UNIVERSITÀ DEGLI STUDI "ROMA TRE"
Dipartimento di Matematica e Fisica
Corso di Laurea Magistrale in Scienze Computazionali

Tesi di Laurea Magistrale

Ricerca della topologia ottimale di un sistema di deep learning per identificazioni di oggetti architettonici

Candidato
Désirée Adiutori

Relatore
Prof. Luciano Teresi

Correlatore
Prof. Roberto D'Autilia

Anno Accademico 2017/2018

Indice

Introduzione

Dall'invenzione dei computer, l'uomo fa sempre più affidamento sulle macchine per risolvere problemi complessi di calcolo. Con l'aumentare delle prestazioni dei computer, man mano si sono sviluppati algoritmi di calcolo sempre più efficienti. Nel 1959 l'ingegnere del MIT, Arthur Samuel coniò il termine *"machine learning"*, descrivendo l'apprendimento automatico come un "campo di studio che dà ai computer la possibilità di apprendere senza essere programmati esplicitamente per farlo".[1]
Definiamo l'apprendimento automatico come un insieme di metodi in grado di rilevare automaticamente i modelli tramite dei dati e quindi utilizzare i modelli scoperti per prevedere i dati futuri o per eseguire altri tipi di processi decisionali in condizioni di incertezza. L'insegnamento alla macchina è, pertanto, imprescindibile dai dati. Generalmente, più dati si passano alla macchina, più può imparare. Per questo motivo con l'avvento di Internet, dagli anni '90 ad oggi il tema del "machine learning" è diventato sempre più attuale, la mole di dati reperibile dal web è cospicuo e ha permesso che questo campo sia esponenzialmente progredito.
Il *"deep learning"* è un tipo particolare di machine learning, che riguarda l'emulazione di come gli esseri umani apprendono. Esso affronta i problemi del machine learning, rappresentando il mondo come una gerarchia di concetti annidati: ogni concetto è definito in relazione a concetti più semplici e le rappresentazioni astratte vengono calcolate in termini di concetti meno astratti. Il Deep Learning implica l'utilizzo di reti neurali artificiali (*deep artificial neural networks*) algoritmi e sistemi computazionali, ispirati al cervello umano, per affrontare i problemi del Machine Learning.
L'analogia di Shehzad Noor Taus Priyo può aiutare a capire meglio cosa siano le reti neurali:
"Immaginiamole come una serie di porte da passare, dove l'input è l'uomo che le deve oltrepassare e ogni volta che lo fa cambia qualcosa nel suo comportamento finché, all'ultima porta oltrepassata, l'uomo è diventato una persona del tutto differente, rappresentando l'output di questo processo."[2]
Questa tesi si focalizza su un problema particolare di machine learning: la Classificazione (*Classification*) in particolar modo della classificazione di immagini. L'obiettivo principale è trovare un'architettura ottimale per l'algoritmo che identifica le immagini di oggetti architettonici, inserendo tra i parametri anche le coordinate geofisiche dell'oggetto. Per fare questo bisogna trovare la giusta topologia, la giusta profondità e la giusta larghezza di ogni livello della rete neurale, in due parole: l'architettura ottimale.
Nel primo capitolo si descrive cosa sono e come sono strutturati gli algoritmi di apprendimento. Nel secondo si introduce il concetto di rete neurale,

ponendo particolare attenzione sulle reti neurali di tipo convoluzionale. Nel terzo capitolo si approfondiscono gli algoritmi di apprendimento dedicati alla Classificazione. Nel quarto si descrive il linguaggio di programmazione TensorFlow di Python, mostrando alcuni esempi concreti di programmazione per la classificazione di oggetti; in particolar modo viene mostrato il codice utilizzato per la classificazione di oggetti architettonici e quello per la realizzazione di un App, per dispositivo Android, che lo implementi (prevalentemente codice Java). Infine nell'ultima parte vengono mostrati i risultati ottenuti.

Capitolo 1

Algoritmi di apprendimento

Gli algoritmi di machine learning sono solitamente divisi in tre tipi principali:

- Supervised learning (apprendimento supervisionato)
- Unsupervised learning (apprendimento non supervisionato)
- Reinforcement learning (apprendimento per rinforzo)

Quali usare? Perché sceglierne uno piuttosto che un altro?
La scelta dell'algoritmo da utilizzare dipende dal tipo di dati di cui si dispone. Ma la scelta finale va fatta solo esclusivamente dopo aver testato l'algoritmo, e si sceglie in base a quello più performante: un insieme di ipotesi che funziona bene in un dominio, potrebbe funzionare male in un altro.
Teorema del No Free Lunch [3, Wolper,1996]
Non esiste una definizione universale di algoritmo "migliore".

1.1 Costruzione di un algoritmo

Per costruire un algoritmo di apprendimento bisogna avere:

- processi(*task*): compiti che l'algoritmo deve eseguire;
- misuratori di rendiment: rilevatori delle caratteristiche dei processi;
- esperienze: quantitá di dati dal quale imparare.

I processi di apprendimento automatico descrivono come il sistema dovrebbe elaborare un esempio.

Definizione 1 *Un **esempio** è una raccolta di caratteristiche che sono state misurate quantitativamente da alcuni oggetti o eventi elaborati.*

Di solito, un esempio viene rappresentato da un vettore $x \in \mathbb{R}^n$, dove ogni elemento x_i rappresenta una caratteristica.
Ad esempio, se si considera un fiore: le caratteristiche che lo descrivono sono la lunghezza e la larghezza dei suoi petali e il colore. Quindi in questo caso la metrica usata è la distanza euclidea tra le due estremità del petalo:
se si considera $x = (x_1, x_2, x_3)$ il fiore con queste 3 caratteristiche, si ha che:

$$x_1 = d(h_{min}, h_{max}) \qquad x_2 = d(b_{min}, b_{max}) \qquad x_3 = stringa$$

dove h_{min} e h_{max} rappresentano i due punti relativi alle estremità del petalo e $d : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$ t.c $d(x, y) = \sqrt{(x-y)^2} = |x-y|$.
Dato un processo si cerca di capire quale sia la caratteristica principale, sulla quale si deve misurare il suo rendimento. Infine, dobbiamo dare all'algoritmo un'esperienza sulla quale apprendere, che è quella che lo classificherà in uno dei tre tipi principali. Questa esperienza l'apprende dai *dataset*: una collezione di esempi.
I dataset possono essere di vari tipi:

