

Appunti di Machine Learning v0.1.2

Piero Molino

Indice

1	Introduzione	3
1.1	Apprendimento Supervisionato	3
1.2	Apprendimento Non Supervisionato	4
I	Apprendimento Supervisionato	6
2	Regressione Lineare	7
2.1	Regressione Lineare Univariata	7
2.2	Discesa del Gradiente	8
2.3	Discesa del Gradiente per la Regressione Lineare Univariata . . .	9
2.4	Regressione Lineare Multivariata	11
2.5	Discesa del Gradiente per la Regressione Lineare Multivariata . .	11
2.6	Feature Scaling	12
2.7	Regressione Polinomiale	12
2.8	Equazioni Normali	13
3	Regressione Logistica	15
3.1	Discesa del Gradiente per la Regressione Logistica	17
3.2	Algoritmi di ottimizzazione	18
3.3	Classificazione Multiclasse con la Regressione Logistica	19
4	Regolarizzazione	20
4.1	Regolarizzazione nella Regressione Lineare	20
4.2	Regolarizzazione nella Regressione Logistica	21
5	Reti Neurali	23
5.1	Rappresentazione di una Rete Neurale	23
5.2	Esempio: funzioni logiche	25
5.3	Classificazione Multiclasse con le Reti Neurali	28
5.4	Funzione Costo delle Reti Neurali	28
5.5	Algoritmo di Backpropagation	28
5.6	Parametri di una Rete Neurale	30

<i>INDICE</i>	2
6 Applicazione e Valutazione	31
6.1 Diagnosi (Train and Test)	31
6.2 Model Selection	32
6.3 Bias e Varianza	32
6.4 Learning Curves	32
6.5 Consigli pratici	34
6.6 Progettazione di un sistema di Machine Learning	34
6.7 Metriche di valutazione	35
6.8 Dati e Algoritmi	37
7 Support Vector Machines	38
7.1 Funzione Costo ed Ipotesi delle Support Vector Machines	38
7.2 Large Margin Classifier	39
7.3 Kernels	41
7.4 Clasificazione Multiclasse con le Support Vector Machines	44
8 Machine Learning su larga scala	45
8.1 Scelta dell'algoritmo	45
8.2 Stochastic Gradient Descent	45
8.3 Mini-batch Gradient Descent	46
8.4 Convergenza	47
8.5 Online Learning	47
II Apprendimento Non Supervisionato	48
9 Clustering	49
9.1 K-Means	50

Capitolo 1

Introduzione

Definizione di Arthur Samuel (1959): Machine learning, field of study that gives computers the ability to learn without being explicitly programmed.

Definizione di Tom M. Mitchell (1998): A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Due macrocategorie:

- **Supervised Learning** (apprendimento supervisionato): vengono forniti esempi con relative feature in input e un valore di output e si vuole trovare quella funzione che dato un input non conosciuto calcola l'output.

L'obiettivo è quello di fare una predizione basata su proprietà conosciute apprese dai dati in input.

- **Unsupervised Learning** (apprendimento non supervisionato): vengono forniti esempi con relative feature in input, ma non viene fornito alcun valore di output. L'obiettivo è scoprire nuove proprietà nei dati forniti in input, ad esempio raggruppandoli.

1.1 Apprendimento Supervisionato

Vengono forniti esempi con relative feature in input e un valore di output. Se il valore di output è discreto, ad esempio l'appartenenza o la non appartenenza ad una determinata classe, il problema è di **Classificazione**. Se invece l'output è un valore reale continuo in un determinato range allora il problema è di **Regressione**. In entrambi i casi si vuole trovare quella funzione, chiamata ipotesi h , che dato un input non conosciuto stima il valore dell'output. L'obiettivo dei problemi di **Supervised Learning** (apprendimento supervisionato) è quello di fare una predizione basata su proprietà conosciute apprese dai dati in input.

I dati in input vengono chiamati *training set* e sono composti da un numero n di esempi composto da una serie di feature. Il valore di output è chiamato *target*.

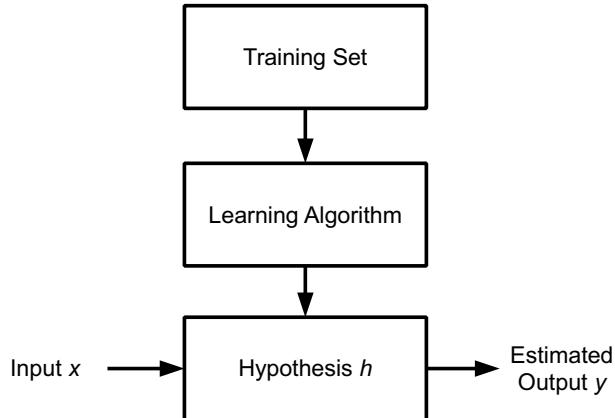


Figura 1.1.1: Supervised Learning

value. Per identificare ciascun esempio e relativo output si userà la notazione $x^{(i)}$ per indicare l'i-esimo esempio e $y^{(i)}$ per indicare l'i-esimo valore di output. Nel caso in cui ci siano feature multiple si userà la notazione $x_j^{(i)}$ per indicare la j-esima feature dell'i-esimo esempio.

1.2 Apprendimento Non Supervisionato

Nell'**Unsupervised Learning** (apprendimento non supervisionato) vengono forniti esempi con relative feature in input, ma non viene fornito alcun valore di output, quindi si hanno a disposizione dati non etichettati. L'obiettivo di questa famiglia di algoritmi è quello di trovare delle strutture all'interno di questi dati non etichettati. Se si vogliono identificare dei raggruppamenti di elementi simili, è un problema di **Clustering**. Se invece si vogliono identificare le diverse sorgenti che hanno contribuito alla creazione dei dati, il problema è di **Blind Source Separation**. Esistono altri problemi di apprendimento non supervisionato, tutti condividono l'avere dati privi di etichette o valori di output e consistono nello scoprire nuove proprietà nei dati forniti in input.

Supervised learning

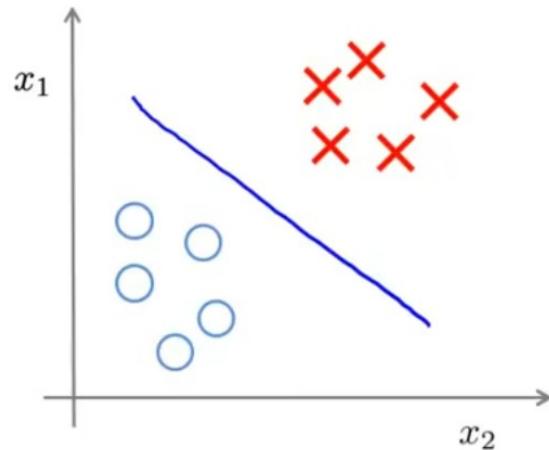


Figura 1.1.2: Esempio di Supervised Learning

Unsupervised learning

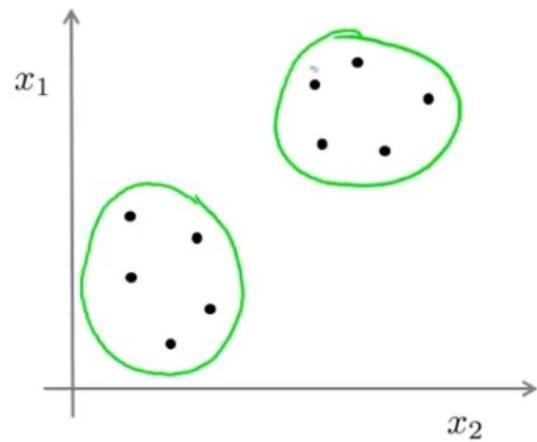


Figura 1.2.1: Esempio di Unsupervised Learning

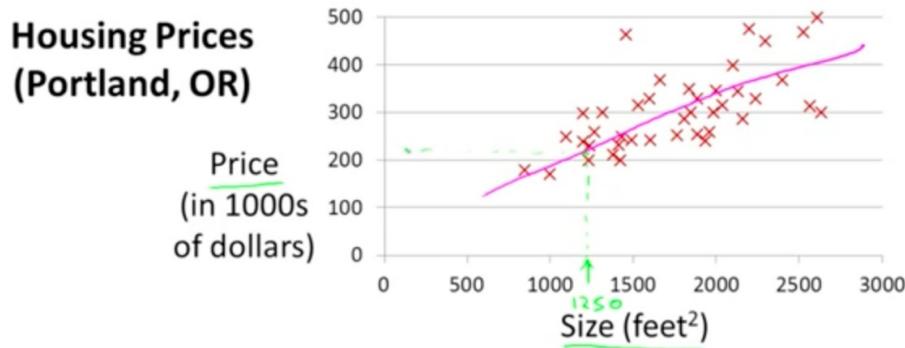
Parte I

Apprendimento Supervisionato

Capitolo 2

Regressione Lineare

La **Linear Regression** (regressione lineare) è un metodo di apprendimento supervisionato che crea un modello di regressione lineare. Permette di apprendere una funzione che, dato un esempio non conosciuto precedentemente ipotizza il valore di output, come ad esempio il caso di predire il costo di una casa data la sua area come mostrato in figura.



2.1 Regressione Lineare Univariata

Un possibile modello della funzione ipotesi h è

$$h_{\theta}(x) = h(x) = \theta_0 + \theta_1 x$$

in cui i θ_i sono parametri della funzione che andranno stimati in base al training set. Nel caso in cui si utilizzi una sola variabile x si parla di **Univariate Linear Regression** (regressione lineare univariata).

L'idea di base è di scegliere i parametri θ_i in modo tale che $h_{\theta}(x)$ sia il più vicino possibile ad y negli esempi di apprendimento costituiti da coppie

(x, y) . Per fare ciò si definisce un problema di minimizzazione che minimizzi la distanza quadratica tra $h_\theta(x)$ e y ,

$$\min_{\theta_0 \theta_1} \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

dove m è il numero di esempi di apprendimento nel training set. La funzione da minimizzare prende il nome di **Cost Function** o funzione costo e si denuncia con J , in questo caso è lo **Squared Error** o errore quadratico. Esistono altre funzioni costo, ma l'errore quadratico funziona bene nella maggior parte dei casi. Possiamo dunque riscrivere il problema di ottimizzazione come

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\min_{\theta_0 \theta_1} J(\theta_0, \theta_1)$$

Riassumendo abbiamo:

- Ipotesi: $h_\theta(x) = h(x) = \theta_0 + \theta_1 x$
- Parametri: θ_0, θ_1
- Funzione Costo: $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2$
- Obiettivo: $\min_{\theta_0 \theta_1} J(\theta_0, \theta_1)$

2.2 Discesa del Gradiente

Per ottenere i parametri tali che la funzione costo sia minimizzata esistono vari metodi, uno dei più utilizzati è il **Gradient Descent** (discesa del gradiente). Esso consiste nel modificare iterativamente il valore dei parametri θ_i nella direzione della derivata della funzione costo finché non si raggiunge il minimo. Il minimo che si può raggiungere è un minimo locale, quale minimo locale verrà raggiunto dipende dal punto di partenza, ovvero i valori iniziali dei parametri θ_i . Un esempio grafico è mostrato in Figura 2.2.1. Le iterazioni possono essere fermate anche in base ad un criterio di convergenza quale la differenza dei parametri θ_i rispetto al passo precedente (che indica se il valore dei parametri non sta migliorando in modo sensibile tra un'iterazione e l'altra) o una soglia prefissata per la funzione costo, sotto la quale i parametri θ_i vengono accettati.

In pratica viene ripetuto finché non si raggiunge la convergenza (o un qual-sivoglia criterio di convergenza) il seguente passo

$$\theta_k := \theta_k - \alpha \frac{\partial}{\partial \theta_k} J(\theta_0, \dots, \theta_i)$$

dove $:=$ significa assegnare simultaneamente e α è un parametro chiamato **Learning Rate**.

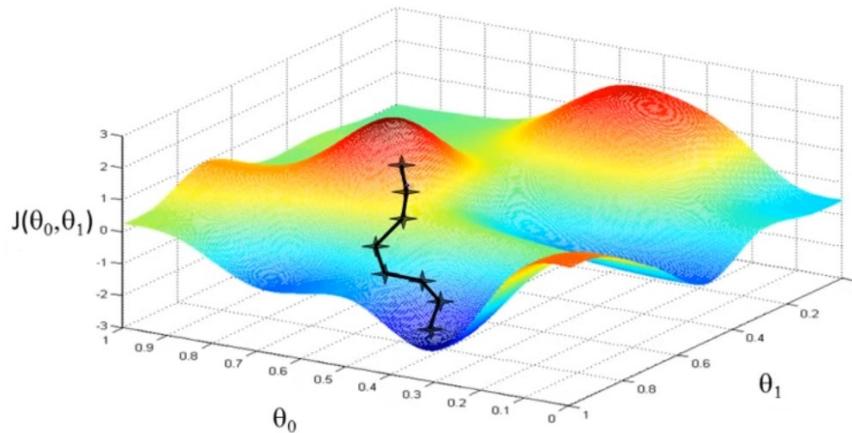


Figura 2.2.1: Gradient Descent

Il learning rate regola la velocità di convergenza: se è molto piccolo, la velocità di convergenza sarà bassa, mentre se è grande la convergenza sarà più veloce ma l'algoritmo potrebbe non convergere, come mostrato in Figura 2.2.2. Inoltre, avvicinandosi al minimo, l'algoritmo effettuerà passi più piccoli in quanto l'inclinazione della tangente diminuirà e quindi la derivata della funzione costo diminuirà, quindi non c'è bisogno di diminuire il learning rate α nel tempo.

