

Artificial Neural Networks and Deep Learning

Antonino Elia Mandri

A.A. 2019-2020

Contents

| | | |
|----------|--|----------|
| 1 | Introduzione | 3 |
| 1.1 | Apprendimento automatico | 3 |
| 1.2 | Percettrone | 4 |
| 1.2.1 | Apprendimento Hebbiano | 5 |
| 1.2.2 | Feed Forward Neural Networks | 5 |
| 1.2.2.1 | Percettrone a singolo strato | 5 |
| 1.2.2.2 | Percettrone multistrato | 6 |
| 1.2.3 | Funzioni di attivazione | 6 |
| 2 | Reti Neurali | 9 |
| 2.1 | Teorema di approssimazione universale | 10 |
| 2.2 | Regressione | 11 |
| 2.2.1 | Variabili Casuali | 11 |
| 2.2.2 | Maximum Likelihood Estimation (stimatore di massima verosomiglianza) | 12 |
| 2.3 | Ottimizzazione | 15 |
| 2.3.1 | Discesa del gradiente | 16 |
| 2.3.2 | Algoritmi di ottimizzazioni della discesa del gradiente | 19 |
| 2.3.2.1 | Momentum | 21 |
| 2.3.2.2 | Nesterov accelerated gradient (NAG) | 22 |
| 2.3.2.3 | Adagrad | 23 |
| 2.3.2.4 | Adadelta | 24 |
| 2.3.2.5 | RMSprop | 25 |
| 2.3.2.6 | Adam | 25 |
| 2.3.2.7 | AdaMax | 26 |
| 2.3.2.8 | Nadam | 27 |
| 2.4 | Altre tecniche di ottimizzazione | 28 |
| 2.4.1 | Shuffling e Curriculum Learning | 28 |
| 2.4.2 | Batch Normalization | 29 |
| 2.4.3 | Early Stopping | 29 |

| | |
|-----------------|----------|
| CONTENTS | 2 |
|-----------------|----------|

| | |
|---|-----------|
| 3 Classificazioni di immagini | 31 |
| 3.1 Dataset CIFAR-10 | 33 |
| 3.2 Nearest Neighbor Classifier | 34 |
| 3.3 K-Nearest Neighbor Classifier | 35 |
| 3.4 Classificatore Lineare (Linear Classifier) | 37 |
| 3.4.1 Interpretazione di un Classificatore Lineare. | 40 |
| 3.4.1.1 Interpretazione Geometrica | 40 |
| 3.4.1.2 Template | 41 |
| 3.4.1.3 Coming Soon | 41 |

Chapter 1

Introduzione

1.1 Apprendimento automatico

Il machine learning è una branca dell'intelligenza artificiale che raccoglie un insieme di metodi quali: statistica computazionale, riconoscimento di pattern, reti neurali artificiali, filtraggio adattivo, teoria dei sistemi dinamici, elaborazione delle immagini, data mining, algoritmi adattivi, ecc; che utilizza metodi statistici per migliorare progressivamente la performance di un algoritmo nell'identificare pattern nei dati. Nell'ambito dell'informativa, l'apprendimento automatico è una variante alla programmazione tradizionale nella quale si predispone in una macchina l'abilità di apprendere qualcosa dai dati in maniera autonoma, senza ricevere istruzioni esplicite a riguardo. $D =$

Immaginiamo di possedere un insieme di una certa esperienza E , per esempio dei dati, che chiameremo $D = x_1, x_2, \dots, x_N$, definiamo quindi i seguenti paradigmi di apprendimento:

- apprendimento supervisionato (supervised learning): in cui al modello vengono forniti degli output desiderati t_1, t_2, \dots, t_N e l'obiettivo è quello di estrarre una regola generale che associa l'input D all'output corretto;
- apprendimento non supervisionato (unsupervised learning): è una tecnica di apprendimento automatico che consiste nel fornire al sistema informatico una serie di input (esperienza del sistema), D nel nostro caso, che egli riclassificherà ed organizzerà sulla base di caratteristiche comuni per cercare di effettuare ragionamenti e previsioni sugli input successivi;
- L'apprendimento per rinforzo (reinforcement learning): è una tecnica di apprendimento automatico che punta ad attuare sistemi in grado di apprendere ed adattarsi alle mutazioni dell'ambiente in cui sono immersi, attraverso la distribuzione di una "ricompensa" detta rinforzo che consiste nella valutazione delle loro prestazioni. Il modello produce una serie di azioni a_1, a_2, \dots, a_N che interagiscono con l'ambiente e ricevendo una serie di

ricompense r_1, r_2, \dots, r_N impara a produrre azioni che massimizzino le ricompense nel lungo periodo.

1.2 Percettrone

Nell'apprendimento automatico, il percettrone è un tipo di classificatore binario che mappa i suoi ingressi \mathbf{x} (un vettore di tipo reale) in un valore di output $f(\mathbf{x})$ (uno scalare di tipo reale) calcolato con

$$f(\mathbf{x}) = \chi(\langle \mathbf{w}, \mathbf{x} \rangle + b) \quad (1.1)$$

dove \mathbf{w} è un vettore di pesi con valori reali, l'operatore $\langle \cdot, \cdot \rangle$ è il prodotto scalare (che calcola una somma pesata degli input), b è il bias, un termine costante che non dipende da alcun valore in input e $\chi(y)$ è la funzione di output. Le scelte più comuni per la funzione $\chi(y)$ sono:

1. $\chi(y) = \text{sign}(y)$
2. $\chi(y) = y\Theta(y)$
3. $\chi(y) = y$

dove $\Theta(y)$ è la funzione di Heaviside.

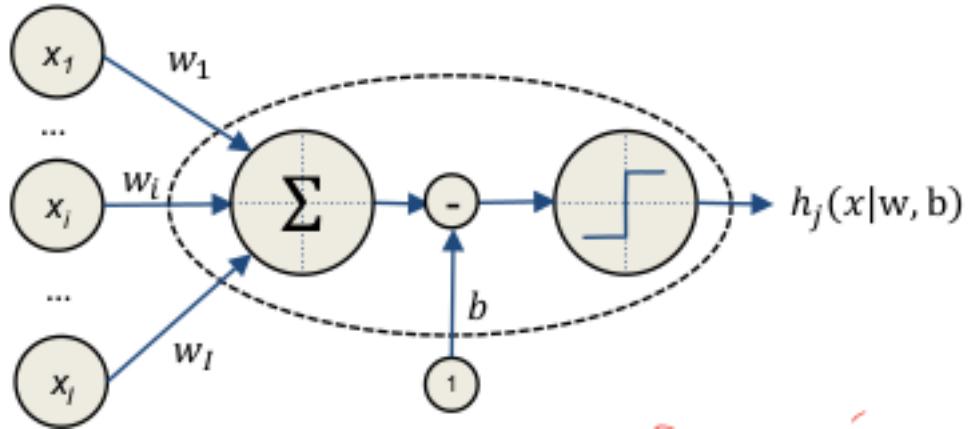


Figure 1.1: percettrone

$$h_j(\mathbf{x}|\mathbf{w}, b) = h_j \left(\sum_{i=1}^I w_i \cdot x_i - b \right) = h_j \left(\sum_{i=0}^I w_i \cdot x_i \right) = h_j (\mathbf{w}^T \mathbf{x}) \quad (1.2)$$

Non tutti i problemi di classificazione sono affrontabili con strumenti lineari come il percettrone. Sorgono spontanee alcune domande, come inizializziamo e modifichiamo il vettore di pesi \mathbf{w} del percettrone? Quale funzione di attivazione scegliamo?

1.2.1 Apprendimento Hebbiano

«The strength of a synapse increases according to the simultaneous activation of the relative input and the desired target» (Donald Hebb, *The Organization of Behavior*, 1949).

La regola di Hebb è la seguente: l'efficacia di una particolare sinapsi cambia se e solo se c'è un'intensa attività simultanea dei due neuroni, con un'alta trasmissione di input nella sinapsi in questione. L'apprendimento Hebbiano può essere riassunto come segue:

$$\begin{cases} w_i^{k+1} = w_i^k + \Delta w_i^k \\ \Delta w_i^k = \eta \cdot x_i^k \cdot t^k \end{cases} \quad (1.3)$$

dove:

- η : rateo di apprendimento;
- x_i^k : l'i-esimo input al tempo k ;
- t^k : l'output desiderato al tempo k .

L'inizializzazione dei pesi parte con dei valori casuali. La soluzione può non esistere e se esiste non essere unica, ma ugualmente corrette. Questo algoritmo può non convergere alla soluzione per due motivi:

1. La soluzione non esiste;
2. η è troppo grande, continuiamo a modificare i pesi con passo elevato, viceversa un valore di η troppo piccolo aumenta sensibilmente il tempo di convergenza.

1.2.2 Feed Forward Neural Networks

Una rete neurale feed-forward ("rete neurale con flusso in avanti") o rete feed-forward è una rete neurale artificiale dove le connessioni tra le unità non formano cicli, differenziandosi dalle reti neurali ricorrenti. Questo tipo di rete neurale fu la prima e più semplice tra quelle messe a punto. In questa rete neurale le informazioni si muovono solo in una direzione, avanti, rispetto a nodi d'ingresso, attraverso nodi nascosti (se esistenti) fino ai nodi d'uscita. Nella rete non ci sono cicli. Le reti feed-forward non hanno memoria di input avvenuti a tempi precedenti, per cui l'output è determinato solamente dall'attuale input.

1.2.2.1 Percettrone a singolo strato

La più semplice rete feed-forward è il percettrone a singolo strato (SLP dall'inglese single layer perceptron), utilizzato verso la fine degli anni '60. Un SLP è costituito da un strato in ingresso, seguito direttamente dall'uscita. Ogni unità di ingresso è collegata ad ogni unità di uscita. In pratica questo tipo di rete

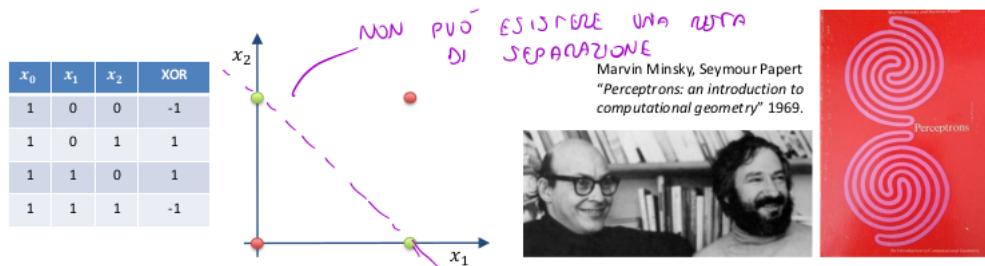


Figure 1.2: Problema XOR percepitrone

neurale ha un solo strato che effettua l'elaborazione dei dati, e non presenta nodi nascosti, da cui il nome. Gli SLP sono molto limitati a causa del piccolo numero di connessioni e dell'assenza di gerarchia nelle caratteristiche che la rete può estrarre dai dati (questo significa che è capace di combinare i dati in ingresso una sola volta). Famosa fu la dimostrazione, che un SLP non riesce neanche a rappresentare la funzione XOR.

1.2.2.2 Percettrone multistrato

Il Percettrone multistrato (in acronimo MLP dall'inglese Multilayer perceptron) è un modello di rete neurale artificiale che mappa insiemi di dati in ingresso in un insieme di dati in uscita appropriati. È fatta di strati multipli di nodi in un grafo diretto, con ogni strato completamente connesso al successivo. Eccetto che per i nodi in ingresso, ogni nodo è un neurone (elemento elaborante) con una funzione di attivazione non lineare. Il Percettrone multistrato usa una tecnica di apprendimento supervisionato chiamata backpropagation per l'allenamento della rete. La MLP è una modifica del Percettrone lineare standard e può distinguere i dati che non sono separabili linearmente.

