortemente sensibili unicamente quando l'input è più vicino a 0. La saturazione rende l'apprendimento basato sul gradient più difficile.

- Sigmoidale:  $h(x) = \frac{1}{1+e^{-x}}$ .
- Tangente iperbolica:  $h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ .

#### 11.2.2.4 Architettura

La decisione rispetto alla profondità e larghezza di una rete neurale si basa principalmente su risultati empirici. Un risultato teorico da parte di Cybenko 1989: reti a 2-layer con output lineari con del squashing non-linearity nelle hidden-unit possono approssimare ogni funzione continua su dominio compatto ad accuratezza arbitraria. Questo risultato è valido anche per funzioni non lineari. Questo implica che per ogni funzione che si cerca di imparare, un grande *MLP* è in grado di impararla. Nonostante questo non è garantito che l'algoritmo di training sia capace di imparare tale funzione.

### 11.2.3 Backpropagation

La backpropagation è il modo in cui la rete impara i propri pesi. Consiste di tre passaggi:

1. Feedforward propagation: si accetta l'input  $x$ , si passa attraverso stages intermedi e si ottiene l'output.
2. Si usa l'output computato per computare un costo scalare dipendente dalla loss
3. La backpropagation permette all'informazione di fluire all'indietro dal costo per computare il gradiente.

Si utilizza il gradient descent: si necessitano le derivate degli errori per ogni peso nella rete:

$$w_{jk}^{(i)} := w_{jk}^{(i)} - \nu \frac{\partial L}{\partial w_{jk}^{(i)}}$$

Dai training data non si conosce cosa dovrebbero fare gli hidden layers, ma si può computare quanto velocemente l'errore cambia cambiando la loro attività: si usano le derivate dell'errore con rispetto alle sue attività. Ogni hidden unit può avere effetto su diverse unità di output e effetti separati sull'errore che vengono combinati. Si può calcolare la derivata di queste unità efficientemente: una volta che si ha la derivata delle attività nascoste è facile ottenere l'errore per i pesi che arrivano.

#### 11.2.3.1 Operazione di feedforward

Dall'input all'output:

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

#### 11.2.3.2 Computare l'errore e train

L'errore della rete sul training set:

$$L(X; w) = \sum_{i=1}^N \frac{1}{2} (y_i - \hat{y}(x_i; w))^2$$

Non ha una soluzione di forma chiusa, si utilizza il gradient descent. Si deve calcolare la derivata di  $L$  su un singolo esempio. Si può considerare un modello lineare semplice con output  $\hat{y} = \sum_j w_j x_{ij}$ :

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

**11.2.3.3 Backpropagation**

L'unità di attivazione generale in una rete multilayer:

$$z_t = h\left(\sum_j w_{jt} z_j\right)$$

La forward propagation calcola per ogni unità  $a_t = \sum_j w_{jt} z_j$ . Dove  $a_t$  è l'input della funzione di attivazione di un'unità di livello  $t$ . Ora

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$$

Sia  $\partial a_t = \delta_t$  e l'unità di output con attivazione lineare  $\delta_t = \hat{y} - t$ . L'unità hidden  $t$  che manda l'output all'unità  $S$ :

$$\begin{aligned} \delta_t &= \sum_{s \in S} = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t} = \\ &= h'(a_t) \sum_{s \in S} w_{ts} \delta_s \end{aligned}$$

**11.2.3.4 Esempio**

Sia l'output  $f(a) = a$  e l'hidden:  $h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$ , ora  $h'(a) = 1 - h(a)^2$ . Dato  $x$ , il feedforward inputs:

- Input to hidden  $a_j = \sum_{i=0}^d w_{ij}^{(1)} x_i$ .
- Net output  $\hat{y} = a = \sum_{i=0}^m w_j^{(2)} z_j$ .
- Hidden input  $z_j = \tanh(a_j)$ .

L'errore sull'esempio  $x$  è  $L = \frac{1}{2}(y - \hat{y})^2$  e l'unità di output:  $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$ . Si computa  $\delta$  per le unità hidden:

$$\delta_j = (1 - z_j^2) w_j^{(2)} \delta$$

Le derivate con rispetto ai pesi:

- $\frac{\partial L}{\partial w_{ij}^{(1)}} = \delta_j x_i$ .
- $\frac{\partial L}{\partial w_j^{(2)}} = \delta z_j$ .

E si aggiornano i pesi:

- $w_j = w_j - \nu \delta z_j$ .
- $w_{ij}^{(1)} = w_{ij}^{(1)} - \nu \delta_j x_i$ .

**11.2.3.4.1 Output multidimensionale** Per l'output multidimensionale la loss sull'esempio è:

$$\frac{1}{2} \sum_{k=1}^K (y_k - \hat{y}_k)^2$$

Per ogni unità di output:  $\delta_k = y_k - \hat{y}_k$ . Per l'unità hidden  $j$ :

$$\delta_j = (1 - z_j^2) \sum_{k=1}^L w_{jk}^{(2)} \delta_k$$

### 11.2.4 Scelta di un ottimizzatore

Il gradient è il vettore delle derivate parziali con rispetto di tutte le coordinate dei pesi:

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

Ogni derivata parziale misura quanto velocemente la loss cambia in una direzione. Quando il gradiente è zero, la loss non cambia in nessuna direzione. Dà problemi nei saddle points e nei local minima. Il gradient descent trova pertanto l'insieme di parametri che rendono la loss il più piccola possibile. I cambi nei parametri dipendono dal gradiente della loss con rispetto dei pesi della rete. La backpropagation è il metodo per computare i gradienti. Si analizzano altri miglioramenti come Stochastic gradient descent. Il gradient descent nelle reti neurali può essere calcolato in diversi modi.

#### 11.2.4.1 Batch Gradient Descent (BGD)

In BGD i gradienti sono computati su ogni update per l'intero training con un alto costo computazionale ma garantendo una grande stabilità nella stima del gradiente. Il learning rate  $\varepsilon_k$  può cambiare nel tempo.

---

```
: Batch Gradient Descent at iteration K
return: Learning rate  $\varepsilon_k$ 
return: Initial Parameter  $\theta$ 
while stopping criteria not met do
    %Compute gradient estimate over  $N$  examples
     $\hat{g} = +\frac{1}{N} \nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ 
    %Apply update
     $\theta = \theta - \varepsilon_k \hat{g}$ 
```

---

#### 11.2.4.2 Stochastic gradient descent (SDG)

In SDG si computa il gradiente solo su un campione e non sull'intero training set in modo da ottenere performance migliori. Il learning rate cambia ad ogni passo, tipicamente decade linearmente.

---

```
: Stochastic Gradient Descent at iteration K
return: Learning rate  $\varepsilon_k$ 
return: Initial Parameter  $\theta$ 
while stopping criteria not met do
    %Compute gradient estimate over sample example  $(x^{(i)}, y^{(i)})$  from training
    set
     $\hat{g} = +\nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ 
    %Apply update
     $\theta = \theta - \varepsilon_k \hat{g}$ 
```

---

#### 11.2.4.3 Minibatches

Le minibatches risolvono il problema di SDG rispetto ai dati rumorosi. Il tempo di computazione per aggiornamento non dipende dal numero di esempi di training  $N$ , permettendo la creazione di minibatches più grandi che vengono computate parallelamente. Sono tipicamente di dimensione  $2^n$  per le proprietà di calcolo della GPU.

#### 11.2.4.4 Momento

Un problema di BGD e SGD è il fatto che minimizzano l'errore in molto tempo. Una soluzione diversa è data dalla tecnica del momento, che introduce un vettore velocità  $v$  di aggiornamenti, una media con decay esponenziale per i gradienti utilizzato per aggiornare i pesi. Introduce un vettore momento che regola il trade-off tra il gradiente allo step corrente e quelle vecchie.

---

```
: Stochastic Gradient Descent with momentum
```

---

```

return: Learning rate  $\varepsilon_k$ 
return: Momentum parameter  $\alpha$ 
return: Initial Parameter  $\theta$ 
return: Initial velocity  $v$ 
while stopping criteria not met do
    %Compute gradient estimate over sample example  $(x^{(i)}, y^{(i)})$  from training
    set
     $\hat{g} = +\nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ 
    %Compute velocity update
     $v = \alpha v - \varepsilon_k \hat{g}$ 
    %Apply update
     $\theta = v$ 
```

---

#### 11.2.4.5 Adaptive learning rate method

Alcune volte è bene usare un learning rate diverso per ogni peso. Un metodo che lo implementa è Adagrad o adaptive gradient optimizer. Questo fa downscale di un parametro del modello della radice della somma dei quadrati dei valori storici, in questo modo parametri con una grande derivata parziale hanno learning rate che si abbassano rapidamente. Si adatta al learning rate dei parametri svolgendo aggiornamenti minori associati con feature che più frequenti e grandi per feature meno frequenti. Utile per gestire dati sparsi.