Un criterio per poter scegliere il learning rate è visualizzare l'andamento della funzione costo J al variare di α all'interno di valori di diversi ordini di grandezza come ad esempio moltiplicando sempre per 10 ottenendo $\dots, 0.001, 0.01, 0.1, 1, \dots$ oppure moltiplicando per 3 ottenendo $\dots, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, \dots$

2.3 Discesa del Gradiente per la Regressione Lineare Univariata

Calcolando le derivate parziali, si ottengono le funzioni per aggiornare i due parametri nel caso della regressione lineare:

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

Poiché la funzione costo per la regressione lineare è sempre convessa, ovvero ha un solo minimo ed è il minimo globale, il gradient descent converge sempre.

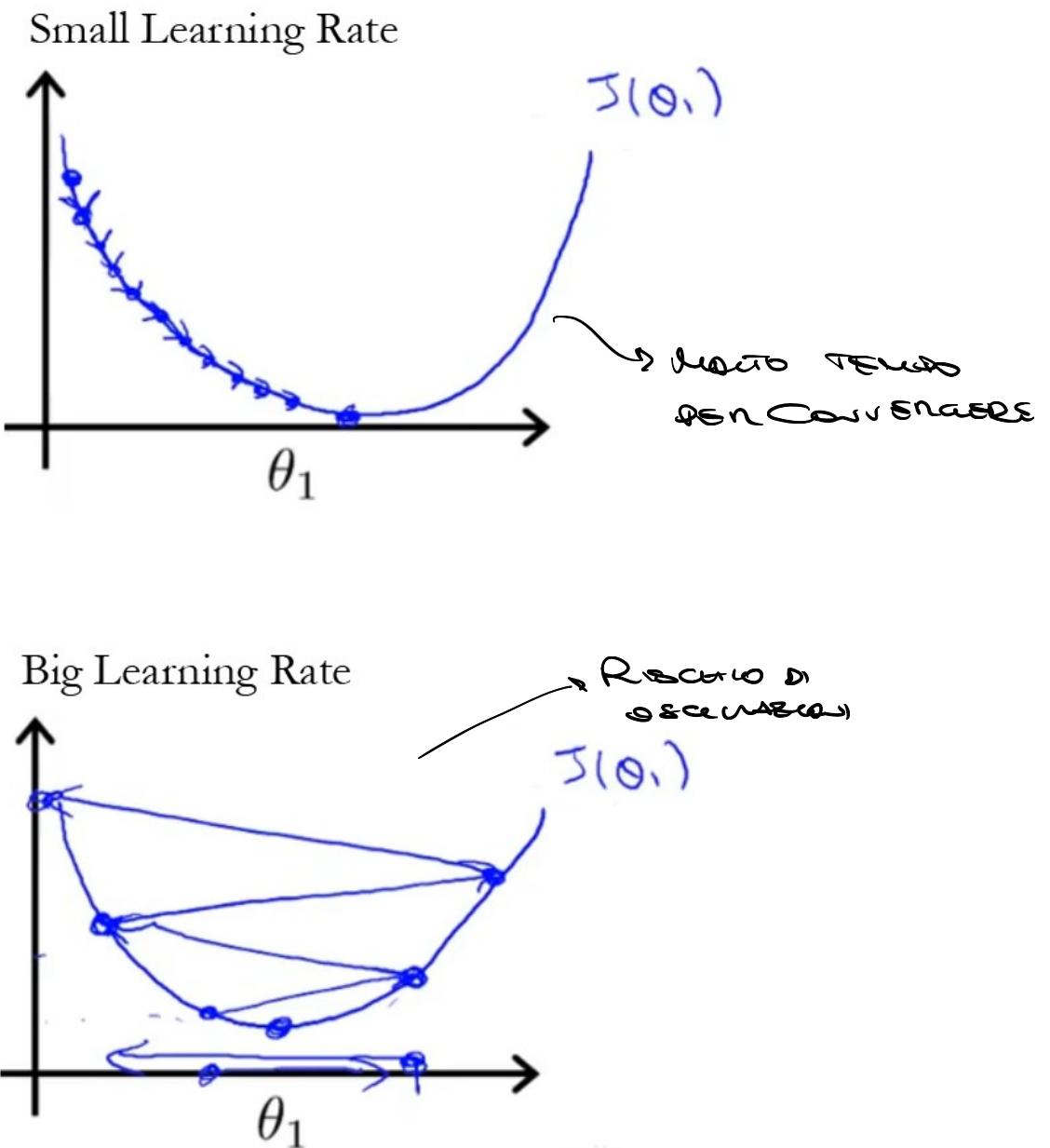


Figura 2.2.2: Learning Rate

2.4 Regressione Lineare Multivariata

La Multivariate Linear Regression (regressione lineare multivariata) è la regressione lineare nel caso in cui si utilizzi più di una variabile x , quindi nel caso in cui ogni esempio x è composto da più feature x_1, \dots, x_n . Si userà la notazione $x_j^{(i)}$ per indicare la j -esima feature dell' i -esimo esempio.

la funzione ipotesi h sarà diversa da quella per il caso univariato perché dovrà prevedere anche le altre feature ed i relativi parametri θ_i :

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

Per convenienza di notazione definiamo $x_0 = 1$. Così facendo possiamo riscrivere l'ipotesi h come:

$$h_{\theta}(x) = \theta_0 x_0 + \theta_1 x_1 + \dots + \theta_n x_n$$

Racchiudendo i parametri all'interno di dei vettori possiamo infine scrivere una versione più compatta dell'ipotesi h :

$$\begin{aligned} x &= \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \\ \theta &= \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1} \end{aligned}$$

$$h_{\theta}(x) = \theta^{\top} x$$

2.5 Discesa del Gradiente per la Regressione Lineare Multivariata

Riscriviamo la funzione costo J e il passo di aggiornamento dei parametri in forma vettorializzata come fatto per l'ipotesi h .

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2$$

$$\theta_k := \theta_k - \alpha \frac{\partial}{\partial \theta_k} J(\theta)$$

Calcolando le derivate parziali, si ottengono le funzioni per aggiornare i parametri:

$$\theta_k := \theta_k - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)}$$

La differenza rispetto al caso univariato è la presenza di $x_k^{(i)}$ in tutti i casi, anche per θ_0 , solo che essendo $x_0 = 1$, non c'è alcuna differenza rispetto al caso univariato, che può essere visto un caso particolare del multivariato.

2.6 Feature Scaling

L'idea di base è fare in modo che tutte le feature siano in una scala simile, poiché la convergenza del gradient descent può essere molto lenta se le scale dei valori delle feaure sono molto diverse. Si deve fare in modo che i valori delle feature siano approssimativamente compresi tra -1 e $+1$. Non è importante che la scala sia esattamente la stessa, basta che non siano molto diverse, una regola del pollice può essere quella di cercare di non cambiare l'ordine di grandezza.

Una possibilità è di utilizzare la *Mean Normalization* che consiste nel sottrarre la media μ e dividere per il range, ovvero il massimo meno il minimo:

$$x_i := \frac{x_i - \mu_i}{\max(x_i) - \min(x_i)}$$

In alternativa si può utilizzare la *Z-Score Normalization* in cui si divide il valore per la varianza σ :

$$x_i := \frac{x_i - \mu_i}{\sigma_i}$$

Un esempio dell'impatto del feature scaling è presentato in Figura 2.6.1.

Un altro importante fattore da prendere in considerazione è la scelta delle feature, in quanto spesso usare un numero minore di feature più informative può risultare in modelli migliori rispetto all'utilizzo di molte feature poco informative. Ad esempio per predire il prezzo di una casa probabilmente è più utile utilizzare come feature la sua area che la sua lunghezza e larghezza.

2.7 Regressione Polinomiale

Non sempre un modello lineare è il più adatto a descrivere i dati di input. Per ottenere modelli polinomiali è sufficiente modificare l'ipotesi h per prevedere gradi non unitari per alcune variabili. In questo modo si possono ottenere modelli quadratici.

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

Questi modelli descrivono una parabola e non sempre sono adatti ai dati. Ad esempio un modello del genere che valuta il prezzo di una casa in base alla

Feature Scaling

Idea: Make sure features are on a similar scale.

E.g. $x_1 = \text{size (0-2000 feet}^2)$

$x_2 = \text{number of bedrooms (1-5)}$

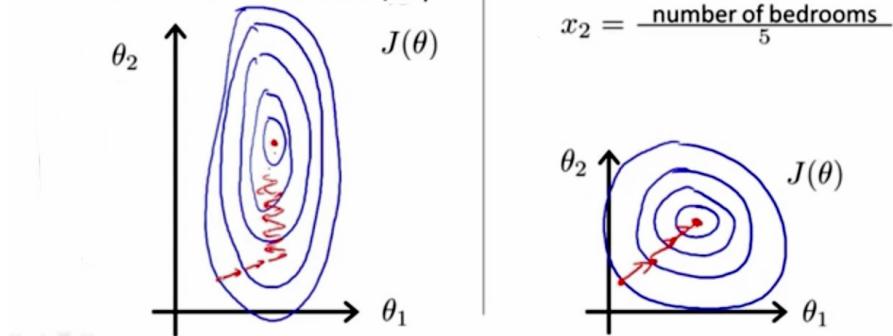


Figura 2.6.1: Esempio di Feature Scaling

sua area prevederebbe l'abbassamento dei prezzi in modo parabolico dopo una certa dimensione.

Solitamente in questi casi si preferisce usare modelli cubici:

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

che invece continuano ad aumentare, oppure, per evitare problemi di scalatura delle feature, modelli che utilizzano la radice quadrata, che ha un andamento più prevedibile:

$$h_{\theta}(x) = \theta_0 + \theta_1 x + \theta_2 \sqrt{x_2}$$

Un esempio delle differenze che i termini polinomiali possono apportare è mostrata in Figura 2.7.1.

2.8 Equazioni Normali

Per trovare il minimo della funzione J in modo analitico, ovvero per trovare i parametri θ , esiste un metodo più semplice del gradient descent: le **Normal Equations** (equazioni normali). Data la matrice dati $X \in \mathbb{R}^{m \times (n+1)}$ ed $y \in \mathbb{R}^m$ il vettore di output, con m numero di esempi di training e n numero di feature, allora:

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

Dove X e y sono:

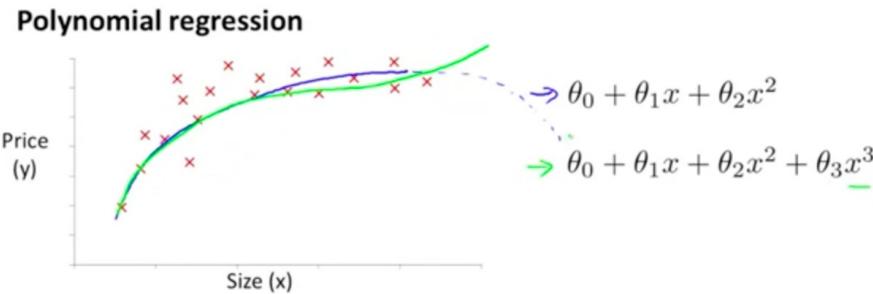


Figura 2.7.1: Regressione Polinomiale

$$X = \begin{bmatrix} (x^{(1)})^\top \\ (x^{(2)})^\top \\ \vdots \\ (x^{(m)})^\top \end{bmatrix}$$

$$y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Comparando in questo caso il gradiente discendente con le equazioni normali otteniamo che il primo è migliore in alcuni casi, le seconde in altri. Le caratteristiche sono qui riassunte:

Gradient Descent

- - Bisogna scegliere il learning rate α
- - Converge dopo molte iterazioni
- + Lavora bene anche con un grande numero di esempi

Normal Equations

- + Non c'è bisogno di scegliere il learning rate α
- + Non si effettuano iterazioni
- + Non è necessario effettuare feature scaling
- - Bisogna calcolare $(X^\top X)^{-1}$ che ha complessità $O(n)^3$
- - Lento se m , il numero di feature, è molto grande

Capitolo 3

Regressione Logistica

I problemi di classificazione, al contrario di quelli di regressione che prevedono un output continuo, restituiscono valori discreti: $y \in \{0, 1\}$ nel caso di classificazione binaria, $y \in \{1, \dots, n\}$ nel caso di classificazione multiclasse. Per il momento ci concentreremo sulla classificazione binaria. Non conviene applicare la regressione per risolvere i problemi di classificazione in quanto in quel caso avremmo che $h_\theta(x)$ può essere < 0 e > 1 ed inoltre dovremmo impostare un threshold arbitrario sull'ipotesi per distinguere quando restituire una classe e quando l'altra, ad esempio restituire 1 per $h_\theta(x) > 0.5$ e restituire 0 per $h_\theta(x) < 0.5$. Per affrontare problemi di classificazione si può usare una tecnica chiamata **Logistic Regression** (regressione logistica) che fornisce un output dell'ipotesi tale che $0 \leq h_\theta(x) \leq 1$.