- di addestramento (*training set*)
- di prova (*test set*)
- di validazione (*validation set*)

L' **insieme di addestramento** è una parte dell'insieme di dati che vengono utilizzati per addestrare un sistema di apprendimento supervisionato.
Da questo insieme, l'algoritmo deve costruire una funzione che capisca, dai parametri, quali caratteristiche descrivono le varie categorie.
L' **insieme di prova** è un insieme di dati che, con l'insieme di addestramento, forma una partizione del dataset di partenza. Questi nuovi dati vengono utilizzati per valutare l'apprendimento dell'algoritmo "addestrato".
L' **insieme di validazione** è usato in maniera analoga all'insieme di prova, ma dei dati inseriti per testare l'algoritmo già si conosce la risposta (una parte di essi può far parte dell'insieme di addestramento) e da questa si valuta se l'output ottenuto è ottimale o meno.
Questi tre insiemi possono coesistere e la scelta delle loro cardinalità non è universale: dipende dal tipo di problema che viene affrontato.
Vediamo ora come valutare l'efficienza di un algoritmo:

Definizione 2 *L'**errore di allenamento**(training error) è una misura di errore che si può calcolare sul set di allenamento. Indica quanto l'algortimo sta apprendendo.*

Definizione 3 *La **generalizzazione** è la capacità di un algoritmo di essere ottimale in seguito ad un input proveniente dall' insieme di prova.*

Definizione 4 *L'**errore di generalizzazione**(generalization error) è una misura di errore che si può calcolare sull'insieme di prova. Verifica se l'algoritmo ha imparato o solo memorizzato. Esso viene detto anche errore di test (test error).*

Si ipotizza che tutti gli esempi siano eventi indipendenti e che tuti gli insiemi, in cui si partiziona l'insieme di dati, hanno la stessa distribuzione di probabilità uniforme.

Definizione 5 *Una **funzione di perdita** (loss function)*

$$L(y, \hat{y}) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$$

è una funzione che misura la distanza (o l'errore) tra i valori di output previsto $\hat{y}$ e i valori effettivi y.

Si possono usare varie misure, ad esempio l'errore quadratico medio(MSE):

$$L(y, \hat{y})_{train} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_{(train)} - y_{(train)})_i^2 \tag{1.1}$$

La funzione di predizione dipenderà da dei parametri, rappresentati da un vettore w, lo scopo è di minimizzare l'errore di allenamento variando w. In base al tipo di apprendimento e al problema da affrontare, verranno usati vari algoritmi per risolvere problemi di minimizzazione libera. Gli algoritmi che risolverono il problema:

$$f(x^*) = \min_{x \in \mathbb{R}^n} f(x), \qquad f \in C^2(\mathbb{R}^n, \mathbb{R}) \tag{1.2}$$

Per risolvere problemi di questo tipo, spesso viene utilizzato il metodo di discesa del gradiente [4].
La struttura generale di un metodo di discesa iterativo di minimizzazione è:

$$x_{k+1} = x_k + \beta_k d_k \tag{1.3}$$

dove x_k rappresenta il valore della x al k-esimo passo. Si parte dal valore x_0 assegnato, $\beta_k \in \mathbb{R}^+$ è il passo e $d_k \in \mathbb{R}^n$ è la direzione lungo la quale ci si muove che, essendo di discesa, sarà tale che: $(d_k, \nabla f(x_k)) < 0$. Sia il passo che la direzione vanno scelti opportunamente ad ogni iterazione, in modo che

$$f(x_{k+1}) < f(x_k)$$

Per ogni passo imponiamo:

$$f(x_k + \beta_k d_k) = \min_{\beta}\{f(x_k + \beta d_k)\} \qquad (strategia\ di\ ricerca\ esatta)$$

Mentre per la direzione: $d_k = -\nabla f(x_k)$. La derivata direzionale di f nella direzione d_k vale

$$\frac{\partial f}{\partial d_k}(x_k) = \frac{(d_k, \nabla f(x_k))}{\parallel d_k \parallel} = - \parallel \nabla f(x_k) \parallel$$

e per la disuguaglianza di Cauchy-Schwartz si ha anche:

$$\frac{\mid (d_k, \nabla f(x_k)) \mid}{\parallel d_k \parallel} \leq \frac{\parallel d_k \parallel\parallel \nabla f(x_k) \parallel}{\parallel d_k \parallel} = \parallel \nabla f(x_k) \parallel$$

che mostra come la direzione di ricerca sia quella in cui la derivata direzionale di f è negativa e di modulo massimo.
Le condizioni di arresto del metodo sono:

$$\parallel x_{k+1} - xk \parallel \leq m, \qquad \parallel \nabla f(x_{k+1}) \parallel \leq m\prime$$

oppure:

$$k > k_{max}$$

dove m e m' sono soglie date e k_{max} il numero massimo di iterazioni da effettuare.
I risultati ottenuti sono garantiti dai seguenti teoremi di convergenza:

Teorema 1 *Sia $f(x) \in C^1$, strettamente convessa sull'insieme $\Sigma_0 = \{x \in \mathbb{R}^n : f(x) \leq f(x_0)\}$, e la successione $\{x_k\}$ sia generata tramite l'algoritmo (1.3). Si supponga:*

1. *che l'insieme Σ_0 sia compatto;*
2. *che le direzioni d_k siano t.c. $\frac{(d_k, \nabla f(x_k))}{\|d_k\|\|\nabla f(x_k)\|} \leq -\cos\theta$ per $k \in I$, con I insieme illimitato di indici;*
3. *che per $k \in I$, β_k sia ottenuto tramite ricerca esatta.*

Allora la successione $\{x_k\}$ converge all'unico punto x^* di minimo per f.