Prima di addentrarsi in metodologie di setup delle reti neurali è utile introdurre alcuni concetti.

1.2.3 Funzioni di attivazione

Vediamo brevemente diversi tipi di funzioni di attivazione:

ReLU. The Rectified Linear Unit è una funzione di attivazione definita come la parte positiva del suo argomento:

$$f(x) = x^+ = \max(0, x) \quad (1.4)$$

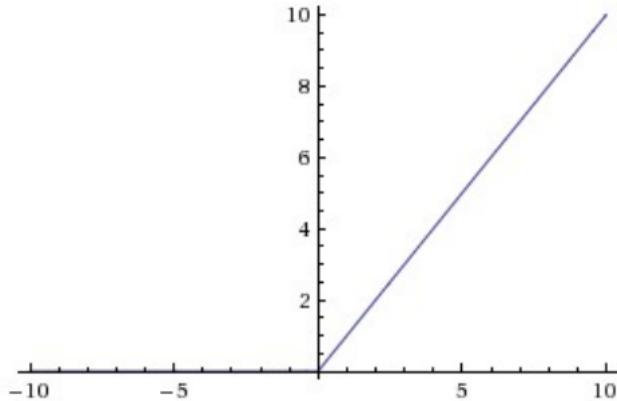


Figure 1.3: grafico ReLU

dove x è l'input a un neurone.

Sigmoid. La funzione sigmoidea è una funzione matematica che produce una curva sigmoide; una curva avente un andamento ad "S". Spesso, la funzione sigmoide si riferisce ad uno speciale caso di funzione logistica mostrata a destra e definita dalla formula:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (1.5)$$

Generalmente, una funzione sigmoidea è una funzione continua e derivabile, che ha una derivata prima non negativa e dotata di un minimo locale ed un massimo locale. Le funzioni sigmoide sono spesso usate nelle reti neurali per introdurre la non linearità nel modello e/o per assicurarsi che determinati segnali rimangano all'interno di specifici intervalli. Un motivo per la relativa popolarità nelle reti neurali è perché la funzione sigmoidea soddisfa questa proprietà:

$$\frac{d}{dx} \text{sig}(x) = \text{sig}(x)(1 - \text{sig}(x)) \quad (1.6)$$

Questa relazione polinomiale semplice fra la derivata e la funzione stessa è, dal punto di vista informatico, semplice da implementare.

Tangente iperbolica:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (1.7)$$

Queste e altre funzioni verranno esaminate più attentamente dopo aver introdotto tecniche di ottimizzazione delle reti neurali. Prima però vediamo delle applicazioni pratiche.

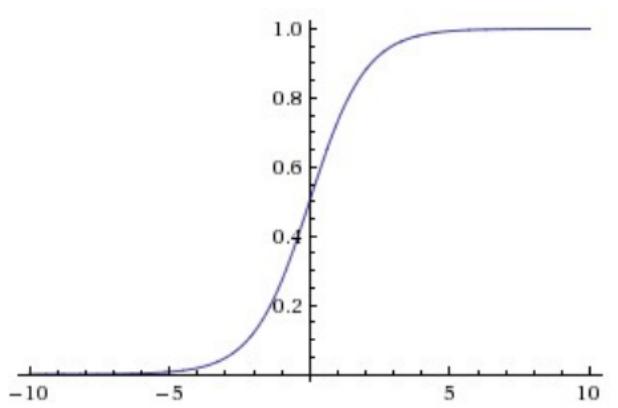


Figure 1.4: grafico sigmoide

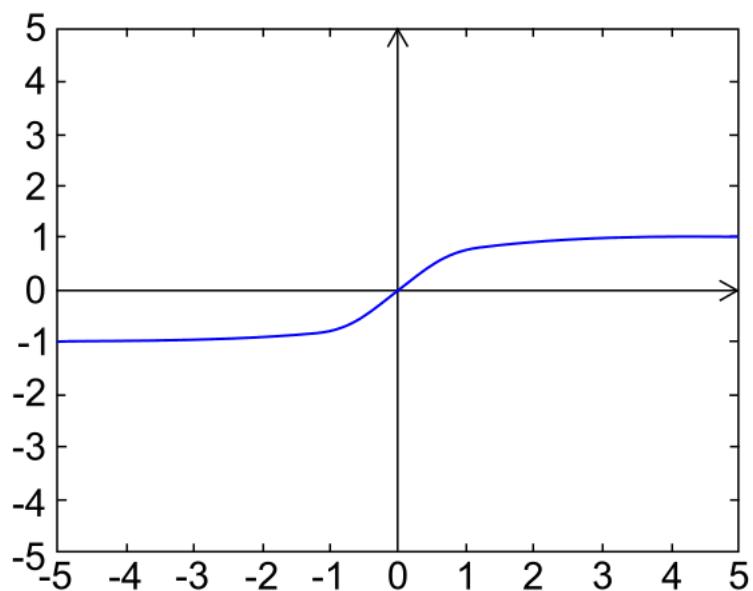


Figure 1.5: grafico tanh

Chapter 2

Reti Neurali

L'utilità dei modelli di rete neurale sta nel fatto che queste possono essere usate per comprendere una funzione utilizzando solo le osservazioni sui dati. Ciò è particolarmente utile nelle applicazioni in cui la complessità dei dati o la difficoltà di elaborazione rende la progettazione di una tale funzione impraticabile con i normali procedimenti di analisi manuale.

I compiti a cui le reti neurali sono applicate possono essere classificate nelle seguenti grandi categorie di applicazioni:

- funzioni di approssimazione, o di regressione, tra cui la previsione di serie temporali e la modellazione;
- classificazione, compresa la struttura e la sequenza di generici riconoscimenti, l'individuazione delle novità ed il processo decisionale;
- l'elaborazione dei dati, compreso il "filtraggio" (eliminazione del rumore), il clustering, separazione di segnali e compressione.

Le aree di applicazione includono i sistemi di controllo (controllo di veicoli, controllo di processi), simulatori di giochi e processi decisionali (backgammon, scacchi), riconoscimento di pattern (sistemi radar, identificazione di volti, riconoscimento di oggetti, ecc), riconoscimenti di sequenze (riconoscimento di gesti, riconoscimento vocale, OCR), diagnosi medica, applicazioni finanziarie, data mining, filtri spam per e-mail.

Pregi

Le reti neurali per come sono costruite lavorano in parallelo e sono quindi in grado di trattare molti dati. Si tratta in sostanza di un sofisticato sistema di tipo statistico dotato di una buona immunità al rumore; se alcune unità del sistema dovessero funzionare male, la rete nel suo complesso avrebbe delle riduzioni di prestazioni ma difficilmente andrebbe incontro ad un blocco del sistema. I software di ultima generazione dedicati alle reti neurali richiedono comunque

buone conoscenze statistiche; il grado di apparente utilizzabilità immediata non deve trarre in inganno, pur permettendo all'utente di effettuare subito previsioni o classificazioni, seppure con i limiti del caso. Da un punto di vista industriale, risultano efficaci quando si dispone di dati storici che possono essere trattati con gli algoritmi neurali. Ciò è di interesse per la produzione perché permette di estrarre dati e modelli senza effettuare ulteriori prove e sperimentazioni.

Difetti

I modelli prodotti dalle reti neurali, anche se molto efficienti, non sono spiegabili in linguaggio simbolico umano: i risultati vanno accettati "così come sono", da cui anche la definizione inglese delle reti neurali come "black box": in altre parole, a differenza di un sistema algoritmico, dove si può esaminare passo-passo il percorso che dall'input genera l'output, una rete neurale è in grado di generare un risultato valido, o comunque con una alta probabilità di essere accettabile, ma non è possibile spiegare come e perché tale risultato sia stato generato. Come per qualsiasi algoritmo di modellazione, anche le reti neurali sono efficienti solo se le variabili predittive sono scelte con cura.

Non sono in grado di trattare in modo efficiente variabili di tipo categorico (per esempio, il nome della città) con molti valori diversi. Necessitano di una fase di addestramento del sistema che fissi i pesi dei singoli neuroni e questa fase può richiedere molto tempo, se il numero dei record e delle variabili analizzate è molto grande. Non esistono teoremi o modelli che permettano di definire la rete ottima, quindi la riuscita di una rete dipende molto dall'esperienza del creatore.

2.1 Teorema di approssimazione universale

Nella teoria matematica delle reti neurali artificiali, il teorema di approssimazione universale afferma che una feed-forward network con un singolo strato nascosto e contenente un numero finito di neuroni può approssimare una qualsiasi funzione misurabile secondo Lebesgue, con qualsiasi grado di accuratezza, su un sottoinsieme compatto di \mathbb{R}^n sotto deboli ipotesi sulla funzione di attivazione dei neuroni. Il teorema non dice nulla su algoritmi di apprendimento da utilizzare. Sebbene una rete feed-forward con un singolo strato nascosto sia un approssimatore universale, l'ampiezza di queste reti deve essere esponenzialmente grande. Nel 2017 Lu et al. [1] dimostrarono una variante del teorema per reti feed-forward con ampiezza limitata. In particolare provarono che una rete di ampiezza $n+4$ con funzione di attivazione ReLU può approssimare una qualsiasi funzione integrabile secondo Lebesgue definita su uno spazio $n - dimensionale$ rispetto alla norma L_1 se è permesso alla rete di crescere in profondità (quindi non più a singolo strato nascosto). Provarono anche la limitata potenza espressiva se l'ampiezza della rete è minore o uguale a n . Nessuna funzione integrabile secondo Lebesgue ad eccezione di quelle definite su insiemi a misura nulla può essere approssimata da una rete con ampiezza n e funzione di attivazione

ReLU. La formulazione originale del teorema non fa assunzioni che la funzione di attivazione sia ReLU ma solo che sia continua, limitata e non costante. I due teoremi sono formalmente enunciati nel modo seguente:

Aampiezza illimitata: Sia $\varphi : \mathbb{R} \rightarrow \mathbb{R}$ una funzione continua, limitata e non costante (chiamata *funzione di attivazione*). Sia I_m l'iper cubo unitario $[0, 1]^m$. Sia $C(I_m)$ lo spazio delle funzioni a valori reali definite su I_m . Dato un $\varepsilon > 0$ e una qualsiasi funzione $f \in C(I_m)$, esiste allora un intero $N \in \mathbb{N}$, costanti reali $v_i, b_i \in \mathbb{R}$ e vettori reali $\mathbf{w}_i \in \mathbb{R}^m$ per $i = 1, \dots, N$ tale che possiamo definire:

$$F(\mathbf{x}) = \sum_{i=1}^N v_i \varphi(\mathbf{w}_i^T \mathbf{x} + b_i) \quad (2.1)$$

come realizzazione approssimativa della funzione f per cui vale:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon$$

per ogni $\mathbf{x} \in \mathbb{R}^m$. In altre parole, funzioni della forma $F(\mathbf{x})$ sono dense in $C(I_m)$. Questo vale ancora se si sostituisce a I_m un qualsiasi sotto insieme compatto di \mathbb{R}^m .