---

```
: AdaGrad
```

---

```

return: Learning rate  $\varepsilon_k$ 
return: Initial Parameter  $\theta, \delta$ 
 $r = 0$ 
while stopping criteria not met do
    %Compute gradient estimate over sample example  $(x^{(i)}, y^{(i)})$  from training
    set
     $\hat{g} = +\nabla_{\theta} \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ 
    %Compute velocity update
     $r = r + \hat{g}$  %Compute update
     $\Delta\theta = -\frac{\varepsilon_k}{\delta + \sqrt{r}} \circ i\hat{g}$ 
     $\theta = \theta + \Delta\theta$ 
```

---

### 11.3 Convolutional Neural Networks (CNN)

Le convolutional neural networks sono un tipo di feedforward neural networks. Queste gestiscono molto bene uno spazio di input localmente strutturato, spazialmente o temporalmente. Sono inoltre molto utili quando l'obiettivo non è la classificazione: ottengono grandi risultati in region extraction,

feature detection, semantic segmentation e structured regression. In particolare le CNN imparano una gerarchia di features: ognuno dei layer estrae delle features dall'output del layer precedente. Tutti i layer vengono trainati insieme. Si definiscono CNN le reti che usano la convolution al posto di una moltiplicazione tra matrici generica in almeno uno dei loro livelli. Si definisce convolution:

$$S(i, j) = (I \cdot K)(i, j) = \sum_M \sum_N I(m, n) K(i - m, j - n)$$

### 11.3.1 Origini

Questo tipo di reti è stato ispirato dalla corteccia visiva dei mammiferi: questa contiene un ordinamento di cellule sensibili a piccole sotto regioni del campo visivo, dette campo recettivo. Queste cellule si comportano come filtri locali sullo spazio di input e sfruttano la correlazione spaziale presente in immagini naturali. Esistono due tipi di cellule:

- Cellule semplici: rispondono al massimo a pattern simili a lati nel loro campo recettivo.
- Cellule complesse: hanno campi recettivi più ampi e sono invarianti localmente rispetto alla posizione esatta del pattern.

In conclusione la convolution è un'operazione di filtraggio generale per le immagini. Si applica una matrice kernel a un'immagine e lavora determinando il valore di un pixel centrale aggiungendo i valori pesati dei suoi vicini. L'output è pertanto una nuova immagine modificata. Può essere utilizzato per lisciare, rendere più nitida un'immagine. Si nota come quest'operazione è commutativa.

### 11.3.2 Architettura

Le CNN sono pertanto delle feedforward neural networks con una struttura di connettività specializzata. Impilano diversi layer di feature extractors:

- Low-level layers estraggono feature locali.
- High-level layer imparano pattern globali.

Tipicamente i layer di CNN trasformano la matrice di input in una predizione della classe di output. Ci sono tre tipi di operazioni distinte:

1. Convolution.
2. Non-linearity.
3. Pooling.

#### 11.3.2.1 Convolutional layers

I convolutional layers sono il nucleo di una CNN, consistono di un insieme di filtri, ognuno dei quali copre una piccola porzione spaziale dei dati di input o campo recettivo. Ogni filtro viene convoluto attraverso la dimensione dei dati di input, producendo una mappa di feature multidimensionale. La rete impara i filtri che si attivano quando trovano uno specifico tipo di feature a una certa posizione spaziale nell'input.

#### 11.3.2.2 Non linearity

La non linearity aumenta la non linearity dell'intera architettura senza avere effetto sui campi recettivi del livello di convolution.

$$y_{i,j} = f(a_{i,l})$$

Per esempio  $f(a) = [a]_+$  o  $f(a) = \text{sigmoid}(a)$ .



### 11.3.2.3 Pooling

Il pooling riduce la dimensione spaziale della rappresentazione riducendo la quantità di parametri e la computazione della rete, controllando l'overfitting.

$$x_{i,j} = \max_{|k| < \tau, |l| < \tau} y_i - k_{ij} - l$$

Inoltre fornisce invarianza rispetto alle traslazioni.

### 11.3.2.4 Esempi di architetture

- LeNet.
- AlexNet.
- VGG.
- GoogleNet.
- ResNet.

### 11.3.3 Conclusione

Le CNN a differenza delle feedforward vanilla presentano un ordinamento spaziale: ad ogni layer le unità sono organizzate in griglie a  $2D$ , le feature maps. Ognuna di esse è il risultato di una convoluzione. Lo stesso filtro convoluzionale è applicato ad ogni locazione. I pesi sono diversi attraverso le feature maps. Una unità a una particolare locazione sulla griglia può solo ricevere l'input da unità a una locazione simile al livello inferiore. Si necessita di dati con labels ed è flessibile a molte applicazioni.

## 11.4 Altre reti neurali

Esistono diversi modelli di reti neurali per diverse necessità. In particolare fino ad ora si è discusso su problemi di predizione in cui si hanno input e output di dimensione fissa. Ci sono casi di sequential prediction tasks in cui l'input o l'output sono sequenze di lunghezza variabile. Si usano per esempio nella predizione dei frame video. Questi si risolvono attraverso recurrent neural networks.

### 11.4.1 Recurrent neural networks

Nelle recurrent networks input precedenti vengono utilizzati per influenzare le decisioni su input successivi. Questo si ottiene attraverso dei layer ricorsivi, in cui la rappresentazione nascosta al tempo  $t$ :

$$h_t = f_w(x_t, h_{t-1})$$

Si nota come la funzione di  $W$  dipende sia dall'input  $t$  che dallo stato precedente. In particolare si nota come considerando una visione "svolta" dei layer ricorsivi si possono mappare diversi input a diversi output in relazione:

- One-to-one.
- One-to-many
- Many-to-one.
- Many-to-many.

### 11.4.2 Autoencoders

Gli autoencoders sono un esempio di unsupervised learning. Sono simili a feedforward neural networks. Il loro scopo è quello di comprimere automaticamente dell'informazione. Hanno una forma a clessidra in cui i layer in mezzo sono quelli più piccoli (chokepoint della rete). Tutto quello che si trova fino alla metà è l'encoder, quello dopo il decoder. La rete si addestra attraverso backpropagation e l'errore è la differenza tra l'output e l'input. Possono essere costruiti simmetricamente anche rispetto ai pesi. Lo scopo è pertanto imparare una rappresentazione a meno dimensioni del training data. L'encoder è passato da essere un linear + nonlinearity come sigmoid, a un deep, fully connected a ReLu con CNN.  $z$  nel mezzo è più piccolo di  $x$ : si svolge una dimensionality reduction. Le feature dovrebbero catturare fattori significativi di variazioni nel dato. Un esempio per capire la differenza tra output e input è la L2 loss function  $\|x - \hat{x}\|^2$ . Dopo il training si scarta il decoder e l'encoder può essere utilizzato per inizializzare un modello supervised che permette un fine-tuning insieme al classificatore.

## Capitolo 12

# Unsupervised Learning

### 12.1 Introduzione

Nel unsupervised learning non si hanno label per guidare il learning algorithm, pertanto il meccanismo di learning sarà molto diverso. Si osservano i dati da una distribuzione sconosciuta  $p_{data} \in \Delta(X)$ . In quanto non si hanno osservazioni riguardo i target si deve introdurre un modello di conoscenza a priori o dare supervisione implicita attraverso il design della funzione obiettivo.

#### 12.1.1 Task tipiche

Le task tipiche dell'unsupervised learning sono:

- Dimensionality reduction: incorporare i dati in uno spazio a meno dimensioni.
- Clustering: dividere i dati in gruppi con proprietà simili.
- Density estimation: imparare una distribuzione di probabilità che fitta meglio il training data.

### 12.2 Dimensionality reduction

#### 12.2.1 Task

Si deve trovare una funzione  $f \in Y^X$  che mappa ogni input a grandi dimensioni  $x \in X$  a dimensioni minori incorporando  $f(x) \in Y$ , dove  $\dim(Y) \ll \dim(X)$ . Questa task permette di preservare le informazioni riducendo la dimensione di ogni dato in input. Questo approccio permette di visualizzare i dati, semplificare la loro computazione.

#### 12.2.2 Principal component analysis

In questo meccanismo di dimensionality reduction la conoscenza implicita messa nell'algoritmo è che la varianza è un indicatore di ricchezza di informazioni dati da una dimensione. PCA trova una trasformazione ottimale nel sistema di coordinate tale che perdere delle dimensioni nelle nuove coordinate porta la più piccola riduzione nella varianza dei dati. PCA in particolare perde l'asse con la varianza minore perdendo meno informazioni possibile.