La rappresentazione dell'ipotesi consiste nell'applicare una funzione sigmoidale g , anche detta funzione logistica, alla combinazione lineare già vista nella regressione lineare:

$$\begin{aligned} h_\theta(x) &= g(\theta^\top x) \\ g(z) &= \frac{1}{1 + e^{-z}} \\ h_\theta(x) &= \frac{1}{1 + e^{-\theta^\top x}} \end{aligned}$$

L'output dell'ipotesi h viene interpretato come la probabilità stimata che un determinato input x abbia come classe $y = 1$, mentre il suo complemento indica la probabilità stimata che $y = 0$:

$$h_\theta(x) = P(y = 1|x; \theta)$$

$$P(y = 0|x; \theta) = 1 - P(y = 1|x; \theta)$$

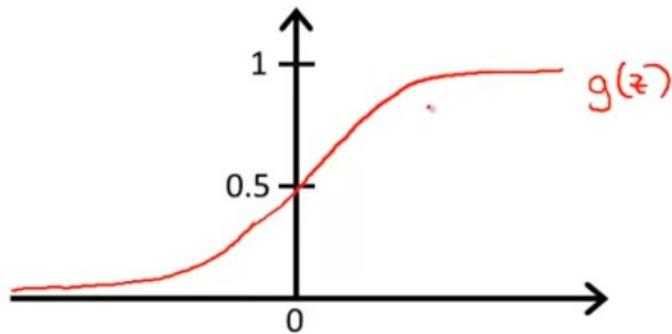


Figura 3.0.1: Funzione sigmoidale o logistica

Osservando l'andamento della funzione sigmoidale si può affermare che $y = 1$ quando $h_\theta(x) \geq 0.5$ il che accade quando $\theta^\top x \geq 0$ e che $y = 0$ quando $h_\theta(x) < 0.5$ il che accade quando $\theta^\top x < 0$.

I parametri dell'ipotesi descrivono una retta, quando l'input è da un lato della retta esso verrà classificato con una classe, se è dall'altro lato verrà classificato come appartenente all'altra classe. Questa retta è chiamata **decision boundary** ed è una proprietà dei parametri appresi e non dei dati di **training**. Aggiungendo elementi polinomiali all'interno dell'iposi h dentro la funzione sigmoidale si apprendono parametri che descrivono decision boundary non lineari.

La funzione costo J nel caso della regressione lineare era definita come $J(\theta) = \frac{1}{2n} \sum_{i=1}^n (h_\theta(x^{(i)}) - y^{(i)})^2$. Poiché in quel caso l'ipotesi h è una funzione lineare, allora J era convessa e dunque aveva un unico minimo globale che poteva essere individuato utilizzando il gradient descent. Nel caso della regressione logistica l'ipotesi h è una funzione sigmoidale, quindi non lineare, e dunque J è non convessa e dunque non abbiamo garanzie di convergenza del **gradient discendente**. Per questo motivo dobbiamo definire una funzione costo differente:

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

$$Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y = 1 \\ -\log(1 - h_\theta(x)) & y = 0 \end{cases}$$

Questo implica che che $y = 1$ per $h = 0$ il costo (l'output della funzione *Cost*) è infinito, mentre se $h = 1$ il costo è 0. Nel caso in cui $y = 0$, per $h = 0$,

Decision Boundary

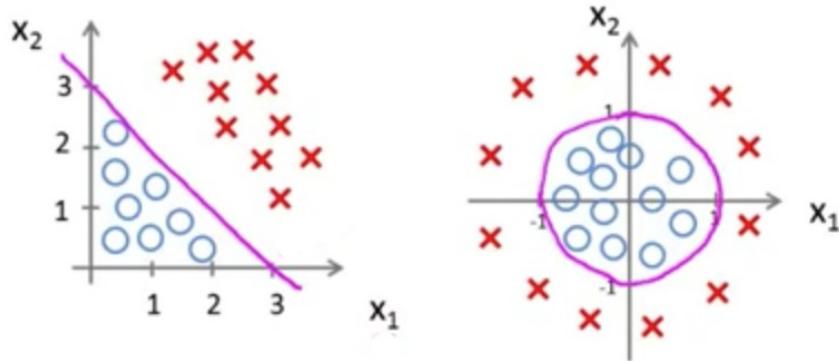


Figura 3.0.2: Decision Boundary

il costo è 0 mentre per $h = 1$, il costo è infinito. Poiché sappiamo che y è 1 o 0, possiamo scrivere la funzione *Cost* in un modo equivalente, ma più sintetico:

$$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

Quindi possiamo riscrivere in maniera completa la funzione costo J (il $-$ è stato portato fuori dalla sommatoria):

$$\begin{aligned} J(\theta) &= \frac{1}{m} \sum_{i=1}^{n+1m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)}) \\ &= -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] \end{aligned}$$

Ora non resta che minimizzare i parametri θ .

3.1 Discesa del Gradiente per la Regressione Logistica

Riscriviamo la funzione costo J e il passo di aggiornamento dei parametri in forma vettorializzata come fatto per l'ipotesi h .

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

$$\theta_k := \theta_k - \alpha \frac{\partial}{\partial \theta_k} J(\theta)$$

Calcolando le derivate parziali, si ottengono le funzioni per aggiornare i parametri:

$$\theta_k := \theta_k - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)}$$

Sono le stesse della regressione lineare. Ciò che cambia è ovviamente la definizione dell'ipotesi h che in questo caso è una funzione sigmoidale.

3.2 Algoritmi di ottimizzazione

Oltre al gradient descent è possibile utilizzare altri algoritmi di ottimizzazione, i più utilizzati sono:

- Conjugate Gradient
- BFGS
- LBFGS

Questi algoritmi hanno il vantaggio di non dover scegliere un learning rate α e di convergere più velocemente. Hanno in ogni caso lo svantaggio di essere più complessi del gradient descent.

Per poterli utilizzare nei software di algebra computazionale bisogna definire una funzione che descriva come calcolare J e le regole di update dei parametri θ . Fornendo un puntatore a questa funzione, alcuni parametri quali numero di iterazioni massimo e i parametri θ iniziali, questi algoritmi stimano i parametri ottimali. La funzione da utilizzare è FMINUNC.

```
// jVal = valore della funzione costo J
// gradient = regola di update per i parametri theta
function [jVal, gradient] = costFunction(theta)
    jval = ...;
    gradient = zeros(n, 1);
    gradient(1) = ...;
    gradient(...) = ...;
    gradient(n) = ...;

options = [ 'GrabObj', 'on', 'MaxIter', '100' ];
initialTheta = [1, 2];
// optTheta = valore ottimizzato dei parametri theta
// functionVal = valore di J all'ultima iterazione
// exitStatus = status di convergenza dell'algoritmo
[optTheta, functionVal, exitStatus] =
    fminunc(@costFunction, initialTheta, options);
```

3.3 Classificazione Multiclasse con la Regressione Logistica

La logistic regression, così come in generale tutti gli algoritmi di classificazione, può essere applicata a casi in cui sono presenti più di due classi. È sufficiente addestrare tanti classificatori quanto sono le classi in modo one-vs-all o one-vs-rest, ovvero decidendo una classe e impostando tutti gli elementi di quella classe come $y = 1$ e impostando tutti gli elementi delle altre classi con $y = 0$. Ripetendo questo procedimento per tutte le classi si ottengono tanti classificatori $h_{\theta}^{(i)}(x)$ quante sono le classi. Per assegnare una classe all'input x , è sufficiente calcolare la classe con probabilità stimata massima $\max_i h_{\theta}^{(i)}(x)$.

Graficamente il processo è mostrato in Figura 3.3.1.

One-vs-all (one-vs-rest):

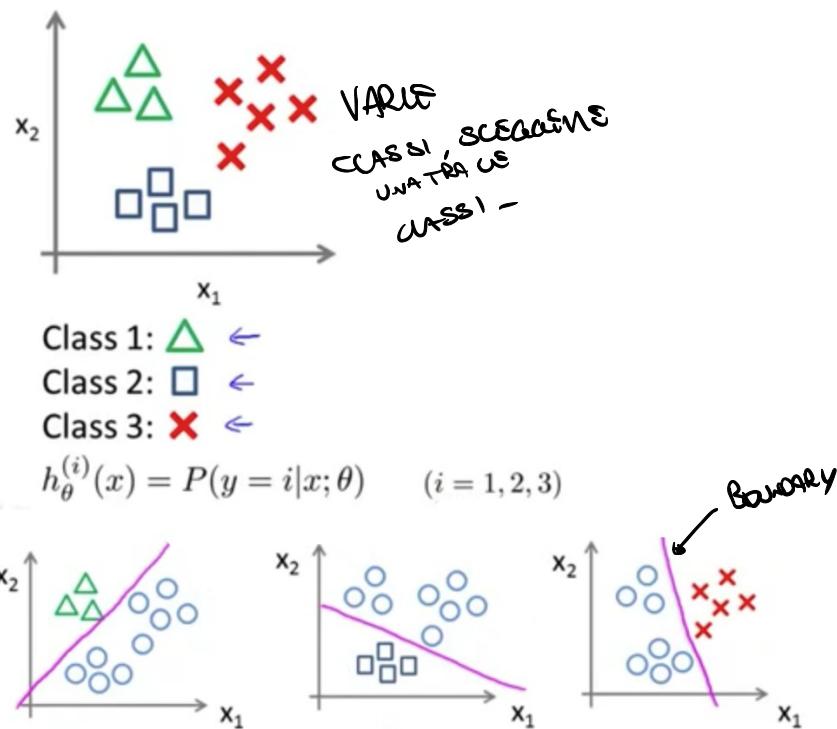


Figura 3.3.1: One-vs-All

Capitolo 4

Regolarizzazione

L'overfitting è un problema ricorrente negli algoritmi di machine learning e consiste nell'avere molte feature e un'ipotesi che apprende molto bene sugli esempi di training ($J(\theta) = \frac{1}{2(n+1)} \sum_{i=1}^{n+1} (h_\theta(x^{(i)}) - y^{(i)})^2 \approx 0$), ma fallisce nel generalizzare il problema e dunque ha scarse prestazioni su nuovi esempi di input. Ad esempio un polinomio di alto grado può classificare molto precisamente gli esempi di training ma essere troppo irregolare e altalenante per riuscire a classificare correttamente nuovi esempi.

Esistono alcune possibili soluzioni per mitigare questo problema:

- Ridurre il numero di feature, selezionando manualmente quali tenere e quali scartare oppure delegando questo compito ad un algoritmo di model selection. Così facendo si perde informazione potenzialmente utile.
- Utilizzare la regolarizzazione ovvero mantenere tutte le feature ma ridurne la magnitudine. Questo approccio funziona bene quando abbiamo molte feature e ciascuna influisce un po' nel predire y .

La regolarizzazione consiste nell'imporre, all'interno della funzione costo J , un costo ulteriore sulla grandezza di ciascun parametro. Questo ha l'effetto di ottenere ipotesi più semplici e meno soggette ad overfitting.

Per avere un'idea del funzionamento della regolarizzazione basta pensare che imporre ad un'ipotesi con termini cubici un alto costo per i termini a più alto grado permette di avere un'ipotesi vicina ad una di grado minore con un piccolo contributo da parte dei termini di grado maggiore.

4.1 Regolarizzazione nella Regressione Lineare

Nel caso di regressione lineare si ottiene:

$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^n \theta_i^2 \right]$$

Il parametro θ_0 non viene penalizzato poiché ha poco effetto sull'ipotesi finale. Il parametro λ è chiamato **parametro di regolarizzazione** e determina l'intensità della penalizzazione: se λ è molto vicino a 0, la penalizzazione non influisce, mentre se λ è troppo alto, si ottiene l'effetto che tutti i θ_i tendono a 0 e l'unico parametro a influire è θ_0 e conseguentemente l'ipotesi corrisponde a una linea orizzontale (un esempio di underfitting).

Le regole di update cambiano di conseguenza in quanto cambia la derivata parziale della funzione costo J :

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\begin{aligned} \theta_k &:= \theta_k - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)} - \frac{\lambda}{m} \theta_k \right] \\ &= \theta_k \left(1 - \alpha \frac{\lambda}{m} \right) - \left[\alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)} \right] \end{aligned}$$

Il valore $(1 - \alpha \frac{\lambda}{m})$ tende ad essere vicino a 0.99 quindi ad ogni update il valore di θ_k diminuisce e viene indirizzato verso la direzione del gradiente discendente. Il parametro θ_0 ha una regola a parte in quanto non viene penalizzato dalla regolarizzazione.

Nel caso volessimo utilizzare le normal equation, è sufficiente calcolare:

$$\theta = \left(X^\top X + \lambda \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} \right)^{-1} X^\top y$$

con $\lambda \geq 0$. Nel caso in cui $X^\top X$ non fosse stata invertibile, questa regolarizzazione la renderebbe invertibile.

4.2 Regolarizzazione nella Regressione Logistica

Nel caso di regressione logistica si ottiene:

$$J(\theta) = -\frac{1}{m} \left[\sum_{i=1}^m y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right] + \frac{\lambda}{2m} \sum_{i=1}^n \theta_i^2$$

Le regole di update cambiano di conseguenza in quanto cambia la derivata parziale della funzione costo J :

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\begin{aligned}
\theta_k &:= \theta_k - \alpha \left[\frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)} - \frac{\lambda}{m} \theta_k \right] \\
&= \theta_k \left(1 - \alpha \frac{\lambda}{m} \right) - \left[\alpha \frac{1}{m} \sum_{i=1}^m (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_k^{(i)} \right]
\end{aligned}$$

È possibile notare che le regole di aggiornamento dei parametri θ sono le stesse della regressione lineare, con l'unica differenza nella funzione h_θ .