Teorema 2 *Sia $f(x) \in C^2$, strettamente convessa sull'insieme (che si suppone compatto) $\Sigma_0 = \{x \in \mathbb{R}^n : f(x) \leq f(x_0)\}$, e la successione $\{x_k\}$ sia generata tramite l'algoritmo (1.3) con $d_k = -\nabla f(x_k)$.*
Allora, se i passi β_k sono determinati tramite ricerca esatta, la successione x_k converge all'unico punto x^* di minimo per f.

Minimizzare l'errore di allenamento non necessariamente comporta l'ottimizzazione di apprendimento dell'algoritmo, potrebbe verificarsi il fenomeno di adattamento insufficiente (*underfitting*) ovvero non si hanno abbastanza dati per creare un modello di predizione accurato. Bisogna quindi valutare anche altri fattori: analizzare l'insieme di prova.
Ricordandoci dell'eq.1.1 calcoliamo l'errore di prova:

$$L(y, \hat{y})_{test} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_{(test)} - y_{(test)})_i^2 \tag{1.4}$$

Vorremmo che, con i parametri trovati per minimizzare l'errore di allenamento, anche questo errore sia minimo (l'ottimalità è 0). Ma come detto in precedenza non sempre questo accade, vorremmo quindi che il divario tra i due errori sia minimo. In caso contrario si verifica il fenomeno di adattamento eccessivo (*overfitting*) del modello all'insieme di dati che descrive, tramite un eccessivo numero di parametri. Il modello quindi non sarà generalizzabile ad un nuovo insieme di dati.
Consideriamo il valore atteso dell'errore di prova, calcolato prendendo una coppia di punti (X, Y) dall'insieme di prova:

$$\mathbb{E}[L(y, \hat{y})_{test}] = \mathbb{E}[(Y - \hat{y}(X))^2] \tag{1.5}$$

e definiamo la funzione dell'output effettivo come:

$$y(X) = \mathbb{E}(Y|X)$$

la quale avrà sicuramente un errore, dovuto da qualche interferenza,
che chiameremo: distorsione stimata (*estimation bias*).
Ma con diversi insiemi di allenamento, possiamo costruire diverse funzioni $\hat{y}$, e anche questo è un'altra fonte di errore: la varianza stimata (*estimation variance*). Possiamo quindi scrivere l'output come:

$$Y = y(X) + \epsilon$$

con ϵ variabile aleatoria indipendente da X , con X distibuzione normale tale che: $\mathbb{E}[X] = 0$ e $Var(X) = \sigma^2$.
Possiamo quindi riscrivere l'equazione 1.5 come:

$$\begin{aligned} \mathbb{E}[L(y, \hat{y})_{test}] &= \mathbb{E}[(Y - \hat{y}(X))^2 | X = x] \\ &= \mathbb{E}[(Y - y(x))^2 | X = x] + \mathbb{E}[(y(x) - \hat{y}(x))^2 | X = x] \\ &= \sigma^2 + \mathbb{E}[(y(x) - \hat{y}(x))^2] \end{aligned} \tag{1.6}$$

dove σ^2 è chiamato errore Bayes e

$$\begin{aligned} \mathbb{E}[(y(x) - \hat{y}(x))^2] &= (\mathbb{E}[\hat{y}(x)] - y(x))^2 + \mathbb{E}[(\hat{y}(x) - \mathbb{E}[\hat{y}(x)])^2] \\ &= Bias(\hat{y}(x))^2 + Var(\hat{y}(x)) \end{aligned} \tag{1.7}$$

Si ottiene cosí il compromesso distorsione-varianza (*bias-variance tradeoff*):

$$\mathbb{E}[L(y, \hat{y})_{test}] = \sigma^2 + Bias(\hat{y}(x))^2 + Var(\hat{y}(x)) \tag{1.8}$$

Se la distorsione ha valori alti e la varianza bassi avremo un fenomeno di adattamento insufficiente, mentre se la distorsione ha valori bassi e la varianza alti avremo un adatteamento eccessivo.[5]
Un modo per equilibrare questo compromesso è usare la convalida incrociata (*Cross-Validation*), che consiste nel ripetere l'addestramento e il test dell'algoritmo ogni volta su sottoinsiemi scelti in maniera casuale.
Alcune varianti di convalida incrociata verranno analizzate in seguito.

1.2 Apprendimento supervisionato

Gli algoritmi di apprendimento supervisionato vengono utilizzati per risolvere problemi di classificazione e di regressione, che analizzeremo rispettivamente nel cap. 3 e nel paragrafo 1.2.1.
Si parla di apprendimento supervisionato quando il dataset che si utilizza contiene delle variabili, una delle quali è un'etichetta.
Dato un vettore di input $x = (x_1, \cdots, x_n)$ ogni x_i è un vettore d-dimensionale di numeri rappresentanti una caratteristica, da questi dati si costruisce l'insieme di addestramento di cardinalità N: $D = \{(x_i, y_i)\}_{i=1}^{N}$, dove $y = (y_1, \cdots, y_m)$ è l'output dei risultati desiderati e y_i è l'etichetta. Lo scopo è di apprendere una regola generale che colleghi i dati in ingresso con quelli in uscita, in modo che l'algoritmo apprenda a classificare un esempio completamente nuovo, non contenente l'etichetta.
Se y_i è di tipo testuale si parla di classificazione, quando invece è di tipo numerico si parla di regressione. Se indichiamo con C il numero delle classi a cui può appartenere l'output: $y \in \{1, ..., C\}$, se $C = 2$ la classificazione sarà binaria (in questo caso spesso $y \in \{0, 1\}$); se $C > 2$ sará multiclasse.

1.2.1 Regressione

La Regressione prevede il valore futuro di un dato, avendo noto il suo valore attuale. Un esempio è la previsione della quotazione delle valute o delle azioni di una società. Nel marketing viene utilizzato per prevedere il tasso di risposta di una campagna sulla base di un dato profilo di clienti; nell'ambito commerciale per stimare come varia il fatturato dell'azienda al mutare della strategia. Questo avviene costruendo una funzione che meglio si adatta ai punti che descrivono la distribuzione delle Y, in funzione delle X, precedentemente osservate. Ovvero: osservati n esempi per cui $f(x_i) = y_i, \forall i = 1, \cdots, n$ si cerca di prevedere il valore di $\hat{y}$, dato un nuovo valore $\hat{x}$, tramite la stima della funzione f.