Aampiezza limitata: Per ogni funzione integrabile secondo Lebesgue $f : \mathbb{R}^n \rightarrow \mathbb{R}$ e ogni $\varepsilon > 0$, esiste una rete fully-connected ReLU A con ampiezza $d_m \leq n+4$ tale che la funzione F_A rappresentata da tale rete soddisfa:

$$\int_{\mathbb{R}^n} |F_A(\mathbf{x}) - f(\mathbf{x})| d\mathbf{x} < \varepsilon. \quad (2.2)$$

2.2 Regressione

L'analisi della regressione è una tecnica usata per analizzare una serie di dati che consistono in una variabile dipendente e una o più variabili indipendenti. Lo scopo è stimare un'eventuale relazione funzionale esistente tra la variabile dipendente e le variabili indipendenti. La variabile dipendente nell'equazione di regressione è una funzione delle variabili indipendenti più un termine d'errore. Quest'ultimo è una variabile casuale e rappresenta una variazione non controllabile e imprevedibile nella variabile dipendente. I parametri sono stimati in modo da descrivere al meglio i dati. Il metodo più comunemente utilizzato per ottenere le migliori stime è il metodo dei "minimi quadrati" (OLS), ma sono utilizzati anche altri metodi.

2.2.1 Variabili Casuali

In matematica, e in particolare nella teoria della probabilità, una variabile casuale (detta anche variabile aleatoria o variabile stocastica) è una variabile che può assumere valori diversi in dipendenza da qualche fenomeno aleatorio.

Le variabili casuali sono per noi molto importanti perché l'obiettivo di una rete neurale per regressione è quello di approssimare una funzione target t incognita avendo a disposizione N osservazioni (coppie input-output della funzione t). Anche per la classificazioni di immagini è possibile ricondursi a variabili aleatorie.

2.2.2 Maximum Likelihood Estimation (stimatore di massima verosomiglianza)

Supponiamo di avere N campioni di una variabile aleatoria di cui conosciamo la distribuzione di probabilità ma non conosciamo tutti i parametri di tale distribuzione, MLE si pone l'obiettivo di trovare i parametri tali per cui è massima la probabilità che gli N campioni appartengano alla distribuzione di probabilità con i parametri così trovati. Dato $\boldsymbol{\theta} = (\theta_1, \theta_2, \dots, \theta_p)^T$ un vettore di parametri, cerchiamo $\boldsymbol{\theta}_{MLE}$:

- Scriviamo la probabilità condizionata $L = P(\mathbf{x}|\boldsymbol{\theta})$ con \mathbf{x} vettore di input;
- opzionalmente, se agevola i calcoli, calcoliamo $l = \log(P(\mathbf{x}|\boldsymbol{\theta}))$;
- cerchiamo il massimo rispetto a $\boldsymbol{\theta}$ con gli strumenti dell'analisi matematica:
 - $\nabla_{\boldsymbol{\theta}}(L) = 0$
 - Controlliamo che il valore di $\boldsymbol{\theta}_{MLE}$ sia un massimo (tramite hessiana o altro)

Questa è la risoluzione analitica, non sempre possibile o conveniente. In ogni caso cerchiamo quel valore di $\boldsymbol{\theta}$ che massimizza la probabilità, quindi in generale si può affrontare il problema con altri metodi quali:

- tecniche di ottimizzazione: per esempio moltiplicatori di Lagrange;
- tecniche numeriche: per esempio la discesa del gradiente, approfondita largamente in seguito;

Vediamo un esempio classico della probabilità:

supponiamo di avere N indipendenti e identicamente distribuiti (i.i.d.) campioni di numeri reali provenienti da una distribuzione gaussiana di varianza σ^2 nota e media μ incognita:

$$\mathbf{x} = x_1, x_2, \dots, x_n \sim N(\mu, \sigma^2)$$

con

$$N(\mu, \sigma^2) = p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

calcoliamo la likelihood $L = P(\mathbf{x}|\boldsymbol{\theta})$:

$$L = p(x_1, x_2, \dots, x_n | \mu, \sigma^2) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}$$

passiamo al logaritmo:

$$\begin{aligned} l &= \log\left(\prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right) = \sum_{i=1}^N \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right) \\ &= N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (x_i - \mu)^2 \end{aligned}$$

deriviamo rispetto a μ che è il parametro che vogliamo stimare:

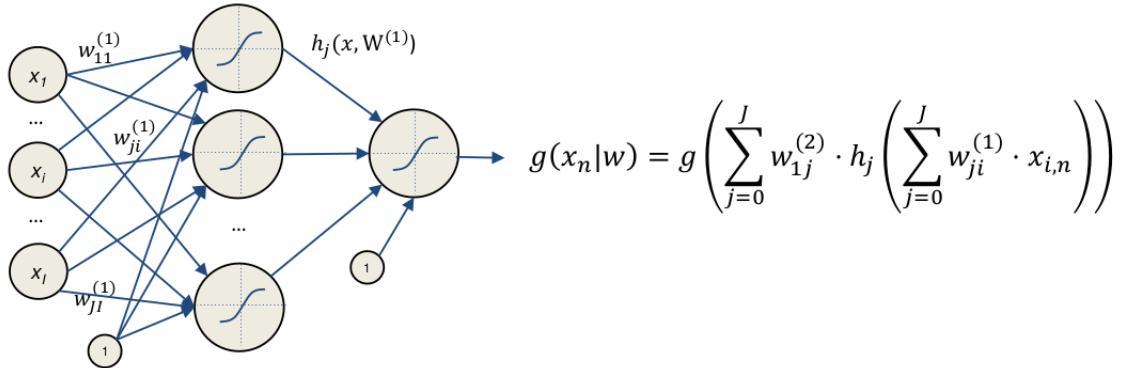
$$\begin{aligned} \frac{\partial l(\mathbf{x}|\mu)}{\partial \mu} &= \frac{\partial}{\partial \mu} \left(N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (x_i - \mu)^2 \right) \\ &= \frac{1}{\sigma^2} \sum_{i=1}^N (x_i - \mu) \end{aligned}$$

ora uguagliamo a zero la derivata parziale:

$$\begin{aligned} \frac{1}{\sigma^2} \sum_{i=1}^N (x_i - \mu) &= 0 \\ \sum_{i=1}^N (x_i - \mu) &= 0 \\ \sum_{i=1}^N x_i &= \sum_{i=1}^N \mu \\ \mu_{MLE} &= \frac{1}{N} \sum_{i=1}^N x_i. \end{aligned}$$

Vediamo un esempio pratico di come applicare lo stimatore di massima verosomiglianza con una generica rete neurale per la regressione.

Neural Networks for Regression



Goal: approximate a target function t having N observations

$$t_n = g(x_n|w) + \epsilon_n, \quad \epsilon_n \sim N(0, \sigma^2)$$

Figure 2.1: Rete neurale per regressione

Il problema della regressione può essere esposto in maniera semplice e intuitiva, sacrificando la formalità, come dati una serie di punti (coppie input-output della funzione incognita) campionati sperimentalmente cerchiamo una qualche funzione che a parità di input fornisca l'output più "vicino" possibile. Osserviamo che non cerchiamo una funzione che passi esattamente dai punti campionati (quello è il compito dell'interpolazione) ma che minimizzi la differenza con i tutti i campioni. Nell'esempio in fig. 2.1 abbiamo una funzione incognita t_n che vogliamo stimare tramite la funzione calcolata dalla rete neurale $g(x_n|w)$. Poiché dobbiamo tenere conto che la nostra approssimazione presenterà degli errori e non avendo alcuna informazione aggiuntiva sulla funzione t_n modellizziamo questo errore come un rumore aggiunto alla funzione $g(x_n|w)$. Per semplicità dei calcoli ipotizziamo che il rumore abbia varianza nota.

$$\begin{aligned} t_n &= g(x_n|w) + \epsilon_n \\ \epsilon_n &\sim N(0, \sigma^2) \end{aligned}$$

è allora intuitivo osservare che in ogni punto t_n ha distribuzione di probabilità gaussiana di media $g(x_n|w)$ e varianza σ^2

$$\Rightarrow t_n \sim N(g(x_n|w), \sigma^2).$$

Come nell'esempio teorico dobbiamo stimare la media di una distribuzione normale.

$$p(t|g(x|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-g(x|w))^2}{2\sigma^2}}$$

scriviamo la likelihood per l'insieme di osservazioni $L(w) = p(\mathbf{t}|g(\mathbf{x}|w), \sigma^2)$

$$\begin{aligned} L(w) &= p(t_1, t_2, \dots, t_N | g(x|w), \sigma^2) = \prod_{i=1}^N p(t_i | g(x_i|w), \sigma^2) \\ &= \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i - g(x_i|w))^2}{2\sigma^2}} \end{aligned}$$

cerchiamo i pesi w che massimizzano la likelihood $L(w)$

$$\begin{aligned} argmax_w L(w) &= argmax_w \left(\prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i - g(x_i|w))^2}{2\sigma^2}} \right) \\ &= argmax_w \left(\sum_{i=1}^N \log \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i - g(x_i|w))^2}{2\sigma^2}} \right) \right) \\ &= argmax_w \left(N \cdot \log \left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (t_i - g(x_i|w))^2 \right) \\ &= argmin_w \left(\sum_{i=1}^N (t_i - g(x_i|w))^2 \right). \end{aligned}$$

Siamo giunti quindi a trovare tramite lo stimatore di massima verosomiglianza la **funzione di errore** da minimizzare che useremo per allenare la nostra rete e quindi trovare i pesi w ottimali per la regressione. Infatti $\sum_{i=1}^N (t_i - g(x_i|w))^2$ è definita SSE (*Sum of Squared Errors*) ed è alla base della tecnica di ottimizzazione e regressione nota come **metodo dei minimi quadrati**. In maniera analoga è possibile procedere anche nel caso la varianza sia incognita.

2.3 Ottimizzazione

Fino ad ora abbiamo solo accennato a come impostare i pesi di una rete neurale (hebbian learning) ma non siamo mai entrati nello specifico. Ricordiamo brevemente come è composta una rete neurale e come valutiamo la sua bontà.

- Una rete è formata da uno o più percetroni, disposti in vari strati di ampiezza variabile. Per il momento ci limitiamo alle reti Fully Connected (FC) in cui ogni neurone è collegato a tutti i neuroni dello strato successivo, ad eccezione ovviamente dello strato di output. Ogni neurone esegue la somma pesata del proprio vettore di input e propaga in uscita il valore della funzione di attivazione (ReLU, sigmoide etc...) applicata alla somma di input.
- Ogni rete deve essere validata tramite una error function (o loss function) che permetta di quantificare la bontà della rete e quindi avere un responso sulla validità dei pesi impostati.

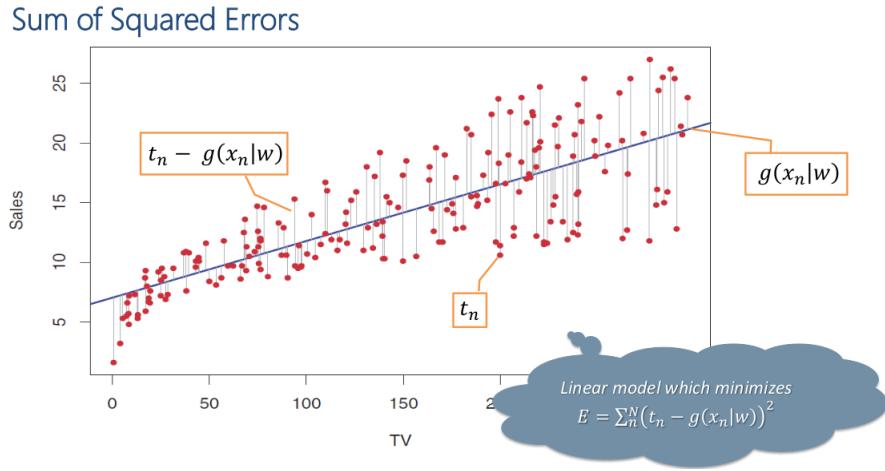


Figure 2.2: SSE

Tale error function che ricordiamo essere funzione della rete e quindi dei suoi pesi w , deve essere differenziabile (o quasi) rispetto a w . L’idea generale è quella di sfruttare gli strumenti del calcolo infinitesimale per trovare il minimo della funzione di errore. Nel caso la funzione di errore fosse differenziabile e convessa sappiamo dalla teoria del calcolo che certamente esiste un minimo assoluto e sappiamo ricavarlo analiticamente. Sfortunatamente nel caso generale possiamo richiedere alla funzione di errore solo la continuità e la quasi differenziabilità, inoltre già una rete neurale di medio bassa complessità presenta al suo interno migliaia o milioni di pesi che rendono praticamente impossibile trovare una soluzione analitica. Quello che si riesce agilmente a fare è calcolare numericamente il gradiente (con un bassissimo tasso di errore) che ricordiamo fornisce la direzione di massima crescita della funzione.