Capitolo 5

Reti Neurali

In molti problemi del mondo reale la classificazione è non lineare e il numero delle feature è molto elevato. Questo comporta che per la regressione lineare e la regressione logistica, utilizzando termini polinomiali del 2° ordine (termini quadratici e prodotti tra altri termini) il numero dei termini è dell'ordine $O(n^2)$, mentre utilizzando i termini cubici si arriva all'ordine di $O(n^3)$. In problemi complessi come la computer vision si raggiungono facilmente milioni di feature, il che rende gli algoritmi finora studiati molto pesanti in termini di tempi di esecuzione. Per questo motivo verrano ora descritte le **Neural Networks** (reti neurali) che sono un metodo di rappresentazione con una famiglia di algoritmi ad essa associati che permettono di realizzare classificatori non lineari più semplici ed efficienti.

Dal punto di vista storico, le reti neurali nascono per cercare di imitare i neuroni del cervello. Sono state molto utilizzate negli anni '80 e primi '90, hanno perso slancio dal punto di vista accademico negli anni '90 mentre oggi, grazie principalmente al superamento di alcuni limiti hardware, sono tornate ad essere utilizzate e sono la soluzione a stato dell'arte per molti problemi.

L'idea di fondo è che ci sia un unico "algoritmo" di apprendimento che possa risolvere tutti i problemi, esattamente come fa il cervello umano. Esso infatti si adatta agli stimoli che riceve e una qualunque parte del cervello umano può imparare a riconoscere segnali da parte di un qualsivoglia organo sensore (neural re-wiring experiments).

5.1 Rappresentazione di una Rete Neurale

Semplificando e astraendo sulla struttura dei neuroni biologici, essi sono strutturati come unità di calcolo con una serie di connessioni in input chiamate dendriti ed una connessione in output chiamata asseone. Ciascun neurone artificiale dunque consiste di input x , una serie di parametri o pesi θ e di un unico output $h_\theta(x)$ che corrisponde a una funzione sigmoidale detta "di attivazione" tale che

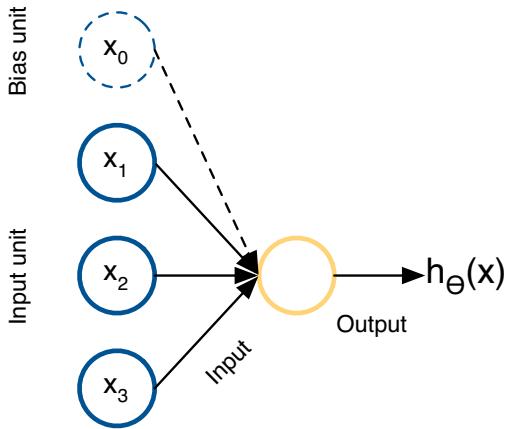


Figura 5.1.1: Modello di neurone: unità logistica

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$, come mostrato in Figura 5.1.1. L'input x_0 è detto "bias unit", ha sempre valore 1 e non viene sempre disegnato.

Una rete neurale è composta da più livelli di neuroni: il primo livello è composto da unità di input, dal secondo livello in poi ci sono i livelli nascosti, gli "hidden layer", mentre l'ultimo livello è il livello di output, come mostrato in Figura 5.1.2.

Con $a_i^{(j)}$ si denota l'i-esimo neurone del j-esimo livello, mentre con $\Theta^{(j)}$ si indica la matrice di parametri o pesi che mappa il livello j di neuroni al livello $j + 1$. Ciò che viene calcolato da ciascun nodo è:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right)$$

Ipotizzando che il livello di output sia il 3° e non ci siano altri livelli nascosti:

$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right)$$

Se la rete neurale ha s_j nodi nel livello j , s_{j+1} nodi nel livello $j+1$, allora $\Theta^{(j)}$ sarà di dimensioni $s_{j+1} \times (s_j + 1)$. Ogni singolo livello della rete effettua un passo computazionale equivalente alla regressione logistica permettendo di calcolare complesse funzioni non lineari sui valori di input. Inoltre le reti possono avere diverse architetture, ovvero un numero diverso di livelli ed un numero diverso di neuroni per ciascun livello, gli unici vincoli sono il numero di nodi nel livello di input e l'unico nodo nel livello di output.

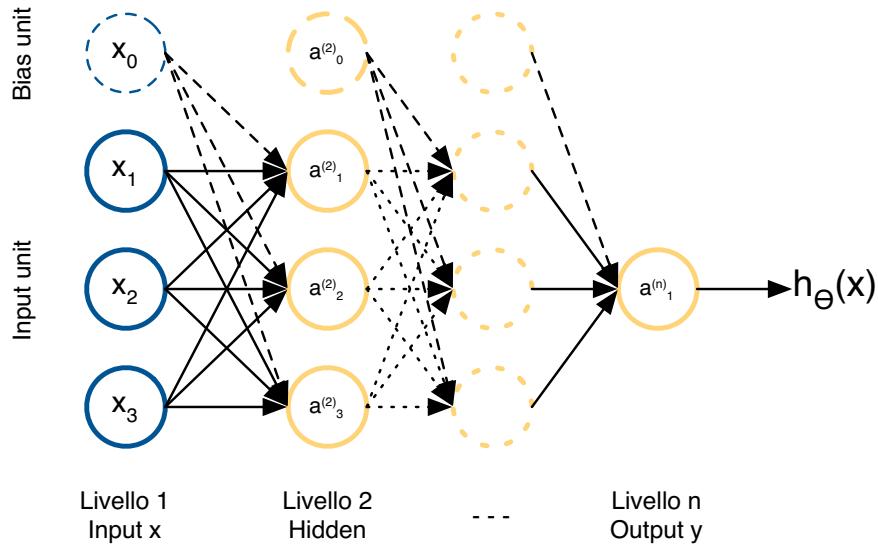


Figura 5.1.2: Modello di rete neurale

Per ottenere una rappresentazione vettorializzata di una rete neurale basta considerare come vettori x , y , a e z , far corrispondere $x = a^{(1)}$ e calcolare $z^{(i+1)} = \Theta^{(i)}a^{(i)}$ e $a^{(i+1)} = g(z^{(i+1)})$ per tutti i livelli i , ricordandosi di aggiungere $a_0^{(i)} = 1$ per ciascun vettore di ciascun livello.

5.2 Esempio: funzioni logiche

Un esempio concreto delle capacità di una rete neurale di calcolare funzioni non lineari è la funzione **XNOR**. [IMMAGINE] La funzione ritorna 1 quando ha come input due 0 e due 1, ritorna 0 altrimenti. Per poterla calcolare definiamo 4 reti neurali, una per calcolare la funzione $x_1 \text{ AND } x_2$, un'altra per la funzione $x_1 \text{ OR } x_2$, un'altra per la funzione **NOT** x_1 e una per calcolare $(\text{NOT } x_1) \text{ AND } (\text{NOT } x_2)$.

Premessa: la funzione sigmoidale che qui chiamiamo g tende ad 1 per $x \rightarrow \infty$ e tende a 0 per $x \rightarrow -\infty$, ma ha valore di output 0.99 per $x = 4.6$ e specularmente ha valore di output 0.01 per $x = -4.6$. Di conseguenza approssimeremo a 1 e a 0 rispettivamente l'output di g per valori maggiori di 4.6 e inferiori a -4.6.

Per la funzione **AND** basta utilizzare due nodi di input, un nodo di bias e un unico nodo di output. Assegnando pesi in questo modo (il primo peso è assegnato al nodo di bias):

$$\Theta_{AND}^{(1)} = [-30 \quad +20 \quad +20]$$

Così facendo la funzione calcolata dal neurone di output è la seguente: $h_{\Theta_{AND}}(x) = g(-30 + 20x_1 + 20x_2)$. Questa funzione calcola, rispettivamente all'input i seguenti valori:

x_1	x_2	$h_{\Theta_{AND}}(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Circa la funzione **OR** i pesi sono assegnati nel seguente modo:

$$\Theta_{OR}^{(1)} = [-10 \quad +20 \quad +20]$$

La funzione calcolata dal neurone di output è la seguente: $h_{\Theta_{OR}}(x) = g(-10 + 20x_1 + 20x_2)$. Questa funzione calcola, rispettivamente all'input i seguenti valori:

x_1	x_2	$h_{\Theta_{OR}}(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

Circa la funzione **NOT** i pesi sono assegnati nel seguente modo:

$$\Theta_{NOT}^{(1)} = [+10 \quad -20]$$

La funzione calcolata dal neurone di output è la seguente: $h_{\Theta_{NOT}}(x) = g(10 - 20x_1)$. Questa funzione calcola, rispettivamente all'input i seguenti valori:

x_1	$h_{\Theta_{NOT}}(x)$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

Seguendo quanto fatto per la funzione **NOT** è facile definire la funzione **(NOT) AND (NOT)** assegnando i pesi nel seguente modo:

$$\Theta_{NOTANDNOT}^{(1)} = [+10 \quad -20 \quad -20]$$

La funzione calcolata dal neurone di output è la seguente: $h_{\Theta_{NOTANDNOT}}(x) = g(10 - 20x_1 - 20x_2)$. Questa funzione calcola, rispettivamente all'input i seguenti valori:

x_1	x_2	$h_{\Theta_{NOTANDNOT}}(x)$
0	0	$g(10) \approx 1$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(-30) \approx 0$

Unendo il tutto in un'unica rete neurale è possibile calcolare la funzione **XNOR**. La rete finale è mostrata in Figura 5.2.1.

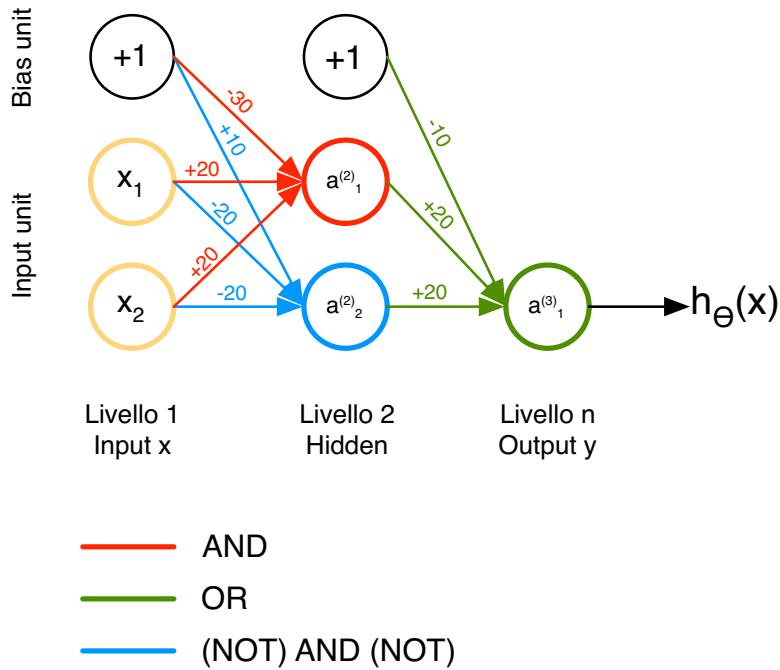


Figura 5.2.1: Rete neurale **XNOR**

I pesi sono assegnati nel seguente modo:

$$\Theta_{XNOR}^{(1)} = \begin{bmatrix} -30 & +20 & +20 \\ +10 & -20 & -20 \end{bmatrix}$$

$$\Theta_{XNOR}^{(2)} = \begin{bmatrix} -10 & +20 & +20 \end{bmatrix}$$

La funzione calcolata dal neurone di output è la seguente: $h_{\Theta_{XNOR}}(x) = g(-10 + 20a_1^{(2)} + 20a_2^{(2)})$. Questa funzione calcola, rispettivamente all'input i seguenti valori:

x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_{\Theta_{XNOR}}(x)$
0	0	0	1	$g(10) \approx 1$
0	1	0	0	$g(-10) \approx 0$
1	0	0	0	$g(-10) \approx 0$
1	1	1	0	$g(10) \approx 1$

In pratica aggiungendo livelli di neuroni è possibile calcolare funzioni non lineari via via più complesse.

5.3 Classificazione Multiclasse con le Reti Neurali

La classificazione multiclasse con le reti neurali si può ottenere in maniera molto semplice aggiungendo alla rete neurale tanti nodi di output quante sono le classi ottenendo che $h_\Theta(x) \in \mathbb{R}^n$ dove n è il numero di classi o di nodi del livello di output della rete. Rappresentando l'output y come un vettore colonna tale che $y \in \mathbb{R}^n$ dove n è il numero di classi è possibile esprimere l'output corretto per la prima classe come $y = [1 \ 0 \ \dots \ 0]^\top$, l'output per la seconda classe $y = [0 \ 1 \ 0 \ \dots \ 0]^\top$ e così via per tutte le classi. Quello che si vorrebbe ottenere dalla rete è un vettore quanto più possibile simile ai vettori che rappresentano le diverse classi: $h_\Theta(x) \approx [1 \ 0 \ \dots \ 0]^\top$ quanto x appartiene alla prima classe, $h_\Theta(x) \approx [0 \ 1 \ 0 \ \dots \ 0]^\top$ quando x appartiene alla seconda classe e così via.

5.4 Funzione Costo delle Reti Neurali

Denotiamo con L il numero di livelli nella rete neurale, con s_l il numero di unità o nodi per ogni livello l senza contare le unità bias e con K il numero di classi del problema di classificazione. Inoltre nel caso multiclasse, il caso più generico, $h_\Theta(x) \in \mathbb{R}^K$ e $h_\Theta(x)_k$ denota l'output del k -esimo nodo del livello di output.