Regressione lineare

Preso un vettore $x \in \mathbb{R}^n$ in input, l'algoritmo cerca di prevedere l'output: $y \in \mathbb{R}$. Dove $y = f(x)$ con f una funzione lineare:

$$y = \alpha + \beta x \qquad (retta\ di\ regressione)$$

Sia $\hat{y}$ il valore di output che l'algoritmo prevede. Definiamo l'output come:

$$\hat{y} = \alpha + \beta x + \epsilon$$

dove ϵ è la componente di errore.
Ora per identificare la retta che meglio si adatta ai punti $P_i(x_i, y_i)$ del sistema cartesiano bidimensionale, bisogna stimare i valori dei parametri α e β, tramite i dati ossrvati su un esempio.
Usiamo il metodo dei minimi quadrati (*least squares*) che minimizza ϵ.

$$\begin{aligned}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 &= \sum_{i=1}^{n}(\hat{y}_i - (\alpha + \beta x_i))^2 \\ &= \sum_{i=1}^{n}(\hat{y}_i - \alpha - \beta x_i)^2 = \min\end{aligned}$$

Per trovare i valori di α e β risolviamo il sistema:

$$\begin{cases}\frac{\partial(\sum_{i=1}^{n}(\hat{y}_i-\alpha-\beta x_i)^2)}{\partial\,\alpha} = 0 \\ \frac{\partial(\sum_{i=1}^{n}(\hat{y}_i-\alpha-\beta x_i)^2)}{\partial\,\beta} = 0\end{cases} \implies \begin{cases}-2\sum_{i=1}^{n}(\hat{y}_i - \alpha - \beta x_i) = 0 \\ -2\sum_{i=1}^{n}(\hat{y}_i - \alpha - \beta x_i)x_i = 0\end{cases}$$

$$\implies \begin{cases}\sum_{i=1}^{n}\hat{y}_i - n\alpha - \beta\sum_{i=1}^{n}x_i = 0 \\ \sum_{i=1}^{n}x_i\hat{y}_i - \alpha\sum_{i=1}^{n}x_i - \beta\sum_{i=1}^{n}x_i^2 = 0\end{cases}$$

Ponendo $\bar{y} = \mathbb{E}[\hat{y}] = \frac{\sum_{i=1}^{n}\hat{y}_i}{n}$, si ottiene:

$$\begin{cases}\bar{y} - \alpha - \beta\bar{x} = 0 & (1.9) \\ \sum_{i=1}^{n}x_i\hat{y}_i - \alpha n\bar{x} - \beta\sum_{i=1}^{n}x_i^2 = 0 & (1.10)\end{cases}$$

Dall'eq 1.9 si ricava $\alpha = \bar{y} - \beta\bar{x}$, che sostituata nella 1.10 si ottiene:

$$\begin{aligned}\sum_{i=1}^{n}x_i\hat{y}_i - (\bar{y} - \beta\bar{x})n\bar{x} - \beta\sum_{i=1}^{n}x_i^2 &= \sum_{i=1}^{n}x_i\hat{y}_i - n\bar{x}\bar{y} + n\beta\bar{x}^2 - \beta\sum_{i=1}^{n}x_i^2 = \\ &= \frac{\sum_{i=1}^{n}x_i\hat{y}_i}{n} - \bar{x}\bar{y} + \beta(\bar{x}^2 - \frac{\sum_{i=1}^{n}x_i^2}{n}) = 0\end{aligned}$$

da cui si ottiene:

$$\beta = \frac{\frac{\sum_{i=1}^{n}x_i\hat{y}}{n} - \bar{x}\bar{y}}{\frac{\sum_{i=1}^{n}x_i^2}{n} - \bar{x}^2} \quad (1.11)$$

ricordando le definizioni di Covarianza e Varianza:

$$\begin{aligned}Cov(x, y) &= \mathbb{E}[x, y] - \mathbb{E}[x] & (1.12) \\ Var(x) &= \mathbb{E}[x^2] - \mathbb{E}[x]^2 & (1.13)\end{aligned}$$

sostituendole nell'eq 1.11,si ricava β, quindi i parametri che si cercavano per α e β sono:

$$\alpha = \mathbb{E}[y] - \beta\mathbb{E}[x] \tag{1.14}$$

$$\beta = \frac{Cov(x,y)}{Var(x)} \tag{1.15}$$

oppure $\beta = \frac{Cov(x,y)}{\sigma^2)}$, dove $\sigma = \sqrt{Var(x)}$ è la deviazione standard.

1.3 Apprendimento non supervisionato

Gli algoritmi di apprendimento non supervisionato vengono utilizzati per risolvere problemi di raggruppamento. All'algoritmo viene passato solo l'input: $D = \{x_i\}_{i=1}^{N}$ e cerca una relazione tra i dati per capire se e come essi siano collegati tra di loro. Non contenendo alcuna informazione preimpostata, l'algoritmo è chiamato a creare una "nuova conoscenza" (*knowledge discovery*). A differenza del caso supervisionato, questo apprendimento non ha una classificazione o un risultato finale con il quale determinare se il risultato è attendibile, ma generalizza le caratteristiche dei dati e in base ad esse attribuisce ad un input un output: serve generalmente ad estrarre informazioni non ancora note, "creando" esso stesso delle classi in cui dividere i dati. Questa tecnica si chiama *clustering*.