2.3.1 Discesa del gradiente

Gradiente Nel calcolo differenziale vettoriale, il gradiente di una funzione a valori reali (ovvero di un campo scalare) è una funzione vettoriale. Il gradiente di una funzione è spesso definito come il vettore che ha come componenti le derivate parziali della funzione, anche se questo vale solo se si utilizzano coordinate cartesiane ortonormali. In generale, il gradiente di una funzione f , denotato con ∇f , (il simbolo ∇ si legge nabla), è definito in ciascun punto dalla seguente relazione: per un qualunque vettore \mathbf{v} , il prodotto scalare $\nabla f \cdot \mathbf{v}$ dà il valore della derivata direzionale di f rispetto a \mathbf{v} (sse f è differenziabile). Nel caso unidimensionale il gradiente corrisponde alla derivata della funzione e indica la pendenza, quindi il tasso di variazione della funzione, e il verso in cui la funzione cresce (nel caso unidimensionale la direzione del vettore è determinata e unica,

il segno invece determina il verso, positiva la funzione cresce a destra, negativa la funzione cresce a sinistra). La derivata è definita formalmente come:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

mentre chiamiamo derivata parziale rispetto a x_i di una generica funzione reale $f : \mathbb{R}^n \rightarrow \mathbb{R}$ derivabile:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n)}{h}$$

con $i \in \{1, \dots, n\}$. Il gradiente è quindi un vettore le cui componenti sono tutte le derivate parziali rispetto agli assi di una funzione:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right)$$

ogni componente indica quanto e in che verso la funzione cresce, né consegue che un vettore così definito individui la direzione e verso in cui la funzione ha crescita massima, inoltre il modulo indica di quanto cresce la funzione in questa direzione.

Considerazioni pratiche. Notiamo che la formulazione matematica del gradiente è definito come il limite del rapporto incrementale con l'incremento h che tende a zero, ovviamente tale operazione non può essere svolta direttamente da un calcolatore ma viene svolta tramite le tecniche del calcolo numerico, in prima approssimazione è sufficiente calcolare il rapporto con un incremento molto piccolo come ad esempio $1e-5$ inoltre spesso funziona meglio (soprattutto in prossimità di punti angolosi, in cui formalmente la derivata non è definita) la *derivata numerica simmetrica*:

$$\frac{f(x+h) - f(x-h)}{2h}$$

Idea generale In ottimizzazione e analisi numerica il metodo di discesa del gradiente (detto anche metodo del gradiente, metodo steepest descent o metodo di discesa più ripida) è una tecnica che consente di determinare i punti di massimo e minimo di una funzione di più variabili. Si voglia risolvere il seguente problema di ottimizzazione non vincolata nello spazio n -dimensionale \mathbb{R}^n

$$\text{minimizzare } f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n.$$

La tecnica di discesa del gradiente si basa sul fatto che, per una data funzione $f(\mathbf{x})$, la direzione di massima discesa in un assegnato punto \mathbf{x} corrisponde a quella determinata dall'opposto del suo gradiente in quel punto $\mathbf{p}_k = -\nabla f(\mathbf{x})$. Questa scelta per la direzione del gradiente garantisce che la soluzione tenda ad un punto di minimo di f . Il metodo del gradiente prevede dunque di partire da

una soluzione iniziale \mathbf{x}_0 scelta arbitrariamente e di procedere iterativamente aggiornandola come

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \eta_k \mathbf{p}_k$$

dove $\eta_k \in \mathbb{R}^+$ corrisponde alla lunghezza del passo di discesa, la cui scelta diventa cruciale nel determinare la velocità con cui l'algoritmo convergerà alla soluzione richiesta. Si parla di metodo stazionario nel caso in cui si scelga un passo $\eta_k = \bar{\eta}$ costante per ogni k , viceversa il metodo si definisce dinamico. In quest'ultimo caso una scelta conveniente, ma computazionalmente più onerosa rispetto a un metodo stazionario, consiste nell'ottimizzare, una volta determinata la direzione di discesa \mathbf{p}_k , la funzione di una variabile $f_k(\eta_k) := f(\mathbf{x}_k + \eta_k \mathbf{p}_k)$ in maniera analitica o in maniera approssimata. Si noti che, a seconda della scelta del passo di discesa, l'algoritmo potrà convergere a uno qualsiasi dei minimi della funzione f , sia esso locale o globale.

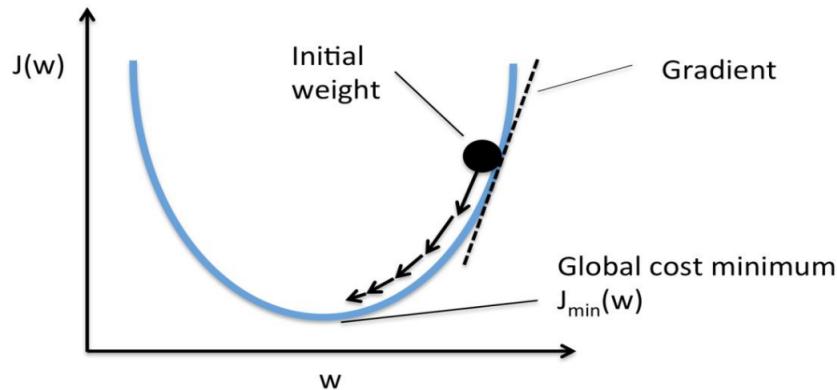


Figure 2.3: Discesa gradiente 1-D

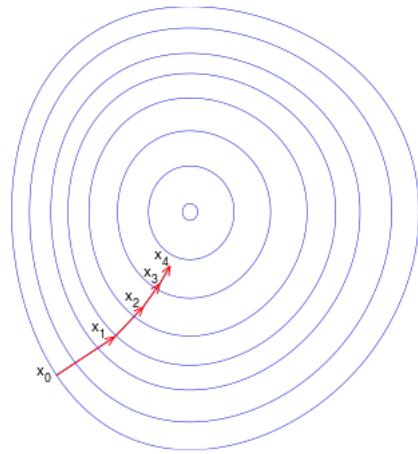


Figure 2.4: Discesa gradiente 2-D

Algoritmo generale

Lo schema generale per l'ottimizzazione di una funzione $f(\mathbf{x})$ mediante metodo del gradiente è il seguente:

Algorithm 2.1 Discesa del gradiente

```

 $k = 0$ 
while  $\nabla f(\mathbf{x}) \neq 0$ 
    calcolare la direzione di discesa  $\mathbf{p}_k = -\nabla f(\mathbf{x})$ 
    calcolare il passo di discesa  $\eta_k$ 
     $\mathbf{x}_{k+1} = \mathbf{x}_k + \eta_k \mathbf{p}_k$ 
     $k = k + 1$ 
end.
```

2.3.2 Algoritmi di ottimizzazioni della discesa del gradiente

Nelle librerie di apprendimento (es: keras) esistono varie implementazioni di algoritmi per ottimizzare la discesa del gradiente. Questi algoritmi sono generalmente usati come black-box, in questa sezione forniremo una panoramica e le intuizioni dietro ad essi. Una prima differenziazione della discesa del gradiente è sulla quantità di dati usati per i calcoli. In questa sezione ci riferiamo alla funzione di costo (loss function, error function etc) da minimizzare come $J(\mathbf{x}|\boldsymbol{\theta}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ funzione di $\mathbf{x} \in \mathbb{R}^n$ parametrizzata da $\boldsymbol{\theta} \in \mathbb{R}^d$. Per brevità, visto che minimizziamo rispetto a $\boldsymbol{\theta}$ indichiamo la funzione di costo semplicemente come $J(\boldsymbol{\theta})$.

Batch gradient descent Batch gradient descent calcola il gradiente rispetto ai parametri θ della funzione di costo sull'intero insieme di dati di allenamento (training dataset):

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_{\theta} J(\theta) \quad (2.3)$$

osserviamo che per eseguire un singolo aggiornamento dei parametri (epoch) dobbiamo calcolare il gradiente su tutto il dataset, questo rende l'algoritmo estremamente lento e intrattabile nel caso in cui la dimensione del dataset è maggiore del quantitativo di memoria del calcolatore. Inoltre non permette l'aggiornamento online del modello aggiungendo o modificando gli esempi nel training set durante la fase di allenamento. Batch gradient descent converge sicuramente al minimo globale se la funzione da ottimizzare è convessa (caso ottimo, altamente improbabile nella realtà) altrimenti converge ad un minimo locale o ad un punto di sella (quest'ultimo non voluto). In codice python:

```
for i in range(nb_epochs):
    params_grad = gradient(loss_function, data, params)
    params = params - learning_rate * params_grad
```

Stochastic gradient descent (SGD) al contrario aggiorna i parametri per ogni esempio di allenamento x_i e il relativo output y_i :

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_{\theta} J(x_i, y_i | \theta). \quad (2.4)$$

L'idea alla base di SGD è che batch gradient descent esegue calcoli ridondanti su interi dataset molto grandi, come ricalcolare il gradiente per esempi simili prima di aggiornare i pesi. SGD elimina questa ridondanza aggiornando i pesi ad ogni computazione del gradiente, questo tuttavia porta ad avere un'alta varianza del gradiente, con conseguenti fluttuazioni nella loss function da minimizzare. Tuttavia è stato dimostrato che decrementando lentamente e gradualmente il learning rate, SGD mostra le stesse caratteristiche di convergenza di batch gradient descent, convergendo certamente ad un minimo locale o globale per una funzione non convessa o convessa rispettivamente.

```
for i in range(nb_epochs):
    for example in data:
        params_grad = gradient(loss_function, data, params)
        params = params - learning_rate * params_grad
```

Mini-batch gradient descent prende il meglio dei due metodi aggiornando i pesi per ogni mini batch di dimensione n esempi di training:

$$\theta_{k+1} = \theta_k - \eta \cdot \nabla_{\theta} J(x_{i:i+n}, y_{i:i+n} | \theta). \quad (2.5)$$

In questo modo risulta ridotta la varianza del gradiente e quindi degli aggiornamenti dei pesi con la conseguenza di una maggiore stabilità della convergenza e permette di velocizzare il calcolo rispetto a SGD sfruttando le librerie ottimizzate del calcolo matriciale.