**12.2.2.1 Varianza lungo una dimensione  $w$** 

Per calcolare la varianza lungo una data unità di direzione  $w$ .  $w$  è un'unità di direzione se  $w^T w = 1$ . Sia  $x_i$  un data point, si necessita di un punto nello spazio  $c$  da cui si applica la direzione  $w$  per calcolare la varianza di tutti i data points:

$$t_i = (x_i - c)^T w$$

La proiezione di  $x_i$  su  $w$ . Il valore atteso  $\mathbb{E}[t]$ :

$$\begin{aligned}\mathbb{E}[t] &= \frac{1}{n} \sum_{i=1}^n t_i = \frac{1}{n} \sum_{i=1}^n (x_i - c)^T w = \\ &= \bar{x}^T w - c^T w\end{aligned}$$

Ora la varianza sulla dimensione definita da  $w$  è:

$$\begin{aligned}\text{Var}[t] &= \frac{1}{n} \sum_{i=1}^n (t_i - \mathbb{E}[t])^2 = \frac{1}{n} \sum_{i=1}^n [(x_i - c)^T w - (\bar{x}^T w - c^T w)]^2 = \\ &= \frac{1}{n} \sum_{i=1}^n (\bar{x}_i^T w)^2 = w^T \left[ \frac{1}{n} \bar{X} \bar{X}^T \right] w = \\ &= w^T C w\end{aligned}$$

Dove  $C$  è la matrice della covarianza e  $\bar{X} = [\bar{x}_1, \dots, \bar{x}_n]$ . Si nota come la computazione della varianza non dipende dal punto  $c$  in quanto è implicitamente calcolata da un punto di vista centrato.

**12.2.2.2 Eigenvalue decomposition**

Sia  $A \in R^{m \times m}$  quadrata e simmetrica. Allora esiste  $U = [u_1, \dots, u_n] \in R^{m \times m}$  e  $\lambda = (\lambda_1, \dots, \lambda_m)^T \in R^m$  tali che:

$$A = U \Lambda U^T = \sum_{j=1}^m \lambda_j u_j u_j^T$$

E  $U^T U = U U^T = I$  e  $U$  è ortonormale. Ogni colonna di  $U$  ha lunghezza di unità e ogni paio di colonne diverse sono ortogonali tra di loro. La matrice  $\Lambda$  è creata con zero da tutte le parti e  $\lambda$  sulla diagonale.  $u_j$  è un eigenvector e  $\lambda_j$  è la eigenvalue corrispondente. Si assume un ordinamento discendente:  $\lambda_1 \geq \lambda_m$ .

**12.2.2.3 First principal component**

Il first principal component è la direzione dove la maggior parte della varianza è conservata. Si deve risolvere il problema di massimizzazione della varianza:

$$w_1 \in \arg \max \{w^T C w : w^T w = 1\}$$

Si devono fare constraints su  $w_1$  altrimenti crescerebbero all'infinito. La varianza di PCA è il valore del più grande eigenvalue della matrice  $C$  di covarianza, la prima componente principale è il corrispondente vettore  $w_1$ . Il maggiore eigenvalue di  $C$  è la varianza lungo il first principal component e il first principal component  $w_1$  è il corrispondente eigenvector.

**12.2.2.3.1 Dimostrazione** Dalla decomposizione di eigenvalue  $C = \sum_j \lambda_j u_j u_j^T$  si assumono le eigenvalue ordinate in ordine discendente. Allora  $W_1^T C w_1 = \sum_j \lambda_j (w_1^T u_j)^2 \leq \lambda_1$  in quanto:

$$\sum_j (w_1^T u_j)^2 = w_1^T \sum_j u_j u_j^T w_1 = w_1^T U U^T w_1 = w_1^T w_1 = 1$$

Segue che  $\lambda_1 \geq w_1^T C w_1 > u_1^T C u_1 = \lambda_1$ , da cui  $w_1^T C w_1 = u_1^T C u_1$ . Pertanto  $u_1$ , l'eigenvector corrispondente al eigenvalue maggiore  $\lambda_1$  di  $C$  è il first principal component e  $\lambda_1$  la varianza lungo di esso.

#### 12.2.2.4 Second principal component

Il second principal component deve essere ortogonale al primo:

$$w_2 \in \arg\{w^T C w : w^T w = 1, w \perp w_1\}$$

Il secondo eigenvalue maggiore di  $C$  è la varianza lungo il second principal component e  $w_2$  è il corrispondente eigenvector.

**12.2.2.4.1 Dimostrazione** Dalla decomposizione di eigenvalue  $C = \sum_j \lambda_j u_j u_j^T$  si assumono le eigenvalue ordinate in ordine discendente. Allora

$$W_2^T C w_2 = \sum_{j=1}^m \lambda_j (w_2^T u_j)^2 = \lambda_1 w_2^T u_1 + \sum_{j=2}^m \lambda_j (w_2^T u_j)^2 \leq \lambda_2$$

In quanto:

$$\sum_j (w_2^T u_j)^2 = w_2^T \sum_j u_j u_j^T w_2 = w_2^T U U^T w_2 = w_2^T w_2 = 1$$

Segue che  $\lambda_2 \geq w_2^T C w_2 > u_2^T C u_2 = \lambda_2$ , da cui  $w_2^T C w_2 = u_2^T C u_2$ . Pertanto  $u_2$ , l'eigenvector corrispondente al secondo eigenvalue maggiore  $\lambda_2$  di  $C$  è il second principal component e  $\lambda_2$  la varianza lungo di esso.

#### 12.2.2.5 I-esimo principal component

Allo stesso modo:

$$w_i \in \arg\{w^T C w : w^T w = 1, w \perp w_j, 1 \leq j < i\}$$

L'i-esimo eigenvalue maggiore di  $C$  è la varianza lungo l'i-esimo principal component. L'i-esimo principal component è  $w_i$  è il corrispondente eigenvector. La dimostrazione è analoga a quella del second principal component.

#### 12.2.2.6 PCA utilizzando eigenvalue decomposition

- Siano i data points  $X = [x_1, \dots, x_n]$ .
- Si centri  $\bar{X} = X - \frac{1}{n} X 1_n 1_n^T$ .
- Si computi la matrice di covarianza  $C = \frac{1}{n} \bar{X} \bar{X}^T$ .
- Eigenvalue decomposition:  $U, \lambda = \text{eig}(C)$ .
- Principal components:  $W = U = [u_1, \dots, u_m]$ , varianze  $\lambda = (\lambda_1, \dots, \lambda_m)$ .

**12.2.2.7 PCA utilizzando singular value decomposition**

Sia  $A \in \mathbb{R}^{m \times n}$ . Allora esiste  $U \in \mathbb{R}^{m \times k}$ ,  $s \in \mathbb{R}^k$  con  $s_1 \geq \dots \geq s_K > 0$  e  $V \in \mathbb{R}^{n \times k}$  tali che:

$$A = USV^T \quad \wedge \quad U^T U = V^T V = I$$

Si computa pertanto la SVD di  $\bar{X} : U, s, V = SVD(\bar{X})$ . Le componenti principali  $U = [u_1, \dots, u_k]$  e le varianze  $(\frac{s_1^2}{n}, \dots, \frac{s_k^2}{n})$ . La matrice  $U$  è la matrice con gli eigenvectors. Le eigenvalue possono essere calcolate in quanto:  $\bar{x} = USV^T$  e  $c = \frac{1}{n} \bar{x} \bar{x}^T = \frac{1}{n} USV^T V S U^T = U \frac{s^2}{n} U^T$

**12.2.2.8 Dimensionality reduction**

Sia  $\hat{W} = [w_1, \dots, w_k]$  le prime  $k$  componenti principali derivate dai data points  $\bar{X} = [\bar{x}_1, \dots, \bar{x}_n]$ . Si cambia a un sistema di coordinate ridotto con le  $k$  componenti principali con  $\hat{W}$  come assi:

$$T = \hat{W}^T \bar{X} \in \mathbb{R}^{k \times n}$$

Dove  $T$  è il principal component scores. Si può usare la eigenvalue decomposition o SVD per computare la decomposizione completa. Con la PCA si ottengono gli eigenvectors o componenti, si possono ordinare e usarli per comporre la matrice  $\hat{W}$ . Una volta fatto quello si usa per calcolare  $T$  che è il dataset dimensionalmente ridotto.

**12.2.2.9 Interpretazioni alternative**

Si può considerare la first principal component come la linea nello spazio con la minore distanza quadrata dai data points. La stessa interpretazione può essere data alle altre componenti principali.