1.4 Apprendimento per rinforzo

Gli algoritmi di apprendimento per rinforzo vengono utilizzati per risolvere problemi di regressione. Lo scopo di questo algoritmo è di realizzare un sistema in grado di apprendere ed adattarsi ai cambiamenti dell'ambiente in cui si trovano, attraverso la distribuzione di una "ricompensa" detta rinforzo, data dalla valutazione delle prestazioni. Questi algoritmi sono costruiti sull'idea che i risultati corretti dovrebbero essere ricordati, per mezzo di un segnale di rinforzo, in modo che diventino piú probabili e quindi piú facilmente riottenuti nelle volte future; viceversa se il rusultato è errato , il segnale sará una penalitá, ovvero si avrá una probabilitá piú bassa legata a quel determinato output.[6]

Capitolo 2

Reti neurali artificiali

Le reti neurali sono i modelli di deep learning per eccellenza. Sono un sistema di elaborazione di informazioni ispirato al funzionamento del sistema nervoso umano.
La rete è strutturata come un grafo orientato. I nodi sono raggruppati in strati (*layers*): il primo strato contiene i nodi di input $x_1, \cdots, x_n$, connessi con lo strato successivo, dove ad ogni arco è associato un peso w_i. L'ultimo strato contiene i nodi di output. Gli strati tra il primo e l'ultimo strato sono chiamati strati nasconti (*hidden layers*). La lunghezza complessiva del percorso determina la profonditá del modello, da cui deriva il nome dell'apprendimento: "deep learning".
In base all'architettura scelta, esistono vari modelli di reti neurali; la scelta dell'architettura della rete è molto importante, poiché in base al numero di nodi usati per ogni strato ed alla profonditá, il costo computazionale cresce o diminuisce: per esempio la scelta di un'architettura poco profonda e con una elevata quantitá di nodi per strato, causa un costo computazionale elevato ed un massiccio utilizzo della memoria.
Lo scopo di questa tesi è di trovare l'architettura ottimale per un algoritmo di classificazione di immagini considerando anche la geolocalizzazione dell'oggetto da riconoscere (e del suo visualizzatore).
Ritornando alle reti neurali, in generale, abbiamo detto che si ispirano al nostro sistema nervoso; vediamo brevemente in che modo.
Ogni neurone, nel nostro cervello, riceve un'intera serie di segnali da altri neuroni, li somma all'interno del suo corpo cellulare e, sulla base della somma, aggiusta la frequenza delle scariche da inviare alla cellula successiva.
I neuroni ricevono sia segnali eccitatori, ovvero che tendono ad aumentare la loro frequenza di scarica, che segnali inibitori, che tendono invece a diminuirla, ma nonostante ricevano due tipi di segnali, ne emettono poi di un solo tipo. Analogamente: ogni neurone aritificiale, rappresentato da un no-

do, diventa attivo se la quantitá totale di segnale che riceve supera la soglia di attivazione, definita dalla cosiddetta funzione di attivazione. Se un nodo diventa attivo, emette un segnale che viene trasmesso lungo i canali di trasmissione fino all'altra unità a cui è collegato.

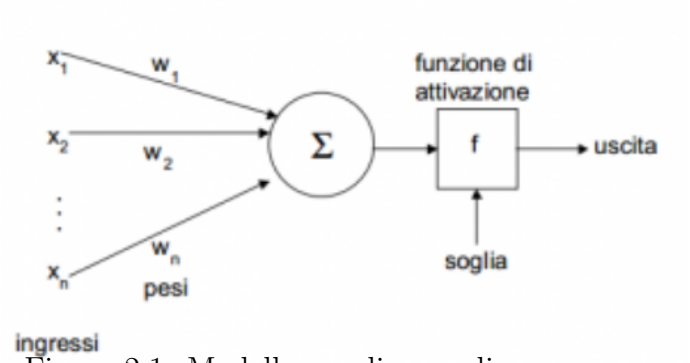


Figura 2.1: Modello non lineare di un neurone artificiale.

La figura 2.1, indipendentemente dal modello di rete utilizzato, mostra l'elaborazione eseguita da un neurone artificiale.
Siano $x_1, \cdots, x_n$ i dati in input, rappresentati dagli n nodi del primo strato, nel k-esimo neurone l'informazione viene elaborata come:

$$y_k = f(b_k + \sum_{i=1}^{n} w_{ki} \cdot x_i)$$

dove:

- y_k è l'output generato dal neurone k;
- b_k è il valore soglia del neurone k;
- w_{ki} è il peso associato all'arco che collega il nodo i-esimo al neurone k;
- $f(\cdot)$ è la funzione di attivazione.

Il motivo principale per cui vengono scelte le reti neurali è la possibilitá di parallelizzare i calcoli.

2.1 Funzioni di attivazione

La funzione di attivazione è una funzione usata per normalizzare, quindi limitare, l'ampiezza dell' output, per non consumare eccessiva memoria e di velocizzare il processo di calcolo.
Generalmente le reti neurali sono utilizzate per implementare funzioni complesse e le funzioni di attivazione non lineari consentono loro di approssimare funzioni arbitrariamente complesse. Le funzioni maggiormente utilizzate a tale scopo sono 3:

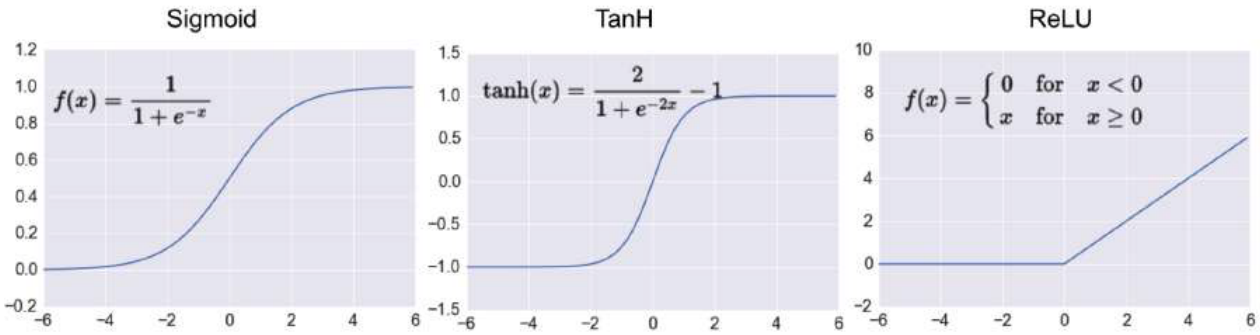


Figura 2.2: Grafici delle funzioni di attivazione più usate.