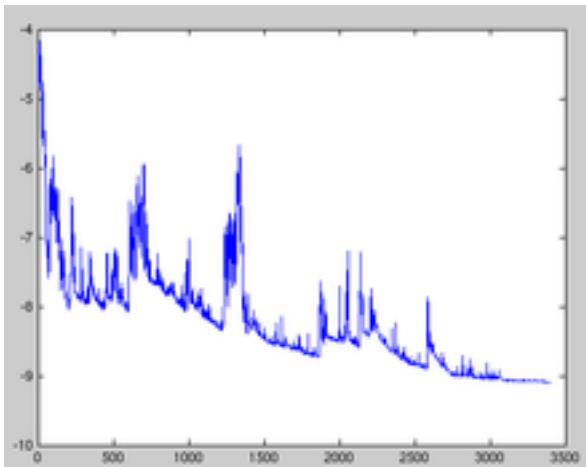


Figure 2.5: Fluttuazioni SGD

```

for i in range(nb_epochs):
    for batch in get_batches(data, batch_size=n):
        params_grad = gradient(loss_function, batch, params)
        params = params - learning_rate * params_grad

```

Sfide. Mini-batch gradient descent non garantisce ancora ottime proprietà di convergenza, ma fornisce delle sfide che devono essere affrontate:

- Scegliere l'adatto learning rate (l.r.) può essere difficile. Un l.r. troppo piccolo porta ad una convergenza dolorosamente lenta, mentre un l.r. troppo alto causa fluttuazioni intorno ad un minimo della loss function o addirittura la divergenza.
- Programmare diversi l.r. durante le fasi di allenamento seguendo schemi predefiniti oppure adattandolo ai risultati intermedi dell'allenamento (learning rate adattivo).
- Usare differenti l.r. contemporaneamente. Se i dati sono sparsi e le features si presentano con differenti frequenze potremmo considerare di utilizzare valori di l.r. elevati quando si presentano le features più rare.
- Uscire dai minimi sub ottimi e punti di sella.

Ora vediamo una rapida carrellata degli algoritmi più noti.

2.3.2.1 Momentum

SGD ha problemi a navigare attraverso i "burroni", per esempio zone in cui la superficie della funzione curva molto più rapidamente rispetto alle altre,

situazione molto comune vicino ai minimi locali. In questo scenario, SGD oscilla lungo le pendenze del burrone e avanza lentamente nella direzione del minimo.

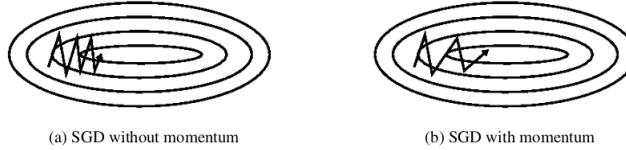


Figure 2.6: Momentum

Momentum è un metodo che aiuta ad accelerare SGD nella direzione rilevante e riduce le oscillazioni come mostrato in Figura 2.6b. Per fare ciò introduce un termine γ nell'aggiornamento del vettore direzione come mostrato:

$$\begin{cases} \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \\ \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t \end{cases} \quad (2.6)$$

Il termine γ comunemente usato è 0.9 o un valore simile. Essendo la direzione seguita al tempo t dipendente dal tempo $t-1$ e quindi ricorsivamente da tutti i tempi precedenti, permette di mantenere una sorta di memoria, inoltre è facile convincersi dalla formula che i cambi di direzioni isolati verranno attenuati mentre la direzione principale, cioè quella che più volte ricorre verrà incrementata. Un' analogia utile a comprendere è immaginare una palla che rotola lungo un sentiero di montagna, la palla tende ad accelerare nella direzione di discesa ma subisce anche altre accelerazioni isolate dovute a parziali intralci nel percorso, come ad esempio un sasso che la fa deviare momentaneamente verso destra, momentum si occupa di smorzare tali deviazioni e incrementare nella direzione di discesa.

2.3.2.2 Nesterov accelerated gradient (NAG).

Questo medoto è un evoluzione di momentum, tornando all'analogia con la palla che rotola giù da un pendio, l'idea è quella di guardarsi attorno prima di calcolare la prossima direzione e cercare di aggirare preventivamente ostacoli e avvallamenti. Notiamo che in momentum calcoliamo preventivamente il vettore $\boldsymbol{\theta}_{t-1} - \gamma \mathbf{v}_{t-1}$ che fornisce un' approssimazione della prossima posizione dei parametri (i parametri sono raggruppati in un vettore, possono quindi essere visti come punti nello spazio). Allora possiamo effettivamente guardarci attorno calcolando il gradiente non nella posizione attuale dei parametri ma rispetto all'approssimazione futura.

$$\begin{cases} \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_{t-1} - \gamma \mathbf{v}_{t-1}) \\ \boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - \mathbf{v}_t \end{cases} \quad (2.7)$$

Ancora, γ comunemente usato è 0.9 o un valore simile.



Figure 2.7: NAG update

In figura 2.7 vediamo un confronto tra Momentum e NAG. Momentum prima calcola il gradiente corrente (vettore blu piccolo) dopo esegue un grande passo nella direzione accumulata nei passi precedenti (vettore blu grande). NAG invece prima esegue un grande passo nella direzione predetta (vettore grande marrone) poi calcola il gradiente e corregge il passo (vettore rosso piccolo). Questo aggiornamento anticipato ci impedisce di accelerare troppo e aumenta la reattività durante l'allenamento.

Adesso siamo capaci di adattare gli aggiornamenti dei pesi alla pendenza della funzione di errore e di accelerare SGD. Il prossimo passo è adattare i nostri aggiornamenti specificamente per ogni peso in modo da eseguire aggiornamenti minori o maggiori a seconda dell'importanza del peso.

2.3.2.3 Adagrad

adatta il learning rate ai parametri, eseguendo aggiornamenti maggiori ai pesi meno frequenti e aggiornamenti minori ai pesi più frequenti. Per questa ragione è adatto nel caso in cui i dati siano sparsi (???? che significa sparsi????). Sia:

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

inoltre definiamo:

$$G_t = \begin{bmatrix} \sum_{j=0}^t \left(\frac{\partial j(\boldsymbol{\theta})}{\partial \theta_1} \right)^2 & 0 & \dots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \sum_{j=0}^t \left(\frac{\partial j(\boldsymbol{\theta})}{\partial \theta_i} \right)^2 & \vdots \\ 0 & \dots & \dots & \dots & \sum_{j=0}^t \left(\frac{\partial j(\boldsymbol{\theta})}{\partial \theta_d} \right)^2 \end{bmatrix}$$

la matrice diagonale appartenente a $\mathbb{R}^{d \times d}$ in cui ogni elemento in posizione i, i è la somma dei quadrati della derivata parziale rispetto al parametro $i - esimo$ dal tempo 0 al tempo t . La regola di aggiornamento diventa:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \quad (2.8)$$

dove \odot è il prodotto elemento per elemento tra matrici. In questo modo risulta evidente che il l.r. risulta inversamente proporzionale agli aggiornamenti

precedenti, più un peso viene aggiornato più il termine corrispondente $G_{t,ii}$ cresce e di conseguenza il learning rate decresce. Il termine ϵ è necessario per evitare divisioni per zero (comunemente nell'ordine di 10^{-8}). I ricercatori inoltre hanno notato che senza l'operatore di radice quadrata l'algoritmo funziona molto peggio, probabilmente dovuta a una rapida decaduta dei coefficienti. Uno dei benefici di Adagrad è l'eliminazione della necessità di modificare manualmente il learning rate. D'altra parte introduce un'altra debolezza, anche estraendo la radice quadrata, a denominatore sommiamo sempre quantità positive, questo alla lunga porta il learning rate a valori infinitesimi, bloccando di fatto l'apprendimento. Il prossimo algoritmo ha lo scopo di risolvere questo problema.

2.3.2.4 Adadelta

è un'estensione di Adagrad che cerca di ridurre l'aggressività con cui monotonicamente decresce il learning rate. Invece di accumulare tutti i quadrati dei precedenti gradienti, Adadelta restringe l'accumulo con una finestra di una fissa dimensione w . Inoltre, invece di salvare inefficientemente tutti i w gradienti al quadrato salva una media decadente dei precedenti gradienti. La media è definita nel modo seguente:

$$E[g^2]_0 = \mathbf{0}$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2$$

con ancora:

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

e γ un termine simile al momento (circa 0.9). Abbiamo affermato precedentemente che Adadelta è l'evoluzione di Adagrad, richiamiamo il vettore aggiornamento di Adagrad:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t$$

adesso sostituiamo semplicemente la matrice diagonale G_t con la media definita:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

notiamo che adesso a denominatore abbiamo un vettore e non una matrice. Definiamo, per brevità di scrittura

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon}$$

dove RMS sta per root mean squared, allora

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$

il prossimo passo consiste nel rendere anche il numeratore dipendente in qualche modo dai parametri, definiamo quindi un' altra media, questa volta una media degli aggiornamenti precedenti:

$$\begin{aligned} E[\Delta\theta^2]_0 &= \mathbf{0} \\ E[\Delta\theta^2]_t &= \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma) \Delta\theta^2. \end{aligned}$$

La root mean squared degli aggiornamenti dei parametri è:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$

Siccome all'aggiornamento al tempo t non possiamo conoscere $RMS[\Delta\theta]_t$, lo approssimiamo usando RMS al tempo precedente. Siamo giunti alla formula finale di Adadelta:

$$\begin{cases} \Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} = \theta_t - \Delta\theta_t \end{cases}. \quad (2.9)$$

2.3.2.5 RMSprop

è molto simile ad Adadelta, infatti i due metodi sono stati sviluppati indipendentemente nello stesso periodo e con lo scopo di risolvere i problemi di Adagrad. L'aggiornamento dei pesi segue questi passi:

$$\begin{aligned} E[g^2]_t &= \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \quad (2.10)$$

gli autori suggeriscono di usare $\gamma = 0.9$ e $\eta = 0.001$.

2.3.2.6 Adam

Adaptive Moment Estimation (Adam) è un altro metodo per adattare il learning rate ad ogni parametro. In aggiunta alla media decadente dei gradienti precedenti al quadrato v_t come Adadelta e RMSprop, Adam mantiene anche una media decadente dei gradienti passati m_t (NB non il quadrato):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

Notiamo che il termine v_t è del tutto analogo a $E[g^2]_t$ di Adadelta e RMSprop. m_t e v_t sono stime del momento primo (media) e momento secondo (varianza) dei gradienti rispettivamente, da qui il nome del metodo. Entrambi i termini vengono inizializzati a 0, gli autori hanno notato però che questa inizializzazione porta un bias che fa tendere l'aggiornamento a 0, soprattutto con valori di β_1, β_2

vicini a 1. Per contrastare questo bias introduciamo una correzione a entrambi i termini:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

L'aggiornamento diventa allora:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (2.11)$$

Gli autori propongono come valori di default: $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\epsilon = 10^{-8}$. Hanno mostrato empiricamente che Adam lavora bene e con prestazioni simili a Adadelta e RMSProp.

2.3.2.7 AdaMax

In Adam calcoliamo $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$, notiamo che v_t è direttamente proporzionale alla norma l_2 del gradiente e quindi l'aggiornamento è inversamente proporzionale alla norma del gradiente. Possiamo generalizzare l'aggiornamento usando l_p norma.

$$\begin{aligned} v_t &= \beta_2 v_{t-1} + (1 - \beta_2) |g_t|^p \\ &= (1 - \beta_2) \sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p. \end{aligned}$$

La norma l_p è numericamente instabile per grandi valori di p , questo è il motivo per cui generalmente si usa l_1, l_2 . Tuttavia la norma l_∞ mostra ancora un comportamento molto stabile, per questo gli autori di AdaMax la propongono. Definiamo $u_t = \lim_{p \rightarrow \infty} (v_t)^{1/p}$ allora:

$$\begin{aligned} u_t &= \lim_{p \rightarrow \infty} (v_t)^{1/p} = \lim_{p \rightarrow \infty} ((1 - \beta_2) \sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \\ &= \lim_{p \rightarrow \infty} (1 - \beta_2)^{1/p} (\sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \\ &= \lim_{p \rightarrow \infty} (\sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \\ &= \max(\beta_2^{t-1} |g_1|, \beta_2^{t-2} |g_2|, \dots, \beta_2 |g_{t-1}|, |g_t|) \end{aligned}$$

che può essere riscritta ricorsivamente come:

$$u_t = \max(\beta_2 u_{t-1}, |g_t|).$$

Al solito, $u_0 = 0$. Sostituiamo nella formula di Adam $\sqrt{\hat{v}_t} + \epsilon$ con u_t , otteniamo così la regola di aggiornamento AdaMax:

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

I valori consigliati di default sono $\eta = 0.002$, $\beta_1 = 0.9$, $\beta_2 = 0.999$.