**12.2.2.10 Scalare delle variabili**

PCA è sensibile alla scala delle features, pertanto è raccomandato scalarle secondo la standard deviation.

**12.2.2.11 Componenti principali da considerare**

Il numero di componenti per la dimensionality reduction dipende dall'obiettivo e dall'applicazione. Non ci sono modi per validarla a meno di volerla usare in un contesto di un modello con supervision, ma si può calcolare la proporzione cumulativa di varianza spiegata che per i primi principal component è:

$$\frac{\sum_{j=1}^k \lambda_j}{\sum_{j=1}^m C_{jj}}$$

Per la eigenvalue decomposition e

$$\frac{\sum_{j=1}^k s_j^2}{\sum_{ij} \bar{X}^2 C_{ji}}$$

Queste formule permettono di stimare la quantità di informazione persa calcolando la percentuale di varianza che si è mantenuta riducendo la dimensionalità dei dati.

**12.2.2.12 Kernel PCA**

La PCA riduce la dimensionalità attraverso una trasformazione lineare. Pertanto utilizzando il kernel trick si può applicare una PCA in uno spazio a più dimensioni ottenendo una trasformazione non lineare nello spazio originale.

**12.3 Clustering****12.3.1 Task**

Si deve trovare una funzione  $f \in N^X$  che assegna ogni input  $x \in X$  a un indice di cluster  $f(x) \in N$ . Tutti i punti mappati allo stesso indice formano un cluster. Ci sono diversi tipi di cluster: partizionali, gerarchici e overlapped. Permette di trovare gruppi di dati con delle proprietà interne, comprimere i dati riducendo il numero di data points invece di ridurre la dimensionalità delle features.

**12.3.2 K-means clustering**

Il K-means clustering richiede di sapere prima il numero  $k$  di gruppi in cui si vogliono dividere i dati. Siano i data points  $X = [x_1, \dots, x_n] \in \mathbb{R}^{d \times n}$ . Fissato un numero di cluster  $k$ , si vuole trovare una partizione di data points in  $k$  insiemi  $\mathcal{L}_1, \dots, \mathcal{L}_k$  che minimizza la variazione  $V(\mathcal{L}_j)$  in ogni set  $\mathcal{L}_j$

$$\min_{\mathcal{L}_1, \dots, \mathcal{L}_K} \sum_{j=1}^k V(\mathcal{L}_j)$$

La variazione è tipicamente data da  $V(\mathcal{L}_j) = \sum_{i \in \mathcal{L}_j} \|x_i - \mu_j\|^2$ , dove  $\mu_j = \frac{1}{|\mathcal{L}_j|} \sum_{i \in \mathcal{L}_j} x_i$ , il centroide di  $\mathcal{L}_j$ . Si deve definire una funzione obiettivo che minimizza la somma di variazioni in ogni set creato. L'algoritmo di ottimizzazione è semplice, inizializza con centroidi casuali e poi mentre i cluster cambiano assegna ogni datapoint al centroide più vicino formando nuovi cluster e computando nuovi centroidi. Questo algoritmo ha una convergenza garantita in quanto migliora strettamente la assignment di cluster e siccome sono finiti a un certo punto converge per forza. Non è comunque garantito trovi il minimo in quanto è un problema NP-hard anche sul piano. È sensibile alla scala delle features.

**12.3.2.1 Distanza**

La distanza euclidea:

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

Non è sempre una buona scelta: per esempio, nel fare clustering di documenti si sceglie una feature per ogni parola e il valore è il numero di volte che tale parola compare. Si nota come qui due distribuzioni possono avere una grande distanza ma distribuzioni simili.

**12.3.2.1.1 Cosine similarity** Per risolvere questo problema si utilizza la cosine similarity:

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

Questa è correlata con l'angolo tra due vettori. Varia tra 0 e 1. È buona per dati di testo e molti altri. Di facile computazione in quanto si possono considerare solo features con valori non zero in entrambi gli esempi. La cosine distance:

$$d(x, y) = 1 - \text{sim}(x, y)$$

### 12.3.2.2 Proprietà di K-means

**12.3.2.2.1 Convergenza** L'algoritmo è garantito che converga in quanto migliora strettamente l'obiettivo se c'è almeno un cambio di cluster e l'insieme delle partizioni è finito. Questo avviene in quanto durante il primo step di assignment ogni altro assignment causerebbe una loss maggiore e durante la computazione del centroide la media di un insieme di valori minimizza l'errore quadratico.

**12.3.2.2.2 Minimo** Non è garantito che trovi il minimo globale ma uno locale. Questo avviene in quanto tipicamente la loss function è non convessa.

**12.3.2.2.3 Selezione dei centroidi** I risultati possono variare enormemente sulla scelta dei centroidi con random seed selection: alcuni possono causare povera convergenza o convergenza a cluster sub-ottimali. Alcune euristiche comuni sono:

- Punti casuali nello spazio.
- Esempi scelti casualmente.
- Punti meno simili a ogni centro esistente.
- Si provano diversi punti di inizio.
- Si inizializza con i risultati di un altro metodo di clustering.

### 12.3.3 Problematiche del clustering

Il clustering presenta diverse problematiche:

- Rappresentazione di un esempio per il cluster.
- Simiglianza e distanza tra esempi.
- Clustering piatto o gerarchico.
- Numero di cluster fisso o data driven.

### 12.3.4 Algoritmi di clustering

#### 12.3.4.1 Flat algorithms

Tipicamente iniziano con un partizionamento causale parziale che viene raffinato iterativamente. Sono per esempio K means, model based e spectral.



### 12.3.4.2 Hierarchical algorithms

Si dividono in agglomerativi bottom-up o divisivi top-down.

### 12.3.4.3 Hard clustering

Nell'hard clustering ogni esempio appartiene esattamente ad un cluster.

### 12.3.4.4 Soft clustering

Nel soft clustering ogni esempio può appartenere a più di un cluster. È pertanto probabilistico.

## 12.3.5 EM clustering

Si nota come il K-means clustering assume sempre dei cluster circolari. Per questo EM clustering assume che i dati vengano da un'insieme di gaussiane: sono così ellittici. Ogni dato viene assegnato a un cluster con una certa probabilità: soft clustering. È molto simile a un alto livello a K-means: itera tra assigning points e ricalcolando i centri dei cluster. Le differenze principali sono che si assumano cluster ellittici e che sia un algoritmo di soft-clustering. Pertanto si inizia con dei centri di cluster iniziale, successivamente si assegnano soft points ad ogni cluster calcolando  $p(\theta_c|x)$  la probabilità che ognuno di essi appartenga a un cluster. Si ricalcolano i centri del cluster con nuovi parametri  $\theta_c$ , la massima probabilità dei centri dati il corrente soft clustering. I centri ottengono una contribuzione pesata dai punti.

### 12.3.5.1 Mixture of Gaussians

In una dimensione una gaussiana:

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

In  $m$  dimensioni invece:

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

Si impara pertanto la media di ogni cluster o suo centro e la matrice di covarianza, ovvero quanto si disperde, la forma del contorno.

### 12.3.5.2 Soft cluster points

Si assegna soft ogni punto a ogni cluster calcolando  $p(\theta_c|x)$  o la probabilità di ogni punto di appartenere al cluster. Si utilizza l'equazione della Gaussiana per ogni cluster normalizzata per creare una probabilità.

### 12.3.5.3 Ricalcolo dei centri

Si calcolano nuovi parametri di cluster  $\theta_c$ . Il maximum likelihood cluster center dati il soft clustering corrente. Questo si fa fittando una gaussiana.

#### 12.3.5.3.1 Fit di una gaussiana

Si fa calcolando la media  $\mu$  e la varianza  $\sigma$  dei dati.

#### 12.3.5.4 Conclusione

EM clustering sta per expectation maximization. Per expectation si intende dato il modello corrente, si trova le probabilità aspettate dei data points ad ogni cluster. La massimizzazione è dato l'assignment probabilistico di tutti i punti la stima del nuovo modello  $\theta_c$ . Come  $k$ -means è garantito che converga ad un ottimo locale, è un algoritmo general purpose per fare training di un modello senza labels.

#### 12.3.6 Altri algoritmi di clustering

K-means e EM-clustering non possono gestire tutte le task di clustering. In particolare non possono gestire dati non distribuiti secondo una gaussiana, soffrono dello stesso problema dei modelli lineari in quanto non sono in grado di prendere decisioni locali.

##### 12.3.6.1 Spectral clustering

Nello spectral clustering avviene una partizione a grafo: si definisce una matrice di somiglianza e si taglia il grafo.