Sigmoid

La funzione Sigmoid viene usata sopratutto nei modelli in cui si deve prevedere la probabilità come output, poiché il suo codominio è l'intervallo: $(0, 1)$ ed è definita come:

$$S : \mathbb{R} \longrightarrow (0, 1)$$

$$S(x) = \frac{1}{1 + \exp^{-x}}$$

La Sigmoid è una funzione monotona e differenziabile, infatti la sua derivata:

$$\frac{dS}{dx} = \frac{\exp^{-x}}{(1 + \exp^{-x})^2}$$

è continua, positiva e derivabile in ogni punto del dominio.

tanh

La funzione tangente iperbolica è definita come:

$$\tanh : \mathbb{R} \longrightarrow (-1, 1)$$
$$\tanh(x) = \frac{2}{1 + \exp^{-2x}} - 1$$

è simile alla Sigmoide, infatti sono legate dalla relazione:

$$\tanh(x) = 2 \cdot S(2x) - 1$$

ma ha un intervallo di output più ampio, che le consente di produrre anche output di segno negativo. Anche questa funzione è monotona e differenziabile.
Purtroppo essendo funzioni limitate, per valori alti tendono ad un valore specifico:

$$\lim_{x \to +\infty} \tanh(x) = 1$$

. Questo comportamento può portare ad una perdita di informazioni: se x corrisoponde ad un valore elevato e subisce una grande variazione, per la funzione invece avrá subito una variazione minima.
Lo stesso problema si presenta con le derivate, provocando la "scomparsa del gradiente"(*vanishing gradient*), fondamentale per approssimare al meglio la funzione; ma questo argomento verrá affrontato in 2.2.

ReLu

La funzione ReLu (Rectified Linear Unit) è la funzione piú utilizzata nel deep learning e, in particolare, nelle reti convoluzionali. È definita:

$$Relu : \mathbb{R} \longrightarrow [0, \infty)$$
$$Relu(x) = \begin{cases} x & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases}$$

Sia la funzione, che la sua derivata sono monotone. A differenza delle funzioni precedenti, la sua derivata è molto semplice da calcolare:

$$Relu'(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{se } x < 0 \end{cases}$$

e non rischia di incorrere nella sparizione del gradiente. Si potrebbero avere problemi nell'origine, per la presenza di un punto angoloso, poiché la derivata è indefinita ma, per convenzione, viene definita uguale a zero.
Risulta immediato il motivo per cui sono molto utilizzate nelle reti formate da tanti strati: mappando i valori negativi in zero, permette di tenere solo i valori positivi, riducendo di molto il numero di neuroni attivati.

2.2 Addestramento di una rete

Come visto in 1.1, per addestrare un algoritmo si ha bisogno di una funzione di perdita e di un metodo per minimizzare l'errore di valutazione.
L'addestramento di una rete neurale si basa sugli stessi principi:
si definisce una mappa:

$$y = f(x, \theta)$$

e si cerca il valore del parametro θ che più accuratamente approssima la funzione. Bisogna quindi trovare dei parametri che minimizzano la funzione di perdita:

$$L[y, f^*(x, \theta)]$$

ovvero, trovare $\hat{\theta}$ tali che:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f^*(x_i, \theta)] \right\}$$

Questo processo è necessario per approssimare una determinata funzione f^*, che descrive il comportamento della rete.
Durante l'addestramento, la rete viene provata più volte, ogni volta con un input diverso, fino a che l'errore di addestramento è molto piccolo. In questa fase, ad ogni iterazione viene passato in input un insieme di addestramento, viene fissato un valore θ_0 iniziale, e tramite la funzione di perdita, si calcola l'errore di addestramento. In questo modo la rete ha il valore di quanto ciascun neurone di output sia lontano dal proprio valore atteso, e in che direzione (positiva o negativa).
Per esempio, nel caso di riconoscimento dei numeri scritti a mano (MNIST): se il risultato atteso è 6, nell'output ci aspettiamo che nella posizione corrispondente al sesto neurone ci sia il valore 1 e 0 in tutti gli altri. Se al neurone 6 è associato il valore 0.7, allora la correzione da fare è di 0.3. Quindi la rete ripete l'addestramento aggiornando il peso corrispondente al neurone valutato con θ_1 e cosí via, fino a che il valore corrispondente al neurone 6 è molto vicino a 1. Viceversa, se il neurone 4 invece di 0 presenta 0.8, allora la correzione sará -0.8 e si aggiorneranno i pesi relativi a questo neurone in modo da abbassarne drasticamente l'output.
L'algoritmo appena descritto a parole prende il nome di *back propagation*.

Algoritmo di back propagation

Si consideri una rete neuronale composta da L strati, dove l'output è composto da un valore solamente. L'algoritmo di back propagation si divide in due fasi:

- propagazione in avanti
- propagazione all'indietro

Durante la propagazione in avanti, si calcolano tutti i valori dei nodi presenti nella rete a_i^k:

$$a_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ji}^k \cdot o_j^{k-1} \tag{2.1}$$

dove:

- w_{ij}^k è il peso associato all'arco che collega il nodo i del k-esimo strato con il nodo j;
- b_i^k è il valore soglia del nodo i nel k-esimo strato;
- o_i^k è l'output del nodo i nel k-esimo strato;
- r_k è il numero di nodi presenti nel k-esimo strato.