La funzione costo da minimizzare è analoga a quella della regressione logistica, sommando però i costi di tutti i nodi di output e regolarizzando per tutti i pesi di tutte le matrici Θ di tutti i livelli:

$$J(\theta) = -\frac{1}{n+1} \left[\sum_{i=1}^{n+1} \sum_{k=1}^K y_k^{(i)} \log(h_\Theta(x^{(i)})_k) + (1 - y_k^{(i)}) \log(1 - h_\Theta(x^{(i)})_k) \right] + \frac{\lambda}{2(n+1)} \sum_{l=1}^{L-1} \sum_{i=0}^{s_l} \sum_{j=0}^{s_{l+1}} (\Theta_{ji}^{(l)})^2$$

Anche in questo caso nella regolarizzazione non vengono considerati i parametri del posto 0 in quanto corrispondono alle unità di bias.

5.5 Algoritmo di Backpropagation

Per poter minimizzare la funzione costo abbiamo bisogno delle funzioni che calcolino le derivate parziali per permettere ad un algoritmo di ottimizzazione di poter auspicabilmente convergere verso un minimo. Nel caso delle reti neurali si possono calcolare le derivate attraverso un algoritmo chiamato di **Backpropagation**.

Esso consiste, per ciascun esempio di input, nel calcolare la Forward Propagation, ovvero tutti gli $a_j^{(l)}$, calcolare per ciascun livello l'errore dei nodi rispetto all'output atteso e accumulare il prodotto di questi valori in una matrice che conterrà proprio le derivate parziali.

Più formalmente, prendendo in considerazione un solo esempio, effettuiamo la forward propagation:

$$\begin{aligned}
 a^{(1)} &= x \\
 z^{(2)} &= \Theta^{(1)} a^{(1)} \\
 a^{(2)} &= g(z^{(2)}) \\
 &\vdots \\
 z^{(L)} &= \Theta^{(L-1)} a^{(L-1)} \\
 a^{(L)} &= h_{\Theta}(x) = g(z^{(L)})
 \end{aligned}$$

Dopodichè calcoliamo i $\delta_j^{(l)}$, ovvero l'errore del j-esimo nodo del l-esimo livello come:

$$\begin{aligned}
 \delta_j^{(L)} &= a_j^{(L)} - y_j \\
 \delta_j^{(L-1)} &= (\Theta^{(L-1)})^{\top} \delta_j^L \cdot * g'(z^{(L-1)}) \\
 &\vdots \\
 \delta_j^{(2)} &= (\Theta^{(2)})^{\top} \delta_j^3 \cdot * g'(z^{(2)})
 \end{aligned}$$

dove $\cdot *$ è il prodotto puntuale e $g'(z^{(l)})$ è la derivata di g in $z^{(l)}$, può essere calcolata come $g'(z^{(l)}) = a^{(l)} \cdot * (1 - a^{(l)})$. Il valore $\delta^{(1)}$ non viene calcolato in quanto non avrebbe senso calcolare l'errore dell'input.

È possibile dimostrare che

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)}$$

se si ignora il fattore di regolarizzazione λ , quindi in questo modo si possono calcolare le derivate parziali da fornire in input ad un algoritmo di ottimizzazione.

Algoritmo 5.1 Algoritmo di Backpropagation

Training Set = $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

- Set $\Delta_{ij}^{(l)} := 0$ for all i, j, l
 - For $i = 1$ to m
 - Set $a^{(1)} := x^{(i)}$
 - Perform Forward Propagation to compute $a^{(2)}, a^{(3)}, \dots, a^{(L)}$
 - Set $\delta^{(L)} := a^{(L)} - y^{(i)}$
 - Perform Back Propagation to compute $\delta^{(L-1)}, \dots, \delta^{(2)}$
 - Set $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
 - $D_{ij}^{(l)} := \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$
 - $D_{ij}^{(l)} := \Delta_{ij}^{(l)}$ if $j = 0$
-

Complessivamente, per tenere conto di tutti gli esempi di input, l'algoritmo di backpropagation calcola le derivate nel modo descritto nell'Algoritmo 5.1 in modo tale che;

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

5.6 Parametri di una Rete Neurale

Inizializzando i parametri delle matrici Θ di una rete neurale a 0 assicuriamo che i pesi che partono da un nodo di un certo livello vero i nodi del livello successivo saranno sempre uguali, perché saranno uguali le derivate. In questo modo la rete non si modificherà in modo da calcolare funzioni utili, ma è come se ogni nodo calcolasse la stessa funzione di tutti gli altri.

Per ovviare a questo problema, tutte le matrici devono essere inizializzate singolarmente in un range di valori $[-\epsilon, +\epsilon]$.

Gli altri parametri da decidere sono il numero dei livelli e il numero dei nodi di ciascun livello. Per il livello di input ed output la scelta è obbligata: si utilizza il numero di feature per i primi e il numero di classi per i secondi. Circa il numero di hidden layer, anche un solo livello è sufficiente, ma un maggior numero permettono migliori prestazioni, mentre il numero di nodi nei livelli hidden dovrebbero essere sempre lo stesso e da 1 a 4 volte maggiori del numero di nodi del livello di input. Ovviamente più livelli hidden e più nodi nei livelli hidden si decide di utilizzare, più pesante sarà il calcolo dal punto di vista computazionale.

Queste semplici regole si sono rivelate efficaci sperimentalmente.

Capitolo 6

Applicazione e Valutazione

Per migliorare le prestazioni di un sistema di Machine Learning è possibile fare diverse operazioni:

- Aumentare il numero di esempi di addestramento;
- Utilizzare un insieme ridotto di feature;
- Utilizzare un numero maggiore di feature;
- Aggiungere feature polinomiali;
- Aumentare il fattore di regolarizzazione;
- Ridurre il fattore di regolarizzazione.

Alcune di queste operazioni possono richiedere anche mesi di lavoro in un contesto reale, e potrebbero comunque non portare ad un risultato utile, dunque è importante avere un criterio per decidere quale operazione effettuare.

6.1 Diagnosi (Train and Test)

Un buon metodo per rendersi conto di cosa attualmente funziona o meno in un algoritmo di apprendimento e per quindi rendersi conto di dove intervenire è la diagnosi. Essa consiste in un test che si può effettuare per avere un'idea più chiara di cosa funziona e cosa non funziona nel sistema di machine learning e di quale sia la via migliore per migliorarne le prestazioni. L'implementazione delle tecniche di diagnosi è lunga, ma il tempo così impiegato è ben speso e permette di guadagnare molto più tempo in futuro.

Per diagnosticare un'ipotesi h è possibile dividere il training set in porzioni (ad esempio 70% - 30%) ed utilizzare la principale per il training e la rimanente per il test. Nel caso di problemi di regressione si può calcolare la differenza quadratica media del valore calcolato da h rispetto al vero valore y , mentre nel caso della classificazione si può calcolare una media del numero di errori di classificazione commessi da h .

CROSS-VALIDATION

6.2 Model Selection

DIVISIONE DEL TRAINING SET

Alcuni algoritmi, ad esempio la regressione lineare, possono ottenere prestazioni molto differenti dipendentemente da alcuni parametri come ad esempio il grado del polinomio della funzione ipotesi h . Per poter determinare il miglior parametro si suddivide il training set in tre parti: il nuovo training set, il cross validation set e il test set. Il primo è utilizzato per apprendere il modello, il secondo per determinare il valore migliore del parametro d ed il terzo per controllare quanto effettivamente l'ipotesi appresa generalizzi su esempi mai visti prima. L'errore del modello rispetto al training set è denotato da $J_{train}(\Theta)$, l'errore del modello utilizzando il miglior parametro d rispetto al cross validation set è denotato da $J_{cv}(\Theta)$ e l'errore dell'ipotesi finale h rispetto al test set è denotato da $J_{test}(\Theta)$.

6.3 Bias e Varianza

Si parla di alto bias quando siamo in caso di underfitting, l'ipotesi h è di grado troppo basso e non si adeguia bene rispetto ai dati del training set. Si parla di alta varianza quando invece c'è overfitting, l'ipotesi h è di grado troppo alto, corrisponde in modo molto preciso ai dati di training ma non riesce a generalizzare, commettendo molti errori sul cross validation set o sul test set.

In Figura 6.3.1 è mostrato l'andamento dell'errore rispetto al test set e rispetto al cross validation set. Sulla sinistra si hanno $J_{train}(\Theta)$ e $J_{cv}(\Theta)$ alti e si è in caso di underfitting, sulla destra si ha $J_{train}(\Theta)$ basso e $J_{cv}(\Theta)$ alto e si è in caso di overfitting. Calcolare i valori e vedere se sono vicini o distanti ci può fare un'idea del comportamento che sta avendo l'algoritmo e può indirizzarci verso un aumento o una diminuzione di grado del polinomio o più in generale, verso un intervento di modifica dei parametri liberi d del modello ce rappresenta l'ipotesi h .

Lo stesso discorso vale anche per il fattore di regolarizzazione λ , che può essere selezionato calcolando l'errore sul cross validations set. In Figura 6.3.2 è mostrato l'andamento dell'errore al variare di λ .

6.4 Learning Curves

Le Learning Curves sono uno strumento utile per accorgersi quando un algoritmo soffre bias o soffre per eccessiva varianza. L'idea è di mostrare l'andamento dell'errore di $J_{train}(\Theta)$ e $J_{cv}(\Theta)$ rispetto al numero di esempi di training m utilizzati. Se i due errori hanno un gap piccolo, come mostrato in Figura 6.4.1, vuol dire che l'algoritmo soffre di bias e aggiungere altri esempi di training non influisce sulle prestazioni, mentre se il gap è consistente, come in Figura 6.4.2, l'algoritmo soffre di alta varianza e quindi aggiungere esempi di training può aiutare a risolvere il problema.

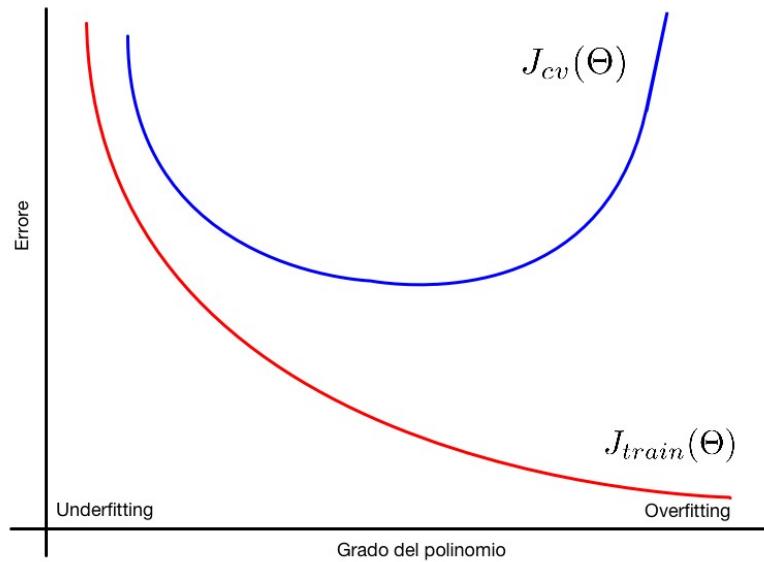


Figura 6.3.1: Errore rispetto al test set e cross validation set al variare del grado del polinomio

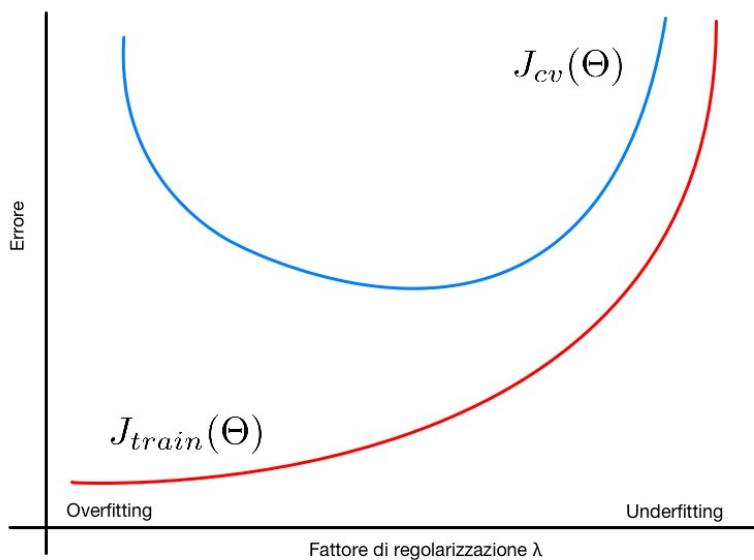


Figura 6.3.2: Errore rispetto al test set e cross validation set rispetto al fattore di regolarizzazione λ

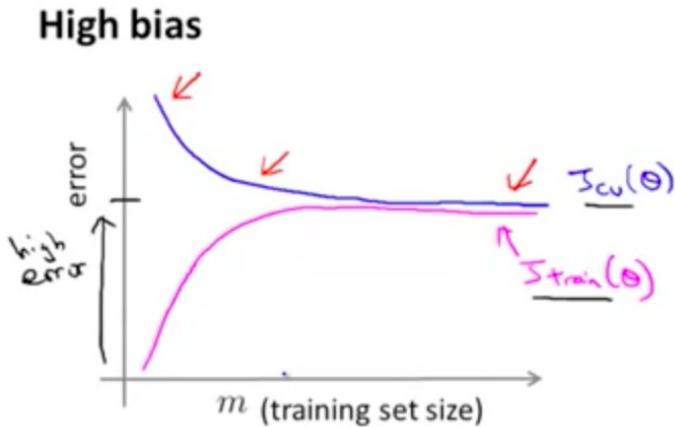


Figura 6.4.1: Learning Curve con alto Bias

6.5 Consigli pratici

Riprendendo le operazioni precedentemente elencate, possiamo ora affermare in quali casi sono utili:

- Aumentare il numero di esempi di addestramento → compensa alta varianza
- Utilizzare un insieme ridotto di feature → compensa alta varianza
- Utilizzare un numero maggiore di feature → compensa alto bias
- Aggiungere feature polinomiali → compensa alto bias
- Aumentare il fattore di regolarizzazione → compensa alto bias
- Ridurre il fattore di regolarizzazione → compensa alta varianza

Nel caso di reti neurali, reti con pochi nodi e pochi livelli hidden sono più pronate all'underfitting, mentre reti grandi e con molti livelli sono più pronate all'overfitting, ma sperimentalmente si nota come sia meglio mitigare l'overfitting di una rete grande aumentando il parametro di regolarizzazione λ che ridurre le dimensioni della rete.