##### 12.3.6.2 Clustering gerarchico

Il clustering gerarchico produce un insieme di cluster nested organizzati come un albero gerarchico detto dendrogram. Viene tipicamente utilizzato per i profili genici.

### 12.4 Density estimation

#### 12.4.1 Task

Si deve trovare una distribuzione di probabilità  $f \in \Delta(X)$  che fitta i dati  $x \in X$ . Permette di ritornare una stima esplicita della distribuzione di probabilità che genera i dati. Permette la generazione di nuovi dati dalla stessa distribuzione e l'individuazione di anomalie.

#### 12.4.2 Generative model

I generative model risolvono la task della density estimation. Sono modelli statistici della distribuzione dei dati sull'input  $p_x$  o di una joint distribution sulla coppia di input label  $p_{XY}$  dipendente dalla disponibilità dei dati obiettivo. La loro abilità principale è di generare nuovi dati dalle distribuzioni osservate.

#### 12.4.3 Modelli discriminativi

I modelli discriminativi sono modelli statistici della distribuzione di probabilità condizionale  $p_{Y|X}$  del target dato l'input. Tipicamente svolgono una task di classificazione come SVM, decision trees e classificatori KNN. Un modello discriminativo può essere costruito da uno generativo attraverso la regola di Bayes ma non viceversa:

$$p_{Y|X}(x) = \frac{p_{XY}(x, y)}{\sum_{y'} p_{XY}(x, y') p_X(x)}$$

### 12.4.4 Tipi di density estimation

Per entrambi i modi si trovano due tipi della task:

- Supervised:  $Z \in X \times Y$ .
- Unsupervised  $Z \in X$ .

#### 12.4.4.1 Explicit density estimation

Si deve trovare una distribuzione di probabilità  $f \in \Delta(Z)$  che fitta i dati  $z \in Z$  dove  $z$  è campionato da una distribuzione di dati sconosciuta  $p_{data} \in \Delta(Z)$ .

#### 12.4.4.2 Implicit density estimation

Si vuole trovare una funzione  $f \in Z^\Omega$  che genera i dati  $f(\omega) \in Z$  da un input  $\omega$  campionato da una distribuzione predefinita  $p_\omega \in \Delta(\Omega)$  in modo che la distribuzione del campione generato fitta la distribuzione sconosciuta  $p_{data} \in \Delta(Z)$ . Non si stima la probabilità  $\omega$  ma ci si concentra unicamente sul riprodurre i dati con la stessa distribuzione di quella sconosciuta originaria.

#### 12.4.4.3 Obiettivo per i modelli generativi

Si deve definire uno spazio di ipotesi  $H \subset \Delta(Z)$ , consiste di un insieme di modelli che possono rappresentare distribuzioni di probabilità definite esplicitamente o implicitamente. Si definisce una misura di divergenza  $d \in R_+^{\Delta(Z) \times \Delta(Z)}$  tra le distribuzioni di probabilità in  $\Delta(Z)$  e si usa la divergenza di Kullback-Leibler. La divergenza è 0 se due distribuzioni hanno un match, altrimenti è negativa. L'algoritmo tenta di trovare  $q^* \in H$  che ha un fit migliore sui dati distribuiti secondo  $p_{data}$  dove il best fit è quello con la divergenza minore.

$$q^* \in \arg \min_{q \in \mathcal{H}} d(p_{data}, q)$$

Si assumerà che la distribuzione dei dati è solo su  $X$  con unsupervised learning, ma il trasporto a supervised è banale.

### 12.4.5 Variational AutoEncode (VAE)

Un autoencoder è un modo per comprimere dati ad alta dimensione in una rappresentazione a meno dimensioni: dimensionality reduction. Un encoder mappa i dati di input  $x$  a una rappresentazione compressa  $\omega$ . La rappresentazione conserva fattori significativi nella variazione dei dati come in PCA.

#### 12.4.5.1 Training

Un encoder viene trainato creando un decoder che mappa la rappresentazione di  $\omega$  indietro nel dominio di input portando a una ricostruzione  $\hat{x}$ . Pertanto l'encoder autoencoda il proprio input. L'obiettivo è quello di minimizzare la divergenza tra l'input  $x$  e la sua ricostruzione  $\hat{x}$  che porta all'algoritmo di training verso la minima perdita di informazione durante la fase di encoding. Dopo il training il decoder non è più necessario in quanto è funzionale solo per stimare l'encoder. L'encoder invece può essere utile per diverse tasks, per esempio per inizializzare o precomputare le caratteristiche per supervised models.

**12.4.5.2 Generare dati con il decoder**

Il decoder potrebbe essere usato per generare nuovi dati ma non li genererà secondo  $p_{data}$  in quanto niente lo obbliga. Se si genera un input a caso  $\omega$  non si è sicuri che  $\omega$  è una combinazione che verrebbe ordinata dalla distribuzione dei dati. La soluzione a questo problema sarebbe di avere una distribuzione a priori  $\Omega$  che ci dice la probabilità di campionare ogni  $\omega$  dallo spazio encodato. Il decoder poi sarebbe capace di tradurre la distribuzione  $\Omega$  nella distribuzione di dati in  $X$ .

**12.4.5.2.1 Training del decoder** In termini formali il decoder:  $q_\theta(x|\omega)$  è una distribuzione di probabilità  $x$  per ogni valore  $\omega$ . La probabilità a priori è  $p_\omega$ . Si può ottenere un'aspettativa marginalizzata

$$q_\theta(x) = \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]$$

Successivamente si pone come obiettivo modificare il parametro  $\theta$  per minimizzare la divergenza tra la distribuzione dei dati e  $q_\theta$ :

$$\theta^* \in \arg \min_{\theta \in \Theta} d(q_\theta, p_{data})$$

Si utilizza KL-divergence. È non negativa e 0 se  $p$  e  $q$  sono la stessa distribuzione.

$$d_{KL}(p, q) = \mathbb{E}_{x \sim p} \left[ \log \frac{p(x)}{q(x)} \right]$$

**12.4.5.2.2 Intrattabilità** Analizzando la divergenza si vede che il suo calcolo è intrattabile a causa del valore atteso:

$$\begin{aligned} d_{KL}(p_{data}, q_\theta) &= \mathbb{E}_{x \sim p_{data}} \left[ \log \frac{p_{data}(x)}{q_\theta(x)} \right] = \\ &= -\mathbb{E}_{x \sim p_{data}} [\log q_\theta(x)] + const = \\ &= -\mathbb{E}_{x \sim p_{data}} [\log \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]] + const \end{aligned}$$

Si può approssimare il valore atteso attraverso stochastic gradient descent.

$$\begin{aligned} \frac{d}{d\theta} d_{KL}(p_{data}, q_\theta) &= -\frac{d}{d\theta} \mathbb{E}_{x \sim p_{data}} [\log \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]] = \\ &= -\mathbb{E}_{x \sim p_{data}} \left[ \frac{d}{d\theta} \log \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)] \right] = \\ &= -\mathbb{E}_{x \sim p_{data}} \left[ \frac{\frac{d}{d\theta} \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]}{\mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]} \right] = \\ &= -\mathbb{E}_{x \sim p_{data}} \mathbb{E}_{\omega \sim p_\omega} \left[ \frac{\frac{d}{d\theta} q_\theta(x|\omega)}{\mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)]} \right] \end{aligned}$$

Queste stime sono comunque dipendenti da  $\omega$  e questo le rende biased e ancora intrattabili.

**12.4.5.2.3 Variational bound** Si introduce pertanto un nuovo termine  $q_\psi(x) \in \Delta(\Omega)$ . Viene usato per calcolare due termini: un reconstruction term e un regolarizzatore. Ora in particolare si

nota come:

$$\begin{aligned}\log \mathbb{E}_{\omega \sim p_\omega} [q_\theta(x|\omega)] &= \log \mathbb{E}_{\omega \sim q_\psi(\cdot|x)} \left[ q_\theta(x|\omega) \frac{p_\omega(\omega)}{q_\psi(\omega|x)} \right] \geq \\ &\geq \mathbb{E}_{\omega \sim q_\psi(\cdot|x)} \left[ \log \left( q_\theta(x|\omega) \frac{p_\omega(\omega)}{q_\psi(\omega|x)} \right) \right] = \\ &= \mathbb{E}_{\omega \sim q_\psi(\cdot|x)} [\log q_\theta(x|\omega)] - d_{KL}(q_\psi(\omega|x), p_\omega)\end{aligned}$$

Dove il primo termine della sottrazione è il ricostruttore e il secondo il regolarizzatore. Si nota come il ricostruttore è ancora di intrattabile computazione ma può essere facile ottenere delle stime del gradiente non biased con rispetto di  $\theta$  e  $\psi$ . Il regolarizzatore può avere una soluzione con forma chiusa utilizzando una distribuzione gaussiana.