2.3.2.8 Nadam

Come visto prima, Adam può essere visto come una combinazione di RMSprop e Momentum: RMSprop contribuisce tramite il termine $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ e momentum tramite il termine $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$. Abbiamo visto anche che Nesterov accelerated gradient (NAG) è superiore a momentum. Nadam (Nesterov-accelerated Adaptive Moment Estimation) combina così Adam e NAG. In NAG calcoliamo il gradiente non nella posizione attuale ma nella posizione stimata a priori, in cui arriveremo dopo l'aggiornamento.

$$g_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t - \gamma \boldsymbol{m}_{t-1})$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

Gli autori di Nadam propongono di modificare NAG in questo modo: piuttosto che applicare il momento due volte, una volta per guardarsi attorno nel calcolo del gradiente g_t e una seconda volta nel calcolo di θ_{t+1} , usiamo il momento corrente m_t per guardarci attorno direttamente nell'aggiornamento dei pesi e non nel gradiente, allora NAG modificato diventa:

$$g_t = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t)$$

Richiamiamo brevemente anche il metodo Adam:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Espandiamo l'ultima equazione:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{m_t}{1 - \beta_1^t} \right) \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{1 - \beta_1^t} \right) \\ &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right)\end{aligned}$$

Notiamo che $\frac{m_{t-1}}{1 - \beta_1^t} = \hat{m}_{t-1}$ sostituendo:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right)$$

Notiamo che questa regola è ancora Adam, per giungere a Nadam dobbiamo inserire NAG modificato in precedenza, notiamo che in NAG modificato usiamo il momento corrente nell'aggiornamento e non il momento del passo precedente, modifichiamo di conseguenza la formula di aggiornamento:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right) \quad (2.12)$$

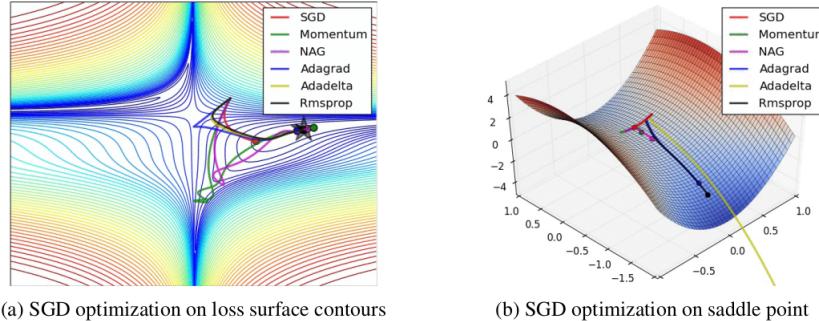


Figure 2.8: Confronto algoritmi

2.4 Altre tecniche di ottimizzazione

2.4.1 Shuffling e Curriculum Learning

Generalmente allenare una rete fornendole gli esempi sempre nello stesso ordine può portare ad un bias nell'algoritmo di ottimizzazione e quindi all'overfitting.

Di conseguenza una buona idea è fornire gli esempi mescolati ad ogni epoca. In altri casi, in cui lo scopo della rete è risolvere via via problemi sempre più complessi, allora è meglio fornire gli esempi in ordine di complessità. Quest'ultimo è il caso del Curriculum learning.

2.4.2 Batch Normalization

è stato mostrato empiricamente che molte volte è utile all'allenamento normalizzare il vettore dei parametri θ portandolo a media nulla e varianza unitaria. Progredendo nell'allenamento aggiornando i pesi tale normalizzazione viene persa, portando ad un decadimento della capacità di apprendimento e lentezza nella convergenza, soprattutto in reti molto profonde, in cui la backpropagation può far divergere i pesi. Batch normalization rinormalizza i pesi per ogni minibatch nello strato in cui è applicata. Normalizzando parte dell'architettura della rete possiamo usare learning rate più elevati e porre meno attenzione sull'inizializzazione dei parametri, inoltre riduce la necessità del Dropout.

2.4.3 Early Stopping

Consiste nel monitorare l'errore durante la fase di validazione (parleremo di validazione quando affronteremo l'overfitting). Quando alleniamo una rete decidiamo a priori per quante epoche allenarla (un'epoca corrisponde a sottoporre alla rete tutto il training set), dopo un certo numero di epoche osserviamo che l'errore di validazione che prima diminuiva inizia ad aumentare, magari oscillando anche, questo è il caso dell'overfitting. Early stopping è metodo parametrizzato da tre fattori, quantità da monitorare, un delta che indica il valore assoluto o percentuale di cambiamento della quantità monitorata per determinare se c'è stato un miglioramento o peggioramento ed infine un parametro patience che determina dopo quante epoche senza miglioramenti fermare l'allenamento.

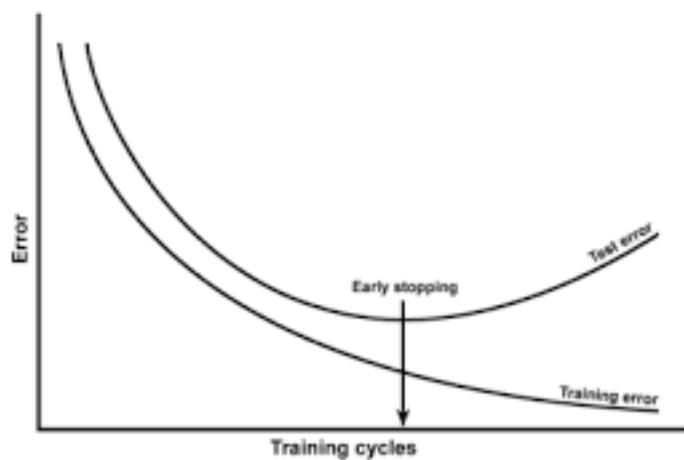


Figure 2.9: Early stopping

Chapter 3

Classificazioni di immagini

La visione artificiale (nota anche come computer vision) è l'insieme dei processi che mirano a creare un modello approssimato del mondo reale (3D) partendo da immagini bidimensionali (2D). Vedere è inteso non solo come l'acquisizione di una fotografia bidimensionale di un'area ma soprattutto come l'interpretazione del contenuto di quell'area. L'informazione è intesa in questo caso come qualcosa che implica una decisione automatica. Un problema classico nella visione artificiale è quello di determinare se l'immagine contiene o no determinati oggetti (Object recognition) o attività. Il problema può essere risolto efficacemente e senza difficoltà per oggetti specifici in situazioni specifiche per esempio il riconoscimento di specifici oggetti geometrici come poliedri, riconoscimento di volti o caratteri scritti a mano. Le cose si complicano nel caso di oggetti arbitrari in situazioni arbitrarie.

Nella letteratura troviamo differenti varietà del problema:

- Recognition (riconoscimento): uno o più oggetti prespecificati o memorizzati possono essere ricondotti a classi generiche usualmente insieme alla loro posizione 2D o 3D nella scena.
- Identification (identificazione): viene individuata un'istanza specifica di una classe. Es. Identificazione di un volto, impronta digitale o veicolo specifico.
- Detection (rilevamento): l'immagine è scandita fino all'individuazione di una condizione specifica. Es. Individuazione di possibili cellule anormali o tessuti nelle immagini mediche.

Altro compito tipico è la ricostruzione dello scenario: dati 2 o più immagini 2D si tenta di ricostruire un modello 3D dello scenario. Nel caso più semplice si parla di un insieme di singoli punti in uno spazio 3D o intere superfici. Generalmente è importante trovare la matrice fondamentale che rappresenta i punti comuni provenienti da immagini differenti.

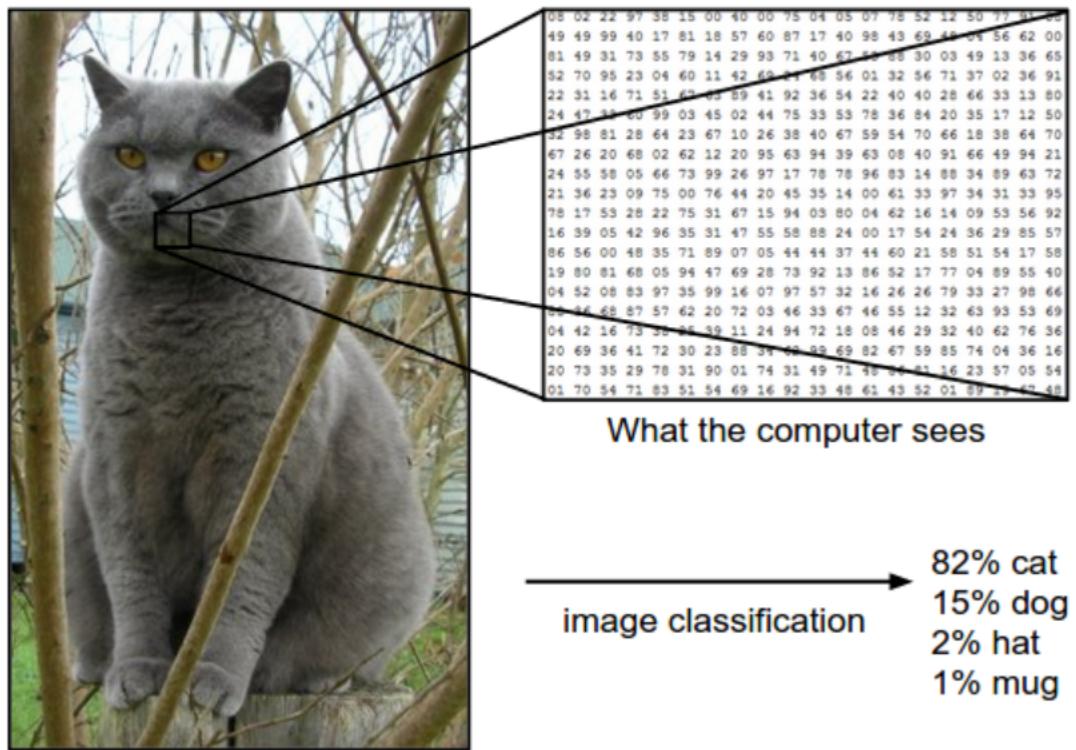


Figure 3.1: Classificazione di un'immagine

Il problema della **classificazione di immagini** è il compito di assegnare ad un'immagine di input una e una sola etichetta proveniente da un insieme fissato di output. La classificazione presenta alcuni problemi:

- Variazione punto di vista (Viewpoint variation): una singola istanza di un oggetto può essere orientata in modi diversi rispetto alla camera, producendo immagini diverse;
- Variazione scala (Scale variation): oggetti diversi appartenenti alla medesima classe possono differire nelle loro dimensioni reali;
- Deformazione (Deformation): alcuni oggetti non sono corpi rigidi e possono apparire deformati in modi diversi;
- Occlusione (Occlusion): parti dell'oggetto da riconoscere è nascosto e non visibile;
- Condizioni di illuminazione (Illumination conditions): l'illuminazione ha un ruolo decisivo nell'informazione codificata all'interno dei pixel che compongono l'immagine;

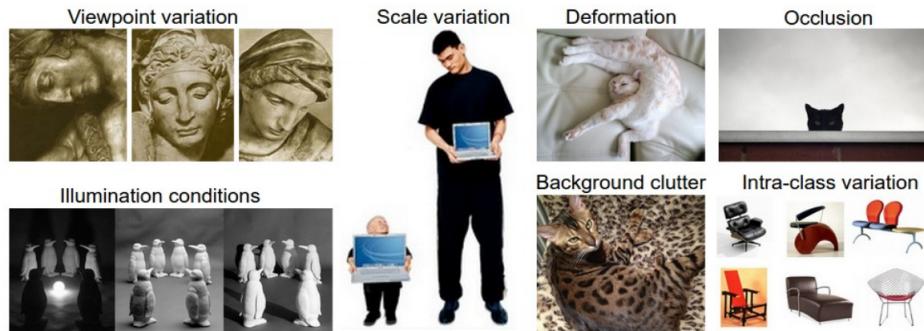


Figure 3.2: Problemi nella classificazione di immagini

- Disordine di sfondo (Background clutter): gli oggetti di interesse possono mescolarsi e confondersi nell'ambiente circostante, rendendo difficile l'identificazione;
- Variazione intra-classe (Intra-class variation): oggetti appartenenti alla stessa classe possono differire significativamente l'uno dall'altro.