6.6 Progettazione di un sistema di Machine Learning

Vengono qui riassunte alcune idee e linee guida per la progettazione di un sistema di machine learning.

High variance

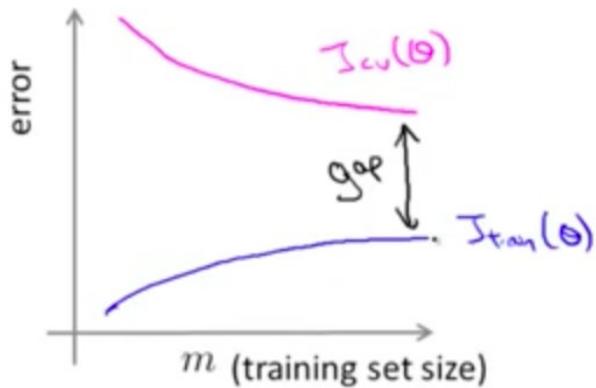


Figura 6.4.2: Learning Curve con alta varianza

La prima cosa da fare è identificare le feature da utilizzare: conviene fare una lista quanto più possibile esaustiva delle possibilità che si hanno a disposizione, analizzarle singolarmente ed in fine decidere quali utilizzare in base alla praticità. Non esiste in effetti un criterio per assicurarsi che le feature scelte saranno quelle che daranno risultati migliori, ma l'analisi degli errori può aiutare ad avere un'idea più chiara.

Per prima cosa bisogna implementare una versione "quick and dirty" di un semplice algoritmo di apprendimento e testarla sui dati di cross validation. Analizzando le learning curve che si ottengono da questo esperimento è possibile avere delle indicazioni sull'utilità di aggiungere altre feature o aumentare il numero di esempi di training o altri interventi. Inoltre analizzando singolarmente i casi in cui l'algoritmo commette degli errori si possono identificare trend e cause comuni di errore, rendendo poi semplice aggiungere feature ad hoc in modo da permettere all'algoritmo di evitare gli errori commessi precedentemente.

Inoltre un'analisi numerica dei risultati degli esperimenti, con metriche appositamente studiate, permette di avere un'idea chiara se una modifica effettuata porta un vantaggio o peggiora la situazione (ad esempio se utilizzare o meno lo stemming dei termini nella spam detection).

!!
!!

6.7 Metriche di valutazione

Tra le metriche per analizzare numericamente gli esperimenti, una possibile metrica è l'**Accuracy** (accuratezza):

$$\text{accuracy} = \frac{\# \text{Classificati Correttamente}}{\# \text{Claddificati}}$$

Solitamente l'accuratezza è un indicatore affidabile, ma nel caso di classi molto ristrette, "skew" in gergo, l'accuratezza non è un buon indicatore. Se ad esempio solo lo 0.5% del test set appartiene alla classe 1 in un problema di classificazione binario, predire sempre la classe 0 avrebbe accuratezza = 99.5%.

Per questo motivo si adoperano due misure di **Precision** (precisione) e **Recall** (richiamo).

		Classe Reale	
		1	0
Classe predetta	1	Vero Positivo	Falso Positivo
	0	Falso Negativo	Vero Negativo

Nella matrice di contingenza mostrata si evidenziano le possibili predizioni dell'algoritmo rispetto alle reali classi degli esempi. Possiamo definire la precisione come la frazione di esempi appartenenti effettivamente alla classe 1 tra tutti quelli che l'algoritmo ha classificato come appartenenti alla classe 1, ovvero la prima cella sulla prima riga:

$$\text{precisione} = \frac{\#VeroPositivo}{\#VeroPositivo + \#FalsoPositivo} = \frac{\#VeroPositivo}{\#PredettiPositivi}$$

Il richiamo invece è la frazione di esempi classificati dall'algoritmo come appartenenti alla classe 1 tra tutti quelli che effettivamente appartengono alla classe 1, ovvero la prima cella sulla prima colonna:

$$\text{richiamo} = \frac{\#VeroPositivo}{\#VeroPositivo + \#FalsoNegativo} = \frac{\#VeroPositivo}{\#RealiPositivi}$$

L'andamento di queste due misure è spesso discordante, al crescere dell'una cala l'altra come mostrato in Figura 6.7.1.

Di conseguenza è utile avere un valore unico che sintetizzi i due valori e ci dia una misura complessiva del comportamento dell'algoritmo. La media dei due valori non è adeguata in quanto non tiene in considerazione il fatto che vorremo che entrambi i valori fossero alti, mentre la media darebbe lo stesso punteggio sia ad un algoritmo con precisione di 0.9 e richiamo di 0.1 sia ad uno con precisione 0.4 e richiamo 0.6 quando in effetti il secondo sarebbe molto più utile. Per questo motivo esiste la **F-measure**, una misura che riassume i valori facendone una media armonica pesata:

$$F_\beta = \frac{(1 + \beta^2)PR}{\beta^2P + R}$$

Ponendo $\beta = 1$ si ottiene la misura più utilizzata che bilancia equamente precisione e richiamo, ovvero la **F₁-measure**:

$$F_1 = \frac{2PR}{P + R}$$

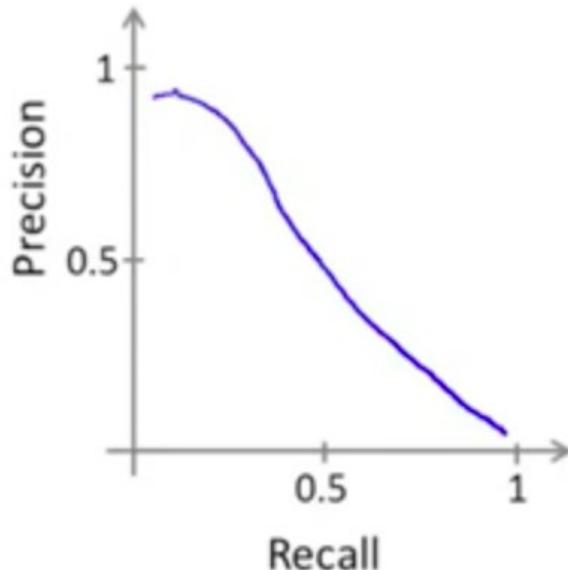


Figura 6.7.1: Richiamo vs. Precisione

6.8 Dati e Algoritmi

Abbiamo precedentemente visto che avere più dati non sempre porta ad avere prestazioni migliori negli algoritmi di machine learning, ma ci sono delle circostanze in cui un maggior numero di dati aiuta molto.

In particolare bisogna partire da un presupposto: dato un esempio di input x con tutte le sue feature, un essere umano esperto del dominio è capace di classificarlo o in generale di assegnare un output y ? Se la risposta è affermativa vuol dire che disponiamo delle informazioni sufficienti su ciascun esempio per classificarlo correttamente. Ad esempio negli esercizi “fill the gap” conoscere le parole attorno al gap è sufficiente per un essere umano per completare l'esercizio, mentre conoscere solamente la dimensione in metri quadrati non è sufficiente neanche per il miglior agente immobiliare per fare una valutazione del prezzo di una casa.

Dopotutto conviene scegliere un algoritmo con molti parametri. Questo garantisce un basso bias dell'ipotesi finale. In fine bisogna dare in pasto un grandissimo numero di esempi all'algoritmo, garantendo dunque che ci sia una bassa varianza. L'algoritmo con basso bias ci garantisce un basso tasso di errore all'interno degli esempi di training. L'alto numero di esempi di training ci garantisce un'alta corrispondenza tra l'errore nel training set e quello nel test set, per cui alla fine si può avere fiducia che il sistema di machine learning abbia un basso errore nel test set.

Capitolo 7

Support Vector Machines

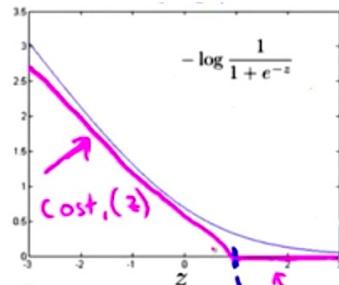
7.1 Funzione Costo ed Ipotesi delle Support Vector Machines

Un punto di partenza per studiare le **Support Vector Machines (SVM)**, un potente algoritmo di apprendimento, è la funzione *Cost* della regressione logistica:

$$Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & y = 1 \\ -\log(1 - h_\theta(x)) & y = 0 \end{cases}$$

Sostituendo l'andamento logaritmico con due funzioni $Cost_1$ e $Cost_0$ con un andamento lineare mostrato in Figura , si ottiene una nuova funzione costo J alternativa a quella della regressione logistica definita come segue:

If $y = 1$ (want $\theta^T x \gg 0$):



If $y = 0$ (want $\theta^T x \ll 0$):

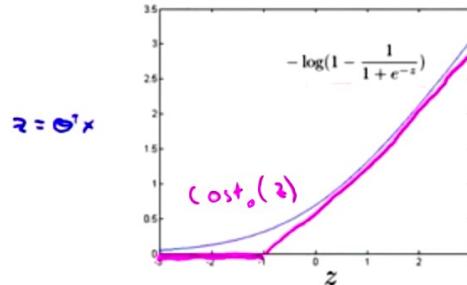


Figura 7.1.1: $Cost_1$ e $Cost_0$

$$J(\theta) = -\frac{1}{n+1} \left[\sum_{i=1}^{n+1} y^{(i)} Cost_1(\theta^\top x^{(i)}) + (1 - y^{(i)}) Cost_0(\theta^\top x^{(i)}) \right]$$

Per scrivere la funzione costo J delle SVM bisogna fare altre due assunzioni, la prima è di notazione: finora abbiamo visto uno schema ricorrente quando si utilizza la regolarizzazione del tipo $A + \lambda B$ dove A è la funzione costo originale, λ è il fattore di regolarizzazione e B è la parte di regolarizzazione vera e propria. Nel caso delle SVM si usa la notazione $CA + B$ dove C è un fattore che, al contrario di λ , indica l'intensità della funzione costo e non del fattore di normalizzazione. Inoltre poiché nella formula della funzione costo della regressione logistica regolarizzata compare come denominatore $n+1$, lo si può eliminare. Questo ci fa giungere alla funzione costo per le SVM:

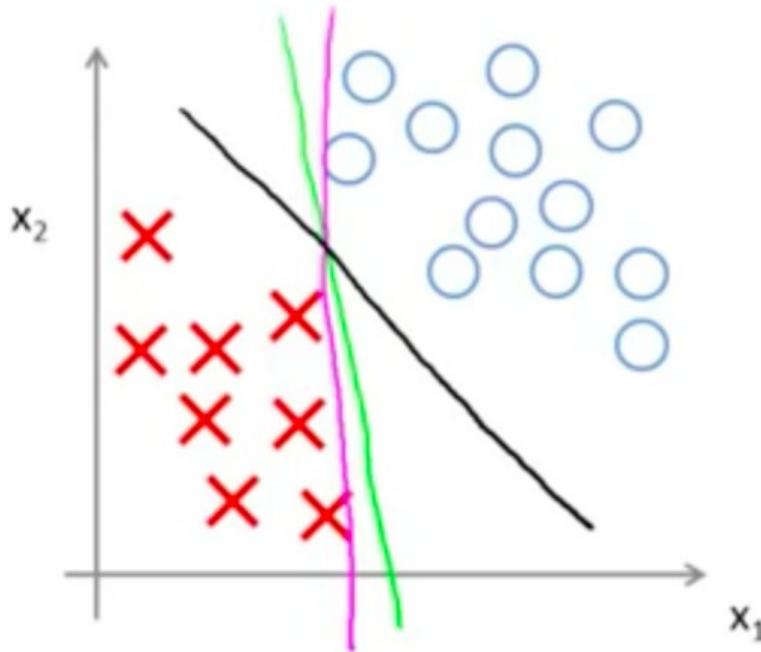
$$J(\theta) = C \left[\sum_{i=1}^{n+1} y^{(i)} Cost_1(\theta^\top x^{(i)}) + (1 - y^{(i)}) Cost_0(\theta^\top x^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

L'ipotesi h è così definita:

$$h_\theta(x) = \begin{cases} 1 & \text{if } \theta^\top x \geq 0 \\ 0 & \text{if } \theta^\top x < 0 \end{cases}$$

7.2 Large Margin Classifier

L'ipotesi e la funzione costo permettono alle SVM di essere un **Large Margin Classifier**, ovvero un classificatore che pone il decision boundary tra le classi in modo da massimizzare la distanza dagli esempi di training.



Il decision boundary in nero nella figura garantisce un margine più ampio dagli esempi rispetto a quello verde e a quello magenta. Questa proprietà è valida per C molto grandi, mentre con C meno grandi non è garantita la migliore distanza, ma l'algoritmo è meno soggetto agli outliers.