**12.4.5.2.4 Training in pratica** Viene campionato un sample  $x$  da  $p_{data}$ . Questo viene passato dal decoder  $q_\psi$  che produce una media  $\mu_{\omega|x}$  e una covarianza  $\Sigma_{\omega|x}$  utilizzati per costruire una gaussiana utilizzata per costruire il regularization term e il sample  $\omega$ .  $\omega$  è l'input del decoder  $q_\theta$  che produce una media  $\mu_{x|\omega}$  e una covarianza  $\Sigma_{x|\omega}$ , utilizzati per costruire una gaussiana, che insieme ad  $x$  costruisce il reconstruction term. Il reconstruction e il regularization term vengono usati per computare il variational lower bound loss, con il secondo come  $P_\omega = N(0, 1)$ , di normale standard e questo viene usato per aggiornare  $\theta$  e  $\psi$ .

#### 12.4.5.3 VAE condizionale

Si assuma di avere un'informazione  $y \in \mathcal{Y}$  e si vuole generare nuovi dati sull'informazione. Denota una distribuzione di probabilità di encoding. Si modifica il encoder e il decoder per prendere le informazioni in input ottenendo

$$\begin{aligned}q_\phi(\omega|x, y) \\ q_\theta(\omega|x, y)\end{aligned}$$

Si definisce i priori condizionati sull'informazione  $p_\omega(\omega|y)$ . Le VAE condizionali sono usate per direzionare l'output che si vuole dal generative model, altrimenti l'output seguirebbe solo la distribuzione di probabilità dei dati.

#### 12.4.5.4 Probabilità dei VAE

- Underfitting: agli stage iniziali il regolarizzatore è troppo forte e tende ad annullare la capacità del modello.
- Blurry samples: il generatore tende a produrre data blurry in quanto la gaussiana tende a produrre blurry risultati.

#### 12.4.6 Generative adversarial Networks (GAN)

Le GAN permettono di stimare la densità implicitamente. Si assuma di avere una densità a priori  $p_\omega \in \Delta(\Omega)$  e un generatore o decoder  $g_\theta \in X^\Omega$  che genera i data points in  $X$  dato un random point da  $\Omega$ . Il generatore campiona direttamente dalla distribuzione implicita che non si conosce. La densità è indotta da  $p_\omega$  e il generatore  $g_\theta$  è dato da  $q_\theta(x) = \mathbb{E}_{\omega \sim p_\omega} \delta[g_\theta(\omega) - x]$  dove  $\delta$  è la funzione

delta di Dirac. L'obiettivo di GAN è trovare  $\theta^*$  tale che  $q_{\theta^*}$  che fitta meglio la distribuzione  $p_{data}$  sotto la divergenza di Jensen-Shannon  $d_{JS}$ :

$$\theta^* \in \arg \min_{\theta} d_{JS}(p_{data}, q_{\theta})$$

Dove:

$$d_{JS}(p, q) = \frac{1}{2}d_{KL}(p, \frac{p+q}{2}) + \frac{1}{2}d_{KL}(q, \frac{p+q}{2})$$

Che è chiaramente intrattabile da computare in quanto richiama la convergenza  $KL$ . Lo stesso vale per il gradiente.

#### 12.4.6.1 Forma equivalente della divergenza JS

$$\begin{aligned} d_{JS}(p, q) &= \frac{1}{2}d_{KL}(p, \frac{p+q}{2}) + \frac{1}{2}d_{KL}(q, \frac{p+q}{2}) = \\ &= \frac{1}{2}\mathbb{E}_{x \sim p}[\log \frac{2p(x)}{p(x)+q(x)}] + \frac{1}{2}\mathbb{E}_{x \sim q}[\log \frac{2q(x)}{p(x)+q(x)}] = \\ &= \frac{1}{2}\mathbb{E}_{x \sim p}[\log \frac{p(x)}{p(x)+q(x)}] + \frac{1}{2}\mathbb{E}_{x \sim q}[\log \frac{q(x)}{p(x)+q(x)}] + \log(2) = \\ &= \log(2) + \frac{1}{2} \max_t \{ \mathbb{E}_{x \sim p}[\log t(x)] + \mathbb{E}_{x \sim q}[\log(1-t(x))] \} \end{aligned}$$

Si deve pertanto imparare  $t(x)$ , un binary classifier che predice se  $x$  deriva da  $p$  o  $q$ .

#### 12.4.6.2 Obiettivo della GAN

Sia  $t_{\varphi}$  un classificatore o discriminatore per i data points in  $\mathcal{X}$ . Allora si trova il limite inferiore sull'obiettivo:

$$\begin{aligned} d_{JS}(p_{data}, q_{\theta}) &= \log(2) + \frac{1}{2} \max_t \{ \mathbb{E}_{x \sim p_{data}}[\log t(x)] + \mathbb{E}_{x \sim q_{\theta}}[\log(1-t(x))] \} \geq \\ &= \log(2) + \frac{1}{2} \max_{\varphi} \{ \mathbb{E}_{x \sim p_{data}}[\log t_{\varphi}(x)] + \mathbb{E}_{x \sim q_{\theta}}[\log(1-t_{\varphi}(x))] \} \end{aligned}$$

Si possono ignorare  $\frac{1}{2}$  e  $\log(2)$  in quanto non cambiano il minimo. Si deve pertanto minimizzare per ottenere il parametro del generatore  $\theta^*$ :

$$\theta^* \in \arg \min_{\theta} \max_{\varphi} \{ \mathbb{E}_{x \sim p_{data}}[\log t_{\varphi}(x)] + \mathbb{E}_{x \sim q_{\theta}}[\log(1-t_{\varphi}(x))] \}$$

Anche in questa forma è intrattabile in quanto dipende dalla densità specifica di  $q_{\theta}$ , ma:

$$\begin{aligned} \mathbb{E}_{x \sim q_{\theta}}[f(x)] &= \int q_{\theta}(x)f(x)dx = \iint \delta[g_{\theta}(\omega) - x]p_{\omega}(\omega)d\omega f(x)dx = \\ &= \iint \delta[g_{\theta}(\omega) - x]f(x)dx p_{\omega}d\omega = \int f(g_{\theta}(\omega))p_{\omega}(\omega)d\omega = \\ &= \mathbb{E}_{\omega \sim p_{\omega}}[f(g_{\theta}(\omega))] \end{aligned}$$

Da cui si può sostituire con  $g_{\theta}$ :

$$\theta^* \in \arg \min_{\theta} \{ \mathbb{E}_{x \sim p_{data}}[\log t_{\varphi}(x)] + \mathbb{E}_{\omega \sim p_{\omega}}[\log(1-t_{\varphi}(g_{\theta}(\omega)))] \}$$

### 12.4.6.3 Game theoretic interpretation

Questo può essere visto come un gioco a due giocatori in cui il giocatore 1 o il generatore tenta di generare i dati che non possono essere distinte dai dati veri. Il giocatore 2 è il discriminatore che cerca di indovinare se l'input viene dalla vera distribuzione o è falso. Il payoff è preso con segno positivo per il discriminatore e negativo per il generatore. Pertanto è descritto come un gioco non cooperativo a due giocatori e zero somma. Il payoff è pertanto:

$$V(\theta, \varphi) = \mathbb{E}_{x \sim p_{data}} [\log t_{\varphi}(x)] + \mathbb{E}_{x \sim p_{\omega}} [\log(1 - t_{\varphi}(g_{\theta}(\omega)))]$$

Si può pertanto scrivere il problema di minmax come:

$$\theta^* \in V(\theta, \varphi)$$

### 12.4.6.4 Ottimizzazione

Il gradiente del obiettivo del GAN rispetto ai parametri del generatore  $\theta$   $\frac{\partial}{\partial \theta} \max_{\varphi} V(\theta, \varphi)$  e richiede la risoluzione del problema di massimizzazione con rispetto al parametro del discriminatore  $\varphi$ . Questo dovrebbe essere proibitivo computazionalmente. In pratica si alterna uno step di update per il generatore dopo  $k$  update per il discriminatore, senza garanzia di convergenza.

#### 12.4.6.4.1 Esempio con vanilla SGD

$$\varphi + \nu \frac{\partial}{\partial \varphi} V(\theta, \varphi).$$

- Il discriminatore si aggiorna  $k$  volte:  $\varphi =$
- Il generatore si aggiorna  $\theta = \theta - \nu \frac{\partial}{\partial \theta} V(\theta, \varphi)$

Le aspettative nei gradienti con rispetto a  $p_{data}$  e a  $p_{\omega}$  sono stimate su mini-batch.