Calcolato l'ultimo valore a^L, corrispondente all'output della rete, inizia la seconda fase. Per calcolare il gradiente della funzione di perdita $L[y, f(x, \theta)]$, si applica la regola della catena per il calcolo e poiché $\theta = (W^1, W^2, \cdots, W^{L-1})$, abbiamo:

$$\frac{\partial L}{\partial w_{ij}^k} = \frac{\partial L}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}. \tag{2.2}$$

Il punto di forza di questo algoritmo consiste nell'introdurre una quantitá di costo (anche chiamata errore):

$$\delta_j^k \equiv \frac{\partial L}{\partial a_j^k} \tag{2.3}$$

attraverso la quale si calcolano le derivate parziali in modo iterativo, ripercorrendo la rete all'indietro.
Dall'eq 2.1 si ricava:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} \right) = o_i^{k-1} \tag{2.4}$$

sostituendola nell'eq 2.2 si ottiene:

$$\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} \tag{2.5}$$

Partendo dallo strato dell'output, considerando come funzione di perdita:

$$L = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}(g_o(a^L) - y)^2,$$

si ha che:

$$\delta^L = (g_o(a^L) - y)g_o'(a^L) = (\hat{y} - y)g_o'(a^L). \tag{2.6}$$

dove g_o è la funzione di perdita per l'output.
Quindi la derivata parziale della funzione di perdita è:

$$\frac{\partial L}{\partial w_{iL-1}^L} = \delta^L o_i^{L-1} \tag{2.7}$$

Per gli strati nascosti invece abbiamo:

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \frac{\partial L}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{a_j^k} = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{a_j^k}, \tag{2.8}$$

Sostituendo in 2.8 la 2.1 e denotando con g la funzione di perdita dello strato nascosto, si ottiene la formula di propagazione all'indietro:

$$\delta_j^k = g'(a_j^k) \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l k + 1 \tag{2.9}$$

e la derivata parziale della funzione di perdita, rispetto ai pesi degli strati nascosti w_{ij}^{k+1} , $1 \leq k < L$, si ottiene con:

$$\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'(a_j^k) o_i^{k-1} \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l k + 1 \tag{2.10}$$

Ora puó avvenire l'aggiornamento dei pesi tramite il metodo SGD (Stochastic Gradient Descent), ovvero in formule:

$$w_{ij}^{k+1} = w_{ij}^k + \eta \frac{\partial L}{\partial w_{ij}} \tag{2.11}$$

dove $\eta \in (0, 1]$ rappresenta il fattore di apprendimento (*learning rate*).
La scelta del fattore di apprendimento influenza molto il comportamento dell'algoritmo in quanto, se si scelgono valori troppo piccoli la convergenza sarà

lenta, mentre se si scelgono valori troppo grandi si rischia di avere una rete instabile con comportamento oscillatorio.
Un metodo semplice per incrementare il fattore di apprendimento, senza il rischio di rendere la rete instabile, è quello di modificare la regola di aggiornamento inserendo nell'equazione 2.11 un ulteriore parametro α (detto momento), ottenendo:

$$w_{ij}^{k+1} = \alpha w_{ij}^{k} + \eta \frac{\partial C}{\partial w_{ij}}. \tag{2.12}$$

Se si espande ricorsivamente la formula si ottiene:

$$w_{ij}^{k+1} = \eta \sum_{l=1} k + 1 \alpha_{k+1-l} \frac{\partial L}{\partial w_{ij}^{l}} \tag{2.13}$$

Inserendo il momento si ha il vantaggio che se la derivata parziale tende a mantenere lo stesso segno su iterazioni consecutive, grazie alla sommatoria, l'aggiornamento produce valori piú ampi e quindi tende ad accelerare nelle discese del gradiente. Se invece la derivata ha segni opposti ad iterazioni consecutive, la sommatoria tende a diminuire l'ampiezza dell'aggiornamento, in questo modo non si corre il rischio di avere dei loop infiniti nel caso di minimi locali della funzione d'errore.
A tal proposito generalmente i criteri di arresto dell'algoritmo possono essere:

- $\|\nabla L\| < \epsilon$
- $L[y, f(x, \theta)] = 0$;
- $L_i[y, f(x, \theta_i)] - L_j[y, f(x, \theta_j)] << \epsilon'$

dove L_i e L_j sono due epoche consecutive. Un'epoca corrisponde alle due fasi di propagazione necessarie per un aggiornamento dei pesi.

Il problema del Vanish Gradient

Come accennato in precedenza, l'uso del metodo del gradiente va applicato con attenzione e parametri adatti, altrimenti si incorre nel fenomeno detto "sparizione del gradiente", ovvero del *vanish gradient*.
Si ipotizzi di avere una rete profonda semplice, ovvero con 3 livelli nascosti e con un solo neurone in ogni strato:

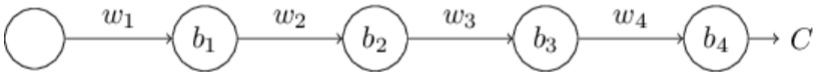


Figura 2.3: Rete neurale semplice costituita da 3 strati con: w_j pesi, b_j i valori soglia e C funzione di costo.

Applicando l'eq 2.1 per il calcolo dell'output di un nodo, alla rete rappresentata in figura 2.3, si ottiene:

$$a_j = \sigma(w_j a_{j-1} + b_j) \tag{2.14}$$

dove σ è la funzione di attivazione Sigmoid.
Si pone $z_j = w_j a_{j-1} + b_j$ e si ottengono le seguenti relazioni:

$$\begin{aligned} a_4 &= \sigma(z_4) \\ z_4 &= w_4\sigma(z_3) + b_4 \\ z_3 &= w_3\sigma(z_2) + b_3 \\ z_2 &= w_2\sigma(z_1) + b_2 \\ z_1 &= w_1\sigma(z_0) + b_1 \end{aligned}$$

Si è indicato con C una funziona di costo, per sottolineare che il costo è una funzione dell'output della rete (in questo caso a_4).
Se C ha un valore basso l'output sará vicino a quello desiderato, viceversa per alti valori di C l'output calcolato si allontanerá dal risultato che ci si auspicava.
Il gradiente associato al primo strato viene calcolato applicando la regola della catena:

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \cdot \frac{\partial a_4}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial b_1} \tag{2.15}$$