La motivazione di questa proprietà è che con C molto grandi l'ottimizzazione favorirà situazioni in cui $A = 0$. Per via dell'andamento delle funzioni $Cost_1$ e $Cost_0$, ciò accadrà quando $\theta^\top x^{(i)} \geq 1$ se $y^{(i)} = 1$ e quando $\theta^\top x^{(i)} \leq -1$ se $y^{(i)} = 0$. Ciò trasforma il problema di minimizzazione in un problema di minimizzazione vincolata:

$$\min_{\theta} \frac{1}{2} \sum_{j=1}^n \theta_j^2$$

con vincoli:

$$\begin{cases} \theta^\top x^{(i)} \geq 1 & if \quad y^{(i)} = 1 \\ \theta^\top x^{(i)} \leq -1 & if \quad y^{(i)} = 0 \end{cases}$$

Poiché $\theta^\top x^{(i)}$ può essere interpretato anche come $p^{(i)} \|\theta\|$, ovvero la lunghezza della proiezione di x su θ per la norma di θ , è facile rendersi conto che

una decision boundary (che è perpendicolare al vettore θ) vicino agli esempi di training farà sì che le proiezioni $p^{(i)}$ siano molto piccole e di conseguenza, per rispettare i due vincoli i valori di θ dovranno essere molto grandi, contravvenendo all'obiettivo da minimizzare. Conseguentemente il decision boundary che verrà preselto sarà quello che garantirà i valori di θ più piccoli, quindi le proiezioni $p^{(i)}$ più piccole e quindi descriverà un boundary quanto più possibile distante dagli esempi di training. Quindi l'SVM sotto la condizione di C molto grande è un large margin classifier, come mostrato in Figura 7.2.1.

7.3 Kernels

Quello che si vuole ottenere è un decision boundary non lineare in modo da poter apprendere funzioni complesse e classificare in modo efficace. Un modo è utilizzare i **Kernels**.

Anziché utilizzare le feature x_i , definiamo dei landmark $l^{(i)}$ e delle nuove feature f_i . I landmark l_i sono dei punti nello spazio (vedremo dopo come selezionarli) e le feature f_i sono ottenute tramite funzioni di similarità tra il vettore dell'esempio x e l' i -esimo landmark $l^{(i)}$. La funzione di similarità che si decide di utilizzare prende il nome di **Kernel**. Ad esempio utilizzando un kernel gaussiano abbiamo che:

$$f_i = sim(x, l^{(i)}) = \exp\left(-\frac{\|x - l^{(i)}\|^2}{2\sigma^2}\right)$$

Con questo kernel, se $x \approx l^{(i)}$ allora $f_i \approx 1$, mentre se x è lontano da $l^{(i)}$, allora $f_i \approx 0$.

Utilizzando gli f_i come feature al posto degli x_i abbiamo che, ad esempio, vogliamo predire $y = 1$ nel caso in cui $\theta_0 + \theta_1 f_1 + \dots + \theta_n f_n \geq 0$. Ipotizzando di aver ottenuto i θ da un algoritmo di ottimizzazione, i valori f_i da inserire all'interno della formula per poter calcolare se il risultato è effettivamente maggiore o uguale di 0 si calcolano per distanza rispetto ai landmark, ovvero utilizzando la funzione kernel, come mostrato nell'esempio in Figura 7.3.1.

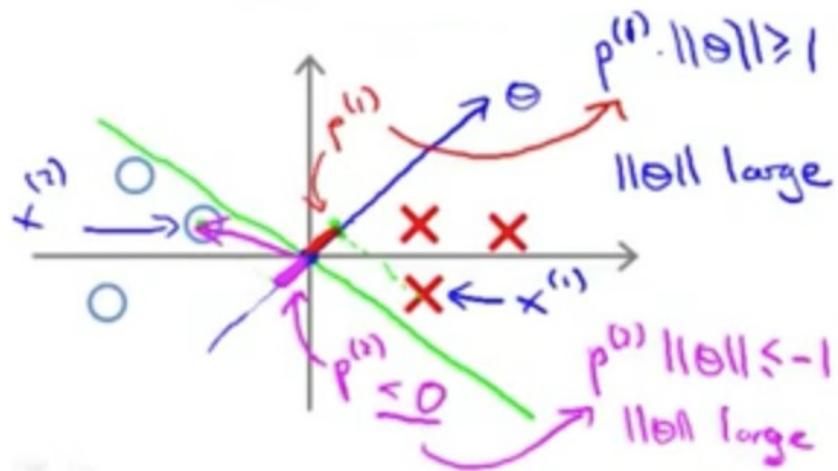
I decision boundary che si possono ottenere in questo modo sono non lineari, come il bordo rosso in Figura 7.3.1.

Un modo per decidere i landmark, il più utilizzato, è quello di utilizzare tutti gli esempi di training come landmark.

Per poter applicare le SVM con i kernel quello che bisogna fare è calcolare il nuovo vettore $f^{(i)}$ a partire da $x^{(i)}$ utilizzando la funzione kernel, per ciascun esempio i , dopodiché risolvere il problema di ottimizzazione:

$$J(\theta) = C \left[\sum_{i=1}^{n+1} y^{(i)} Cost_1(\theta^\top f^{(i)}) + (1 - y^{(i)}) Cost_0(\theta^\top f^{(i)}) \right] + \frac{1}{2} \sum_{i=1}^{n+1} \theta_i^2$$

Small Margin



Large Margin SVM

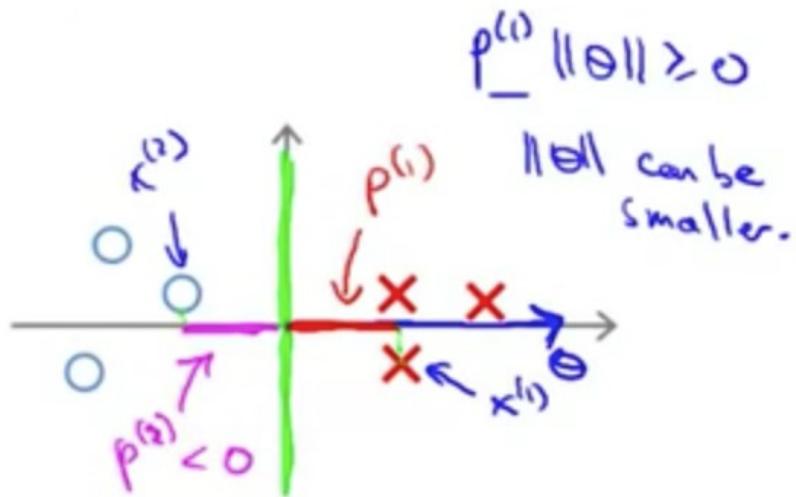


Figura 7.2.1: SVM Decision Boundary: Small Margin e Large Margin

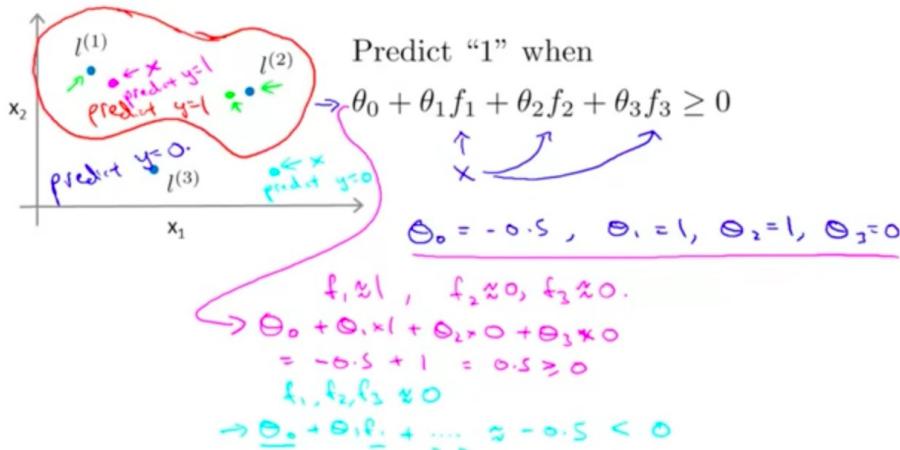


Figura 7.3.1: Esempio Kernel

dove l'apice della sommatoria della regolarizzazione è $n+1$ e non n come al solito poiché nel vettore f si aggiunge una prima componente $f_0 = 1$ e quindi il numero di feature cresce anche esso di uno.

Una piccola differenza che si può notare nelle reali implementazioni è che la sommatoria della regolarizzazione viene sostituita da $\theta^\top M \theta$, poiché $\theta^\top \theta$ è uguale alla sommatoria originale, mentre viene aggiunta una matrice M che dipende dal tipo di kernel adoperato per ottimizzare i calcoli.

Il motivo per cui i kernel non vengono utilizzati all'interno di altri algoritmi di apprendimento è che nel caso delle SVM ci sono degli accorgimenti implementativi che permettono di implementare il tutto in modo efficiente, mentre l'utilizzo dei kernel in altri algoritmi comporterebbe grandi rallentamenti.

In fine, al variare del parametro C si ha un comportamento inverso a quanto detto per il parametro di regolarizzazione λ , ovvero un C grande comporta basso bias e maggiore varianza, mentre un C piccolo comporta alto bias e bassa varianza.

Nel caso del kernel gaussiano, il parametro σ grande comporta alto bias e bassa varianza, mentre un σ basso comporta scarso bias e maggiore varianza.

I kernel più utilizzati sono il kernel gaussiano e il kernel "lineare" (che corrisponde a non utilizzare proprio kernel). Esistono alternative quali kernel polinomiali, kernel a differenza di istogramma e kernel su stringhe, ma sono meno utilizzati.

7.4 Clasificazione Multiclasse con le Support Vector Machines

Nei pacchetti che implementano le SVM sono solitamente presenti strumenti per gestire il caso multiclass, in caso non ci fossero, allora si può utilizzare la tecnica one-vs-all già descritta in precedenza.

Capitolo 8

Machine Learning su larga scala

8.1 Scelta dell'algoritmo

Per scegliere quale algoritmo usare in quale condizione non esiste un criterio unico.

Delle linee guida da seguire però possono essere quelle di utilizzare la regressione lineare o le SVM con kernel lineare nel caso di grande numero di feature relativamente ai numeri di esempi, oppure quando il numero di feature è piccolo ma il numero di esempi è molto grande (50000+).

Le SVM con kernel gaussiano quando il numero di feature è piccolo e il numero di esempi medio (10 – 10000).

Le reti neurali si comportano bene in tutte le condizioni (con architetture adeguate) ma solitamente sono più lente nell'apprendimento.

8.2 Stochastic Gradient Descent

Lo Stochastic Gradient Descent è un algoritmo di ottimizzazione che permette di lavorare con dataset di grandi dimensioni. Il Gradient Descent classico viene chiamato, in contrasto, Batch Gradient Descent.

A differenza del Batch Gradient Descent, nello Stochastic Gradeint Descent, ad ogni passo non si aggiornano i parametri θ rispetto alla somma della derivata della funzione costo rispetto a tutti gli esempi, ma solo uno alla volta. Ciò permette, anche in presenza di grandi dataset, di effettuare picoli aggiornamenti dei parametri molto più veloci che permettono solitamente di raggiungere la convergenza molto prima.

Scriviamo la funzione costo della regressione lineare come esempio, in una nuova formulazione utile per lo Stochastic Gradient Descent:

$$\begin{aligned} cost(\theta, (x^{(i)}, y^{(i)})) &= \frac{1}{2} \left(h_{\theta}(x^{(i)}) - y^{(i)} \right)^2 \\ J(\theta) &= \frac{1}{m} \sum_{i=1}^m cost(\theta, (x^{(i)}, y^{(i)})) \\ \theta_k &:= \theta_k - \alpha \frac{\partial}{\partial \theta_k} cost(\theta, (x^{(i)}, y^{(i)})) \end{aligned}$$

Calcolando le derivate parziali, si ottengono le funzioni per aggiornare i parametri (non è presente la sommatoria poiché si utilizza la derivata della funzione $cost(\theta, (x^{(i)}, y^{(i)}))$):

$$\theta_k := \theta_k - \alpha \frac{1}{m} \left(h_{\theta}(x^{(i)}) - y^{(i)} \right) \cdot x_k^{(i)}$$

L'intero processo è mostrato nell'Algoritmo 8.1. Il numero di ripetizioni può variare a seconda del numero di esempi di training: per numeri molto alti può bastare anche una sola ripetizione, per numeri inferiori si arriva tendenzialmente non oltre le 10 ripetizioni.

Algoritmo 8.1 Stochastic Gradient Descent

```

REPEAT {
    FOR  $i := 1, \dots, m$  {
        FOR  $j := 0, \dots, n$ 
             $\theta_j := \theta_j - \alpha (h_{\theta}(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$ 
    }
}

```

8.3 Mini-batch Gradient Descent

Il Mini-batch Gradient Descent è una via di mezzo tra il Batch Gradient Descent e lo Stochastic Gradient Descent. Si basa su un parametro b che denota il numero di esempi da utilizzare ad ogni iterazione. Impostando $b = m$ l'algoritmo è equivalente al Batch Gradient Descent, mentre impostando $b = 1$ l'algoritmo è equivalente allo Stochastic Gradient Descent. Il vantaggio di questo algoritmo è velocizzare ulteriormente il tempo di apprendimento se si implementa una buona vettorializzazione.