### 12.4.6.5 Problemi con GAN

- Training stability: i parametri oscillano e non convergono.
- Mode collapse: il generatore potrebbe imparare a perfezionare pochi esempi dal training set e riprodurre solo quegli esempi per vincere sempre.
- Vanishing gradient: se il discriminatore è molto bravo e lascia il generatore con poco gradiente per imparare.

### 12.4.6.6 Altri tipi di GAN

Diversi modelli simili a GAN possono essere costruiti considerando diverse divergenze tra le probabilità e applicando trucchi simili per eliminare la necessità di conoscere la densità esplicitamente:

- *f-GANS* costruite su f-divergenza.
- *b-GANS* costruite su Bergman divergenza.
- *Wasserstein GAN* utilizzano la Wasserstein metric.
- Altre GAN possono essere derivate utilizzando altre metriche integrali di probabilità.
- *f-GANS* su f-divergenza.

GAN e VAE possono essere combinate e GAN condizionali esistono che lavorano simile a VAE continual.

## Capitolo 13

# Reinforcement learning

### 13.1 Introduzione

L'idea del Reinforcement learning è che si ha un agente e un ambiente: l'agente svolge azioni che cambiano l'ambiente e conseguentemente l'ambiente ritorna all'agente delle rewards. Questa idea viene presa dalle strategie di animali:

- L'ambiente si trova in uno stato  $s$ .
- L'agente svolge un'azione.
- L'azione modifica l'ambiente.
- L'ambiente ritorna una reward all'agente e il nuovo stato  $s'$ .

#### 13.1.1 Policy

L'agente tenta di imparare una policy o un mapping da stati a azioni in modo da massimizzare la reward data dall'ambiente.

### 13.2 Markov decision process (MDP)

MDP utilizza l'assunzione di Markov in cui uno stato al tempo  $t$  dipende solo dallo stato precedente  $s(t-1)$  e dall'azione precedente  $a(t-1)$  per trovare una policy. Questo utilizza l'equazione di Bellman e la programmazione dinamica. L'obiettivo è trovare una policy, una mappa che ritorna tutte le azioni ottimali di ogni stato sul nostro ambiente.

#### 13.2.1 Componenti

- Stati  $s_i$  che iniziano con uno stato  $s_0$ .
- Azioni  $a$ .
- Modello di transizione  $P(s'|s, a)$ , secondo l'assunzione di Markov la probabilità di arrivare a  $s'$  da  $s$  dipende solo da  $s$  e non da nessuna delle azioni passate o stati passati.



- Reward function  $r(s)$ .
- Policy  $\pi(s)$ , l'azione che un agente svolge in ogni stato dato.

Si nota come una MDP è definita dalla tupla  $(S, A, R, P, \gamma)$ , dove  $S$  è l'insieme degli stati possibili,  $A$  l'insieme delle azioni possibili,  $R$  la distribuzione delle reward dato uno stato,  $P$  la probabilità di transizione o la distribuzione rispetto al prossimo stato dato  $(stato, azione)$  e  $\gamma$  il discount factor. La reward viene utilizzata per ottenere la policy migliore. L'obiettivo è pertanto di trovare la policy ottimale.

### 13.2.2 Ambienti stocastici

MDP viene utilizzata tipicamente negli ambienti stocastici, pertanto lo stato a tempo  $t + 1$  non è deterministico quando si conosce lo stato  $s_t$  e l'azione  $a_t$  :  $s_{t+1}$  viene estratta dalla probabilità di transizione e deriva da un processo stocastico.

### 13.2.3 MDP loop

- A tempo  $t = 0$  l'ambiente si campiona nello stato iniziale  $s_0 \sim p(s_0)$ .
- Si ripete:
  - L'agente seleziona l'azione  $a_i$ .
  - L'ambiente campiona la reward  $r_t \sim R(\cdot | s_t, a_t)$ .
  - L'ambiente campiona il prossimo stato  $s_{t+1} \sim P(\cdot | s_t, a_t)$
  - L'agente riceve la reward  $r_t$  e lo stato successivo  $s_{t+1}$ .

### 13.2.4 Policy

La policy  $\pi(s)$  è la funzione da  $S$  ad  $A$  che specifica quale azione prendere in ogni stato. Si deve trovare la policy  $\pi^*$  che massimizza cumulative discounted rewards. La reward viene data dalla funzione

$$R : (stato, azione) \rightarrow reward$$

#### 13.2.4.1 Cumulative discounted reward

Si supponga di avere una policy  $\pi$  in uno stato di inizio  $s_0$  che porta a una sequenza  $s_0, \dots, s_n$ . La cumulative reward della sequenza è:

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

È pertanto la somma delle reward di una serie di stati. Quella discounted è una cumulative reward che dà un peso a diverse reward in base al tempo dello step:

$$\sum_{t \geq 0} \gamma^t r(s_t) \quad \text{where } 0 < \gamma \leq 1$$

Il discount factor è un modo di pesare l'importanza dei primi passi o dei passi futuri in accordo con il valore di  $\gamma^t$ . Minore è minore l'importanza di reward future e l'agente tende a concentrarsi su azioni che portano a reward immediate. Questa strategia aiuta gli algoritmi a convergere.

## 13.3 Confronto tra reinforcement learning e supervised learning

### 13.3.1 Supervised learning loop

- Prendi input  $x_i$  campionato dalla distribuzione dei dati.
- Utilizza un modello con parametri  $w$  per predire l'output  $y$ .
- Osserva l'output obiettivo  $y_i$  e loss  $l(w, x_i, y_i)$ .
- Aggiorna  $w$  per ridurre la loss con SGD:  $w = w - \eta \nabla l(w, x_i, y_i)$ .

### 13.3.2 Reinforcement learning loop

- Dallo stato  $s_i$  svolgi un'azione  $a$  determinata dalla policy  $\pi(s)$ .
- L'ambiente seleziona lo stato successivo  $s'$  in base al modello di transizione  $P(s'|s, a)$ .
- Osserva  $s'$  e la reward  $r(s)$ , aggiornando la policy.

### 13.3.3 Differenze fondamentali

- Nel SL i dati non dipendono dall'input precedente, mentre in RL le azioni dell'agente influiscono il prossimo input.
- In SL si ha una loss function che guida il modello ai parametri migliori e si può differenziare con rispetto ai pesi, mentre in RL la reward guida lo sviluppo del modello, ma non è differenziabile rispetto ai parametri.
- In SL si trova supervisione rispetto ad ogni passo, mentre in RL le rewards potrebbero essere sparse.

## 13.4 Metodi di reinforcement learning

Per RL si trovano due approcci principali:

- Metodi basati sul valore: una value function  $V(s)$  viene introdotta e l'agente vuole massimizzarla. La value di ogni stato è la quantità totale di reward un agente può aspettarsi di collezionare rispetto al futuro cominciando da uno stato dato.
- Metodo basato sulla policy: si definisce una policy da ottimizzare correttamente, questa definisce come l'agente si comporta nella forma di distribuzione di probabilità tra le azioni da svolgere in uno stato dato. Una policy stocastica dà la distribuzione di probabilità su azioni diverse:  $\pi_\theta(s, a) \approx P(a|s)$ .

### 13.4.1 Value based methods

La funzione di value è una funzione dello stato con rispetto a una certa policy  $\pi$ . Ritorna la quantità totale di reward che l'agente può aspettarsi da uno stato particolare a tutti gli stati possibili a partire da quello. Attraverso la value si può trovare una policy. La value function  $V$  di uno stato con rispetto a una policy  $\pi$  è l'aspettata cumulative reward di seguire tale policy cominciando in  $s$ :

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) | s_0 = s, \pi \right]$$

Con  $a_t = \pi(s_t)$ ,  $s_{t+1} \sim P(\cdot | s_t, a_t)$ . Il valore ottimale di uno stato è il valore raggiungibile seguendo la migliore policy:

$$V^*(s) = \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r(s_t) | s_0 = s, \pi \right]$$

Dice pertanto quanto è buono uno stato. Si trova il valore atteso in quanto la sequenza di stati non è deterministica e non si ha accesso alla probabilità di transizione. Rappresenta il valore che l'agente può aspettarsi da tutti gli stati possibili cominciando dallo stato iniziale  $s$ .