3.1 Dataset CIFAR-10

In queste note si farà riferimento al dataset CIFAR-10 che consiste in 60000 immagini di dimensione 32×32 pixels, ogni pixel ha associato 3 numeri, uno per ogni colore (RGB). Ogni immagine è etichettata con una di 10 classi, Queste 60000 immagini sono partizionate in *training set* di 50000 immagini e *test set* di 10000 immagini. Ogni immagine può quindi essere vista come un vettore appartenente allo spazio $\mathbb{R}^{32 \times 32 \times 3}$.

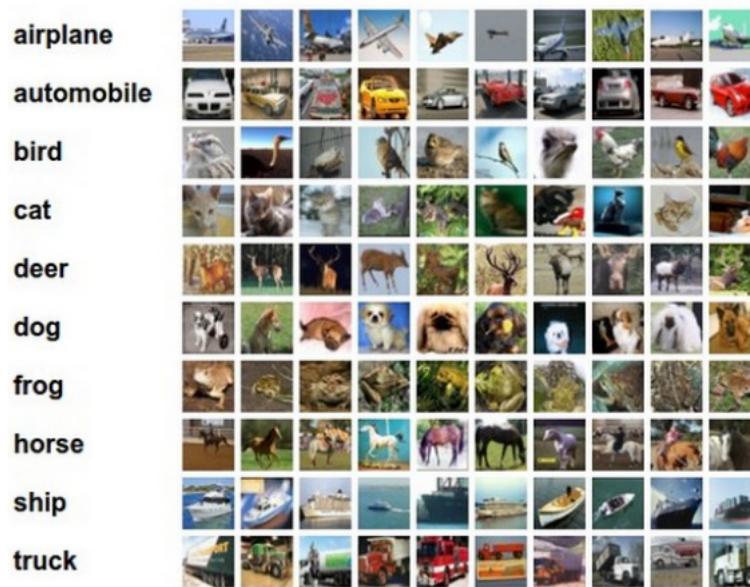


Figure 3.3: CIFAR-10 dataset

3.2 Nearest Neighbor Classifier

Questo classificatore non ha nulla a che fare con le reti neurali e non viene quasi mai usato nella pratica, ma ci permetterà di avere un'idea dell'approccio di base a un problema di classificazione delle immagini. L'idea di questo classificatore è molto semplice, la fase di training consiste nel memorizzare tutte le immagini del training set, la classificazione avviene confrontando l'immagine di test con ogni immagine del training set, quindi si etichetta l'immagine in accordo con l'etichetta dell'immagine che più assomiglia. Cosa intendiamo quando diciamo "*che più assomiglia*"? Occorre quindi formalizzare questo concetto secondo una qualche metrica. Ogni immagine può essere vista come una matrice in cui ogni elemento è un valore numerico che rappresenta l'intensità di un colore appartenente allo spazio RGB di un singolo pixel. Definiamo quindi tre distanze:

1. **L1 distance:**

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

la distanza è calcolata sommando il modulo delle differenze elemento per elemento. è anche nota come distanza di Manhattan.

2. **L2 distance:**

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

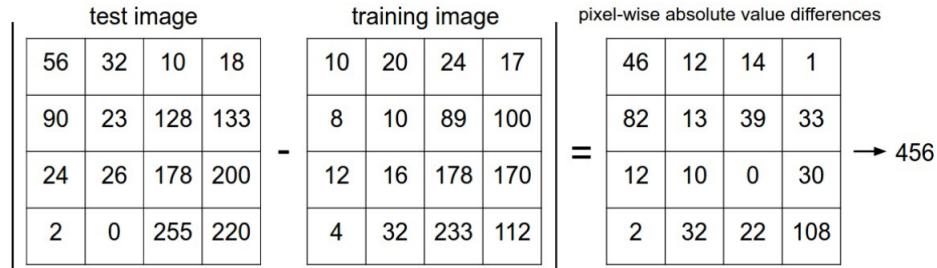


Figure 3.4: L1 distance

è la distanza euclidea classica.

3. L_k distance:

$$d_k(I_1, I_2) = \left(\sum_p (|I_1 - I_2|)^k \right)^{\frac{1}{k}}$$

con $k \in [1, +\infty)$. E' una generalizzazione delle precedenti.

Le distanze comunemente usate sono le prime due, L1 e L2. In altre parole Nearest Neihbor Classifier è ricondotto ad un problema di minimizzazione riformulando il problema di classificazione come segue:

- chiamiamo \mathbf{x}_i l'i-esima immagine di test (da etichettare);
- chiamiamo \mathbf{x}_j la j-esima immagine del training set (già etichettata);
- chiamiamo y_j l'etichetta della j-esima immagine del training set;
- poniamo $y_i = y_{j^*}$ con $j^* = \operatorname{argmin}(d(\mathbf{x}_i, \mathbf{x}_j))$.

Da test effettuati risulta che Nearest Neihbor Classifier, usando la distanza L1, ha classificato correttamente il 38,6% delle immagini del dataset CIFAR-10. Un risultato ragguardevole se comparato alla probabilità di una corretta classificazione assegnando casualmente un'etichetta (10% nel nostro caso) ma ben lontano dalle prestazioni umane e dalle migliori reti neurali convoluzionali. Utilizzando la distanza L2 invece si è ottenuto un'accuratezza del 35,4%.

3.3 K-Nearest Neighbor Classifier

Questo classificatore è un'estensione del precedente e si basa su un'idea molto semplice: invece di assegnare l'etichetta dell'immagine più vicina (rispetto ad una definita distanza), troviamo le k immagini più vicine e assegnamo l'etichetta che compare maggiormente nelle k etichette. In particolare per $k = 1$ otteniamo Nearest Neihbor Classifier precedente. Intuitivamente un alto valore di k ha un

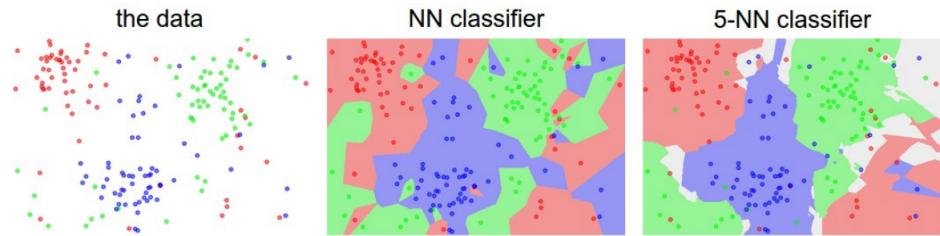


Figure 3.5: confronto NN classifier e 5-NN classifier

effetto "levigante" sui confini decisionali e rende il classificatore più resistente ai valori anomali.

La figura 2.5 mostra un confronto tra K-NN e NN, usando punti bidimensionali come dati da classificare in 3 classi (rosso, blu, verde). Le regioni colorate evidenziano i confini decisionali. Le regioni bianche mostrano punti la cui classificazione è ambigua (per esempio in K-NN nel caso in cui ci sia parità tra due o più classi). Notiamo come nel caso di punti anomali, NN crea piccole isole di probabili previsioni errate, mentre 5-NN smussa queste irregolarità.

Vantaggi e svantaggi K-NN Tra i vantaggi si sottolinea:

- Facile da capire e implementare
- Training in tempo costante $O(1)$ difatti basta salvare il riferimento al training set.

Tra gli svantaggi si sottolinea:

- Complessità temporale: richiede il confronto di ogni immagine di test con tutte quelle appartenenti al trainin set. La complessità dipende linearmente dalla dimensione del training set e test set;
- Elevata complessità spaziale: richiede che tutto il training set sia memorizzato.
- La distanza tra immagini non è sempre significativa come mostrato dalla seguente immagine.

To-do parlare
di overfitting,
cross-
validation,
hyperparameter

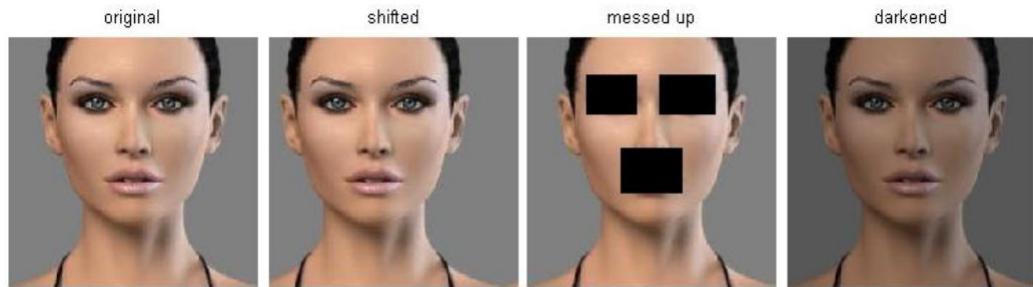


Figure 3.6: Distanza tra immagini, le 3 immagini a sinistra hanno tutte la stessa distanza dall'originale

si pone ora il problema di come scegliere il valore k , come scegliere il tipo di distanza da usare. Soluzione **cross-validation**.

To-Do

3.4 Classificatore Lineare (Linear Classifier)

Un classificatore può anche essere visto come una funzione che associa ad un'immagine \mathbf{x} un vettore le cui componenti sono il punteggio associato ad ogni classe. Intuitivamente un buon classificatore associa alla classe corretta un punteggio più alto rispetto alle classi incorrecte. Più formalmente:

$$\mathbf{F} : \mathbb{R}^d \rightarrow \mathbb{R}^L \quad (3.1)$$

dove d è la dimensione di una immagine ($32 \times 32 \times 3$ nel caso CIFAR-10), L è la cardinalità dell'insieme delle etichette. Quindi $\mathbf{F}(\mathbf{x})$ è un vettore L -dimensionale e la i -esima componente $s_i = [\mathbf{F}(\mathbf{x})]_i$ contiene il punteggio di quanto probabile sia l'appartenenza di \mathbf{x} alla classe i . Per il momento non abbiamo detto nulla su come è fatta la funzione vettoriale $\mathbf{F} : \mathbb{R}^d \rightarrow \mathbb{R}^L$. Nel classificatore lineare \mathbf{F} è una funzione lineare:

$$\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b \quad (3.2)$$

dove $W \in \mathbb{R}^{L \times d}$ è chiamata *matrice dei pesi*, $b \in \mathbb{R}^L$ è chiamato *bias* e sono entrambi parametri. Prima di essere classificata un'immagine necessita di una pre elaborazione, denominata *unroll* che consiste nel espandere la matrice di pixel $I \in \mathbb{R}^{h \times w}$ in un vettore $\mathbf{x} \in \mathbb{R}^{(h \cdot w \cdot 3)}$ in cui ogni componente del vettore rappresenta l'intensità di uno specifico colore RGB di un singolo pixel.