L'intero processo è mostrato nell'Algoritmo 8.2. Il numero di ripetizioni può variare a seconda del numero di esempi di training: per numeri molto alti può bastare anche una sola ripetizione, per numeri inferiori si arriva tendenzialmente non oltre le 10 ripetizioni.

Algoritmo 8.2 Mini-batch Gradient Descent

```

REPEAT {
    FOR  $i := 1, \dots, m$  (increase size =  $b$ ) {
        FOR  $j := 0, \dots, n$ 
             $\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} (h_\theta(x^{(k)}) - y^{(k)}) \cdot x_j^{(k)}$ 
    }
}

```

8.4 Convergenza

Sia lo Stochastic Gradient Descent che il Batch Gradient Descent non hanno una convergenza garantita ma continuano a muoversi nella zona di un minimo locale. Calcolare ad ogni passo $J(\theta)$ per vederne l'andamento non è una buona idea perché per calcolarla bisogna farlo su tutti gli esempi di training. Più utile risulta essere calcolare $cost(\theta, (x^{(i)}, y^{(i)}))$ e visualizzare ogni t iterazioni la media di questa funzione, dove $t = 1000$ o $t = 5000$ o altri valori a seconda della capacità di visualizzare l'andamento desiderata.

Per facilitare la convergenza ed evitare di far muovere di molto i valori di θ attorno al minimo trovato si può utilizzare un valore di α decrescente, come ad esempio:

$$\alpha = \frac{const1}{iterationNumber + const2}$$

dove $const1$ e $const2$ sono dei valori da decidere in base al problema, di base possono essere rispettivamente 1 e 0.

8.5 Online Learning

Per Online Learning si intende l'abilità di un sistema di aggiornare i propri parametri appresi ogni volta che un nuovo esempio si rende disponibile.

Ad esempio su un sito web si possono presentare dei degli item consigliati per un utente e se quell'utente clicca su di uno di quegli item, viene considerato un esempio positivo. A quel punto si può intraprendere l'aggiornamento dei parametri del sistema di apprendimento che ha restituito gli item in base al nuovo esempio, ad esempio effettuando un passo di Stochastic Gradient Descent. Sistemi di questo tipo permettono di adattarsi dinamicamente ai gusti degli utenti e di essere efficaci fin da subito nell'utilizzo in uno scenario applicativo reale.

Parte II

Apprendimento Non Supervisionato

Capitolo 9

Clustering

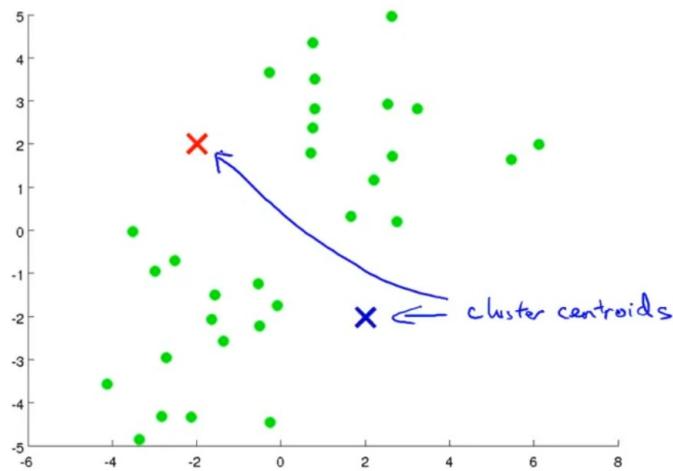
Il Clustering è un tipico esempio di **Unsupervised Learning** (apprendimento non supervisionato) in cui, dato un insieme di dati non etichettati, l'obiettivo è quello di individuare dei raggruppamenti, detti cluster, quanto più possibile coerenti.

9.1 K-Means

L'algoritmo **K-Means** è l'algoritmo di clustering più conosciuto e più utilizzato.

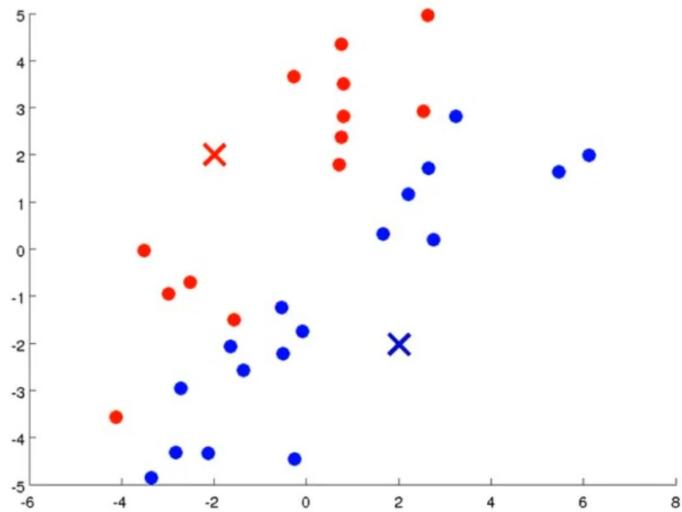
Illustreremo il suo comportamento per mezzo di figure. Inizialmente viene fornito il dataset e vengono inizializzati due punti, detti i centroidi (centroid) dei cluster, mostrati nella figura sottostante per mezzo di una croce rossa e una blu.

Sono due perché l'obiettivo in questo caso è quello di individuare due cluster.

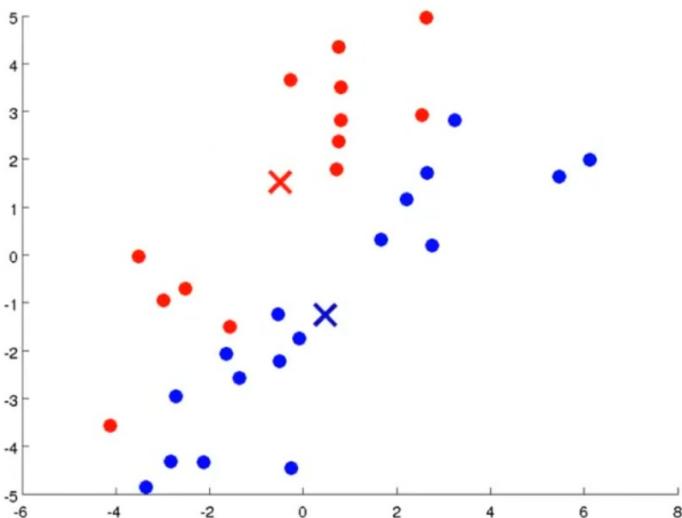


K-Means è un algoritmo iterativo che ripete due passi, il primo è il passo di assegnazione ai cluster, mentre il secondo è il passo di spostamento dei centroidi.

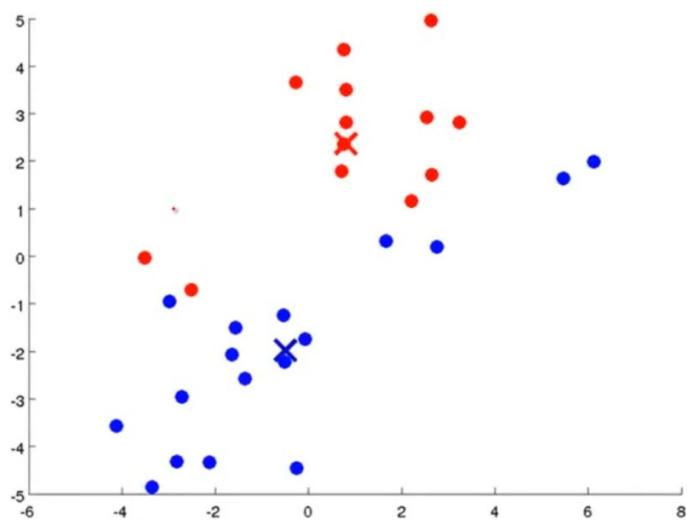
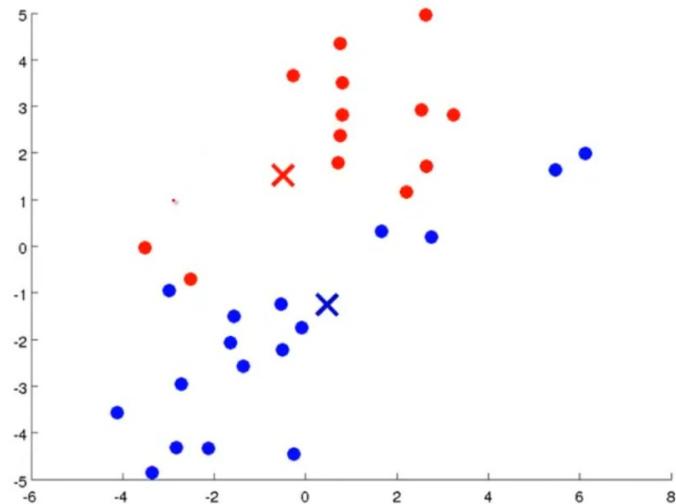
Il primo passo di assegnazione ai cluster controlla la distanza di ogni punto del dataset dai centroidi e assegna ciascun punto al centroide che gli è più vicino. Nella figura sottostante i punti più vicini al centroide del cluster rosso sono stati colorati di rosso, mentre i punti più vicini al centroid del cluster blu sono stati colorati di blu.



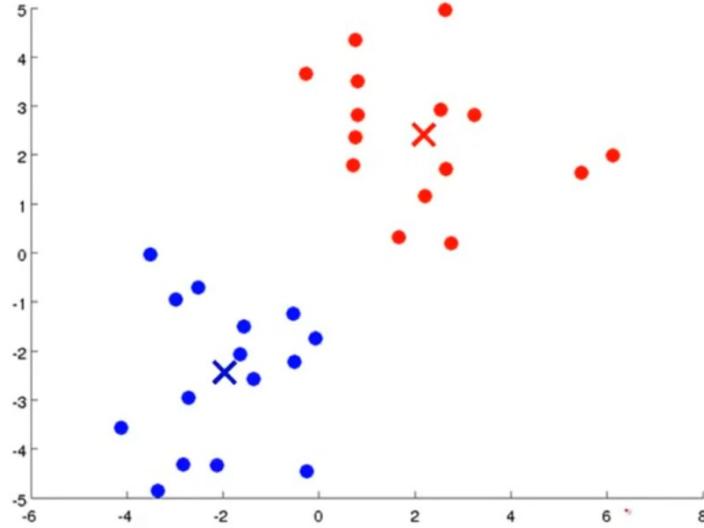
Il secondo passo, quello si spostamento dei centroidi consiste nel calcolare la media di tutti i punti nel dataset che appartengono ad un cluster e muovere in quella posizione il centroide del cluster, come mostrato nella figura sottostante.



Sono mostrate nelle figure sottostanti rispettivamente la seconda iterazione del primo passo e la seconda iterazione del secondo passo.



L'algoritmo continua in questo modo finché i centroidi non si spostano più, il che significa che l'algoritmo ha raggiunto la convergenza, come mostrato nella figura sottostante.



In modo più formale, l'algoritmo ha in input un numero K che indica il numero di cluster (come decidere questo valore verrà discusso in seguito ????????) e un training set $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$, dove ciascun $x^{(i)} \in R^n$, senza seguire dunque la convenzione adottata finora nel Supervised Learning di $x_0 = 1$, e itera un ciclo mostrato in 9.1.

Algoritmo 9.1 K-Means

Number of Clusters = K

Training Set = $\{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$

- Randomly initialize K centroids $\mu_1, \mu_2, \dots, \mu_K \in R^n$
 - Repeat until convergence
 - For $i = 1$ to m
 - * $c^{(i)} := \min_k \|x^{(i)} - \mu_k\|^2$ (the index from 1 to K of cluster centroid closer to $x^{(i)}$)
 - For $k = 1$ to K
 - * $\mu_k := \frac{\sum z}{z} \forall c^{(i)} = k, z = \text{num}(c^{(i)} = k)$ (average (mean) of point assigned to cluster k)
-

Nel caso in cui durante l'esecuzione un cluster non dovesse avere alcun punto assegnato si può decidere di eliminarlo o di reinizializzarla casualmente, ma questo non accade spesso.

Applicare K-Means nel caso di dati non perfettamente separabili può essere utile per creare delle segmentazioni dei dati stessi. Nell'esempio mostrato in

Figura 9.1.1 vengono individuati tre cluster corrispondenti a trediverse tagli di magliette basandosi su altezza e peso delle persone.

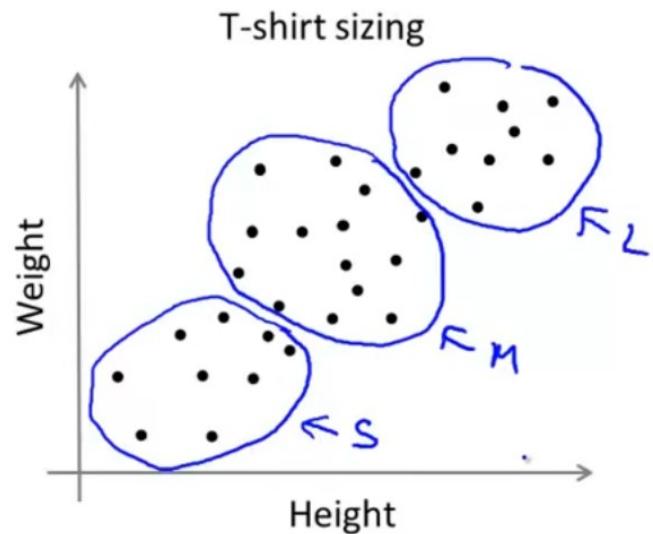


Figura 9.1.1: K-Means su cluster non separati