#### 13.4.1.1 Q value function

Invece di gestire con la value function si utilizza la  $Q$ -value function che lavora con uno stato  $s$ , una azione  $a$  e una policy  $\pi$ . Viene definita come:

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

Il valore ottimale di  $Q$  dice quando è buona una coppia stato azione:

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

La  $Q$  value ottimale viene usata per computare la policy ottimale:

$$\pi^*(s) = \arg \max_a Q^*(s, a)$$

La formula risultante della  $Q$  value function è nella formula di un'equazione di Bellman:

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

Se il valore ottimale della coppia stato azione per il prossimo passo  $Q^*(s', a')$  sono conosciuti allora la strategia ottima è svolgere l'azione che massimizza il valore atteso. Per calcolare il valore  $Q$  dello stato corrente si deve calcolare il valore  $Q$  degli stati vicini e così via con ricorsione.

#### 13.4.1.2 Algoritmo Q-learning

Lo scopo di questo algoritmo è che la matrice di value  $R$  è conosciuta unicamente all'ambiente e l'agente deve impararla con l'esperienza. L'agente possiede una matrice  $Q$  che codifica stato, azione e rewards, ma è inizializzata a 0 e diventa  $R$  attraverso l'esperienza. La policy si ottiene attraverso questa matrice.

1. Inizializza la matrice  $Q$  con zero.
2. Seleziona uno stato iniziale casuale.
3. Per ogni episodio (insieme di azioni che parte dallo stato iniziale e arriva allo stato finale).
4. Mentre lo stato non è lo stato obiettivo.
5. Seleziona una possibile azione casuale per lo stato corrente.
6. Utilizzando l'azione considera arrivare allo stato prossimo.
7. Ottieni il valore di  $Q$  massimo per il prossimo stato.
8.  $Q^*(s, a) = R(s, a) + \gamma \max_{a'} [Q^*(s', a')]$ .

Per trovare la policy ottimale:

1. Si pone lo stato corrente a quello iniziale.
2. Dallo stato corrente si trova l'azione con il maggiore  $Q$  value.
3. Si pone lo stato corrente al prossimo.
4. Si ripetono i passi 2 e 3 fino a che si raggiunge lo stato obiettivo.

#### 13.4.1.3 Deep Q learning

L'equazione di Bellman può pertanto essere usate per imparare tabelle  $Q$  per ritornare la policy ottimale. Nel mondo reale però gli stati e le rewards possono essere troppo grandi per essere calcolate completamente. Per questa ragione viene introdotto un algoritmo di deep  $Q$  learning che sfrutta l'idea di approssimare  $Q$  utilizzando una funzione parametrica trainata da una NN. Si può approssimare  $Q$  a

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

In quanto  $Q_w$  è una funzione dei pesi  $w$  si può trainare una NN per approssimare  $Q_w$ . Ad ogni interazione del training si aggiornano i parametri  $w$  del modello per avvicinare  $Q$  a  $y$ :

$$y_i(s, a) = \mathbb{E}_{s' \sim P(\cdot|s, a)} [r(s) + \gamma \max_{a'} Q_{w_{i-1}}(s', a') | s, a]$$

La loss function

$$L_i(w_i) = \mathbb{E}_{s, a \sim \rho} [(y_i(s, a) - Q_{w_i}(s, a))^2]$$

Dove  $\rho$  è la distribuzione di probabilità sugli stati  $s$  e le azioni  $a$  che si riferiscono come distribuzione di comportamento. Si definisce a ogni iterazione  $i$  e un target  $y_i$  ottenuti attraverso la Bellman equation. Dati questi valori si può derivare  $L_i$ . Aggiornamento del gradiente:

$$\begin{aligned} \nabla_{w_i} L(w_i) &= \mathbb{E}_{s, a \sim \rho} [(y_i(s, a) - Q_{w_i}(s, a)) \nabla_{w_i} Q_{w_i}(s, a)] = \\ &= \mathbb{E}_{s, a \sim \rho, s'} [(r(s) + \gamma \max_{a'} Q_{w_{i-1}}(s', a') - Q_{w_i}(s, a)) \nabla_{w_i} Q_{w_i}(s, a)] \end{aligned}$$

Attraverso il training SGD si sostituisce il valore atteso campionando l'esperienza  $s, a, s'$  utilizzando la distribuzione di comportamento e il modello di transizione.

**13.4.1.3.1 Problematiche** Il training è pronò all'instabilità e a differenza del supervised learning i target si muovono. Esperienze successive sono correlate e dipendenti dalla policy che può cambiare rapidamente con piccoli cambi nei parametri, portando a un cambio drastico nella distribuzione dei dati. Per risolverli si può fare freezing sulla target Q network per aumentare la stabilità della rete e utilizzare un experience replay: un buffer per salvare esperienze e campionare da quello in modo da seguire una strategia greedy che permette un trade-off tra exploitation ed exploration.

## 13.4.2 Policy gradient methods PGM

L'obiettivo dei PGM è di definire parametri che influenzano la policy  $\pi$  e di impararli direttamente. Questa è la soluzione più semplice nel caso in cui si abbia uno spazio degli stati grande e continuo. L'idea è di utilizzare deep neural network per imparare la policy. Si deve pertanto imparare una funzione che ritorna la distribuzione di probabilità di azioni rispetto allo stato corrente:

$$\pi_\theta(s, a) \approx P(s)$$

Si noti come questa ha un significato diverso rispetto ai value-based method: è una probabilità di azioni rispetto agli stati, non una funzione da azioni a stati. In questo caso non si hanno le labels in modo da dare feedback alla NN attraverso backpropagation e viene usata la reward. Si devono pertanto trovare i migliori parametri  $\theta$  della policy per massimizzare la reward aspettata attraverso gradient descent:

$$\begin{aligned} J(\theta) &= \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi_\theta \right] \\ &= \mathbb{E}_\tau [r(t)] \end{aligned}$$

Si vuole il valore atteso di ritorno di traiettorie:  $\tau(s_0, a_0, r_0, \dots, s_n, a_n, r_n)$

$$J(\theta) = \int_\tau r(\tau) p(\tau, \theta) d\tau$$

Dove  $p(\tau, \theta)$  è la probabilità della traiettoria  $\tau$  sotto la policy con i parametri  $\theta$ :

$$p(\tau, \theta) = \prod_{t \geq 0} \pi_\theta(s_t, a_t) P(s_{t+1} | s_t, a_t)$$

Le traiettorie sono una semplificazione, un oggetto che rappresenta stato, azione e reward. In quanto l'integrale è di difficile risoluzione si differenzia utilizzando la log-trasformazione:

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau [r(t) \nabla_\theta \log p(\tau, \theta)]$$

Dove

$$\log p(\tau, \theta) = \sum_{t \geq 0} [\log \pi_\theta(s_t, a_t) + \log P(s_{t+1} | s_t, a_t)]$$

E

$$\nabla_\theta \log p(\tau, \theta) = \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(s_t, a_t)$$

In questo modo si può computare il gradiente senza conoscere la probabilità di transizione  $p$ .

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau \left[ \left( \sum_{t \geq 0} \gamma^t r_t \right) \left( \sum_{t \geq 0} \nabla_\theta \log \pi_\theta(s_t, a_t) \right) \right]$$

Si può inoltre evitare di computare tutte le traiettorie possibili e usare unicamente un'approssimazione stocastica a  $N$  traiettorie.

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^n \left( \sum_{t=0}^{T_i} y^t r_{i,t} \right) \left( \sum_{t=0}^{T_i} \nabla_{\theta} \log \pi_{\theta}(s_{i,t}, a_{i,t}) \right)$$

#### 13.4.2.1 Reinforce

Reinforce è un algoritmo per deep RL che aggiorna i parametri:

1. Campiona  $N$  traiettorie  $\tau_i$  utilizzando la policy corrente  $\pi_{\theta}$ .
2. Stima il gradiente di policy  $\nabla_{\theta} J(\theta)$ .
3. Aggiorna i parametri attraverso gradient ascent  $\theta = \theta + \eta \nabla_{\theta} J(\theta)$ .

#### 13.4.2.2 Single step reinforce

Single step reinforce è una variante del reinforce che stima il gradiente unicamente su uno step e non sull'intera traiettoria:

1. In uno stato  $s$  campiona l'azione  $a$  utilizzando la policy corrente  $\pi_{\theta}$  e si osserva la reward  $r$ .
2. Stima il gradiente  $\nabla_{\theta} J(\theta) \sim r \nabla_{\theta} \log \pi_{\theta}(s, a)$
3. Aggiorna i parametri attraverso gradient ascent  $\theta = \theta + \eta \nabla_{\theta} J(\theta)$ .

Si nota come  $r$  alto aumenta la probabilità dell'azione vista, mentre se è basso si abbassa la probabilità. Si aggiorna il parametri  $\theta$  in accordo con il gradiente in quanto si vuole aumentare la reward o gradient ascent.