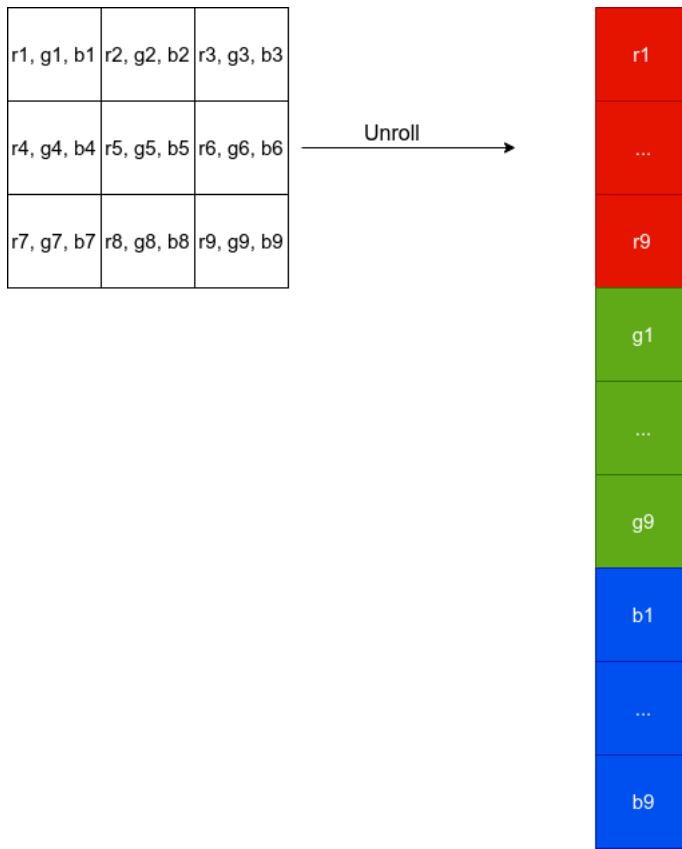


Figure 3.7: Unroll immagine

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline
 -8.1 & \dots & 2.7 & 9.5 & \dots & -9.0 & -5.4 & \dots & 4.8 \\ \hline
 9.0 & \dots & 5.4 & 4.8 & \dots & 1.2 & 9.5 & \dots & -8.0 \\ \hline
 1.2 & \dots & 9.5 & -8.0 & \dots & 8.1 & -2.7 & \dots & 9.5 \\ \hline
 \end{array} \quad W \quad * \quad \begin{array}{|c|} \hline 23 \\ \hline \dots \\ \hline 21 \\ \hline 34 \\ \hline \dots \\ \hline 12 \\ \hline 34 \\ \hline \dots \\ \hline 23 \\ \hline \end{array} \quad + \quad \begin{array}{|c|} \hline -2 \\ \hline 32 \\ \hline -1 \\ \hline \end{array} \quad = \quad \begin{array}{|c|} \hline -4 \\ \hline 22 \\ \hline 33 \\ \hline \end{array} \quad \begin{array}{l} s_1 \text{ dog score} \\ s_2 \text{ cat score} \\ s_3 \text{ rabbit score} \end{array}$$

\mathbf{x}_i

Convolutional Neural Networks for Visual Recognition <http://cs231n.github.io/>

Figure 3.8: Linear Classifier

Il classificatore lineare assegna ad un'immagine di input la classe corrispondente al più alto punteggio:

$$\hat{y}_j = \arg \max_{i=1, \dots, L} [s_j]_i.$$

Nel caso CIFAR-10 ogni immagine viene trasformata in un vettore di $[3072 \times 1]$, la matrice W ha una dimensione di $[10 \times 3072]$, b ha dimensione $[10 \times 1]$.

- Notiamo che una singola moltiplicazione $W\mathbf{x}_i$ corrisponde (nel caso specifico) a 10 classificazioni parallele, in cui ogni riga della matrice W corrisponde a un classificatore, quindi all'importanza che ogni pixel possiede per la i -esima classe.
- Durante la fase di training abbiamo i dati di input (\mathbf{x}_i, y_i) già classificati e fissati, ma noi abbiamo il controllo sui parametri W, b e il nostro obiettivo è trovare quei lavori che massimizzano la classificazione corretta.
- Rispetto a Nearest Neighbour Classifier una volta terminata la fase di training il set di immagini di training può essere eliminato, è necessario solo memorizzare i parametri W, b .
- Inoltre questo classificatore lineare coinvolge solo un prodotto matriciale che è molto più veloce del confronto di un'immagine con tutto il training set.

Loss Function Dopo aver visto formalmente cosa è un Linear Classifier, cioè una funzione lineare $\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b$, ci poniamo ora il problema di trovare i giusti parametri W, b che rendano consistente e migliore la classificazione. Per fare ciò introduciamo una funzione in grado di fornirci una misura di quanto la nostra classificazione sia «infelice» (unhappiness), cioè lontana da un risultato corretto. Si tratterà quindi di una funzione dell'input \mathbf{x} e parametrizzata dai parametri W, b che andrà minimizzata. In letteratura è nota come **loss function** (o altre volte **cost function**).

$$\mathcal{L}(\mathbf{x}, y_i | W, b) \quad (3.3)$$

Bias trick Modifichiamo ora la score function alleggerendo la notazione eliminando il termine b . Richiamiamo la (9) $\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b$, come evidente risulta scomodo mantenere due set di parametri (il bias b e i pesi W) separati. L'idea è di includere il vettore di bias nella matrice W . Sia:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

allora la (9) diventa:

$$\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad (3.4)$$

dove s_i rappresenta il punteggio della classe i -esima. Dalla definizione di prodotto riga per colonna notiamo che la (11) è equivalente a:

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & b_1 \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} & b_m \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}$$

dove abbiamo esteso la matrice W aggiungendo una colonna contenente \mathbf{b} e esteso il vettore \mathbf{x} aggiungendo 1 in posizione $(n+1)$. Rinominando i componenti della matrice W giungiamo alla formula definitiva:

$$\mathbf{F}(\mathbf{x} | W) = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & w_{1,n+1} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & w_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} & w_{m,n+1} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad (3.5)$$

nel nostro esempio CIFAR-10, \mathbf{x} è adesso $[3073 \times 1]$ invece di $[3072 \times 1]$ e W è $[10 \times 3073]$ invece di $[10 \times 3072]$.

3.4.1 Interpretazione di un Classificatore Lineare.

Linear classifier, data in input una immagine (nel caso più generale dato un generico vettore di dati) calcola il punteggio di una classe come una somma pesata di ogni pixel attraverso tutti e tre i canali colori (RGB). Ogni riga della matrice W contiene i pesi per mappare i tre canali colore di ogni pixel nel punteggio di una classe (una riga per classe), quindi in definitiva la funzione lineare ha la capacità di approvare o disapprovare (dipendente dal segno di ciascun $w_{i,j}$) un certo colore in una certa posizione. Per esempio, generalmente, un'immagine della classe "ship" ci aspettiamo abbia un'alta presenza di colore blu ai lati dell'immagine, mentre abbia molto meno altri colori.

3.4.1.1 Interpretazione Geometrica

Poichè trattiamo le nostre immagini di input come vettori colonna possiamo considerare ogni immagine come un punto nello spazio, nell'esempio CIFAR-10 lo spazio è \mathbb{R}^{3072} (escludendo il bias trick). Inoltre dall'algebra lineare sappiamo che l'equazione generica di un iperpiano (sottospazio affine di \mathbb{R}^n di dimensione $n-1$) è $\Sigma : a_1x_1 + a_2x_2 + \dots + a_nx_n + b = 0$ che riscritto in forma più compatta: $\Sigma : \langle \mathbf{a}, \mathbf{x} \rangle + b = 0$ dove $\langle \cdot, \cdot \rangle$ è il prodotto scalare. Osserviamo che il prodotto $W\mathbf{x}$ corrisponde a eseguire m prodotti scalari, quindi $\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b = 0$ definisce m iperpiani nello spazio. Ricordando ora che la distanza di un generico punto \mathbf{P} da un iperpiano Σ è:

$$d(\Sigma, \mathbf{P}) = \frac{|\langle \mathbf{a}, \mathbf{P} \rangle + b|}{\|\mathbf{a}\|}$$

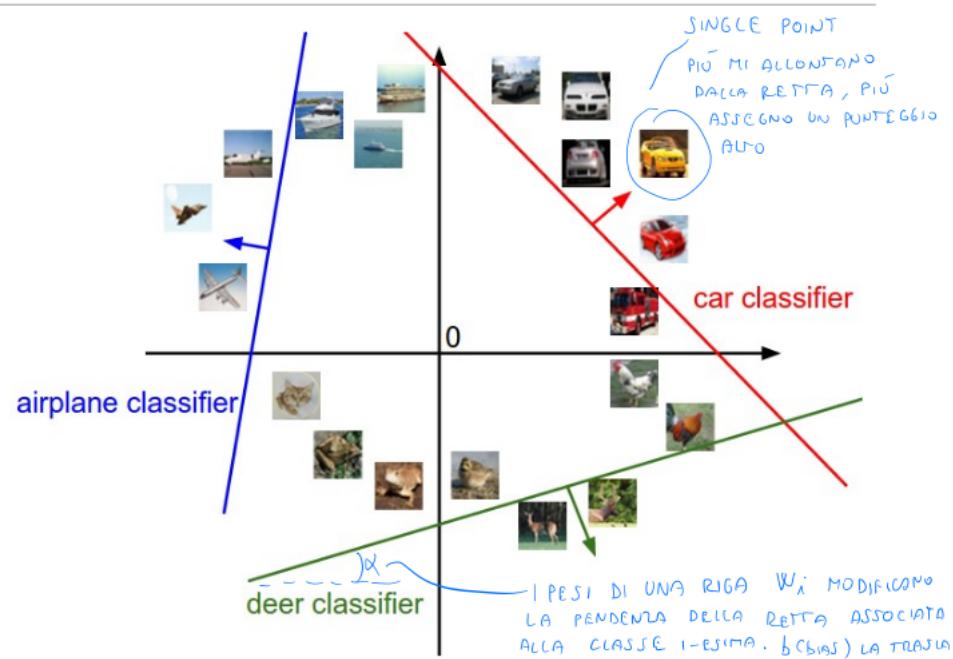


Figure 3.9: Interpretazione geomtrica nello spazio bidimensionale

osserviamo che la distanza è direttamente proporzionale al prodotto scalare dei coefficienti dell'iperpiano con il punto (in modulo, il segno determina se siamo *sopra* o *sotto* l'iperpiano). Possiamo vedere ogni iperpiano come il confine di una regione di accettazione, il punteggio s_i esprime tramite il segno se siamo nella regione di accettazione o no, e tramite il modulo quanto siamo lontani dal confine, quindi un alto punteggio positivo indica un'alta confidenza che quella immagine appartenga alla classe i -esima, viceversa un alto punteggio negativo indica una bassissima confidenza.

3.4.1.2 Template

Un'altra possibile interpretazione per i pesi W è che ogni riga della matrice corrisponda a un template (o prototipo) per ogni classe. Il punteggio di ogni classe è ottenuto tramite prodotto scalare e indica il grado di somiglianza tra l'immagine test e il template.

3.4.1.3 Coming Soon

Fino ad ora non ci siamo posti il problema di come trovare il giusto set di pesi W , è stato evitato di trattare questo argomento perché rientra in un discorso più generale che riguarda le reti neurali in generale, non solo applicate al riconoscimento di immagini. Tratteremo approfonditamente Loss Function,

eror function e tecniche di ottimizzazione, infine torneremo al problema di classificazione delle immagini con questi nuovi strumenti e ne vedremo un'applicazione.

Bibliography

- [1] Lu, Z., Pu, H., Wang, F., Hu, Z., & Wang, L. (2017). The Expressive Power of Neural Networks: A View from the Width. *Neural Information Processing Systems*, 6231-6239.