Calcoliamo singolarmente ogni elemento dell'equazione:

$$\frac{\partial a_1}{\partial b_1} = \frac{\partial \sigma(z_1)}{\partial b_1} = \frac{\partial \sigma(w_1\sigma(z_0) + b_1)}{\partial b_1} = \sigma'(z_1) \tag{2.16}$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial \sigma(z_2)}{\partial a_1} = \frac{\partial \sigma(w_2 a_1 + b_2)}{\partial a_1} = w_2\sigma'(z_2) \tag{2.17}$$

$$\frac{\partial a_3}{\partial a_2} = \frac{\partial \sigma(z_3)}{\partial a_2} = \frac{\partial \sigma(w_3 a_2 + b_3)}{\partial a_2} = w_3\sigma'(z_3) \tag{2.18}$$

$$\frac{\partial a_4}{\partial a_3} = \frac{\partial \sigma(z_4)}{\partial a_3} = \frac{\partial \sigma(w_4 a_3 + b_4)}{\partial a_3} = w_4 \sigma'(z_4) \tag{2.19}$$

Quindi l'eq 2.15 diventa:

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} w_4 \sigma'(z_4) w_3 \sigma'(z_3) w_2 \sigma'(z_2) \sigma'(z_1). \tag{2.20}$$

Si noti che l'equazione 2.20 è il prodotto di termini della forma: $w_j \sigma'(z_j)$, fatta eccezione per il primo termine.
Tipicamente, i valori iniziali dei pesi vengono ricavati dai valori che puó assumere una distribuzione normale avente media 0 e deviazione standard 1, allora:

$$|w_j| < 1, \qquad \forall j.$$

La derivata raggiunge il massimo in 0: $\sigma'(0) = \frac{1}{4}$, quindi ogni termine dell'equazione è tale che

$$|w_j \sigma'(z_j)| < \frac{1}{4}, \qquad \forall j,$$

si ottiene la stima del tipo:

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \underbrace{w_4 \sigma'(z_4)}_{<\frac{1}{4}} \underbrace{w_3 \sigma'(z_3)}_{<\frac{1}{4}} \underbrace{w_2 \sigma'(z_2)}_{<\frac{1}{4}} \underbrace{\sigma'(z_1)}_{<1}$$

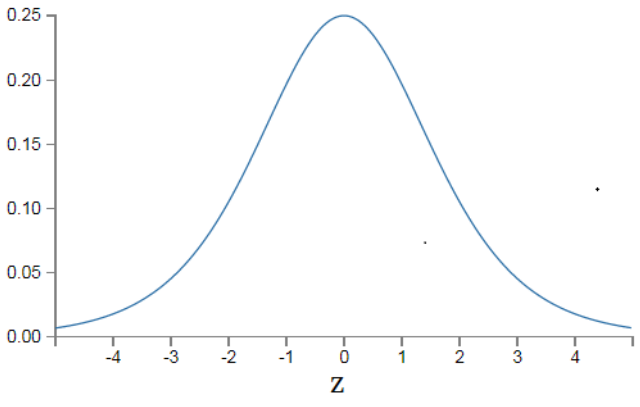
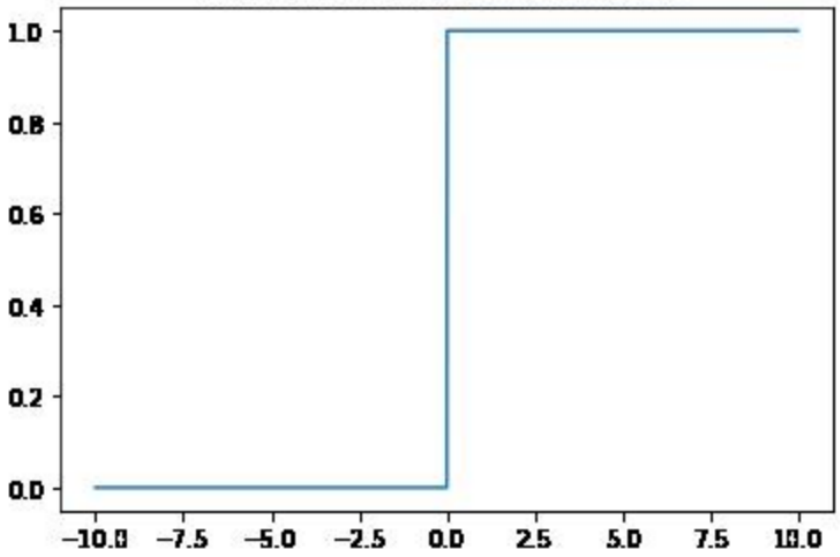


Figura 2.4: Grafico della funzione $\sigma'(z)$

Questo comportamento implica che, con l'aumentare della profonditá della rete, il valore del gradiente tende a diminuire di un fattore di $\frac{1}{4}$ per ogni strato, fino a tendere a 0 prematuramente; ovvero a "scomparire" da un certo strato in poi.
L'utilizzo della funzione di attivazione ReLu risolve il problema, grazie alla sua caratteristica di avere il gradiente sempre pari a 1. Questo comporta che , non solo il gradiente non subisce una diminuzione progressiva in ogni strato, ma mantiene il suo valore inalterato.
Si ricorda che $ReLu(z) = z_+ = max(0, z)$, il che comporta che alcuni neuroni non vengono attivati e quindi non influenzeranno in alcun modo l'algoritmo scelto per individuare i pesi ottimali per la rete.
Ne consegue che $ReLu'(z) = 1$ per ogni nodo rimasto attivo.
Nel caso della rete prima analizzata con la funzione Sigmoid, si è dovuta fare una stima del gradiente per ogni neurone, con l'inserimento della ReLu non è necessario e il calcolo del gradiente è:

$$\frac{\partial C}{\partial b_j} = 1, \qquad \forall j. \tag{2.21}$$

Figura 2.5: Grafico di $ReLu'(z)$



2.3 Reti Deep Feed-forward

Le reti Feed-forward sono le reti neurali profonde con la struttura più semplice, composte da almeno uno strato nascosto. La loro struttura è rappresentata da un grafo aciclico diretto in un'unica direzione, dove ogni nodo di uno strato è connesso con tutti i nodi dello strato successivo e nessun nodo è connesso con un nodo appartanente allo stesso strato.

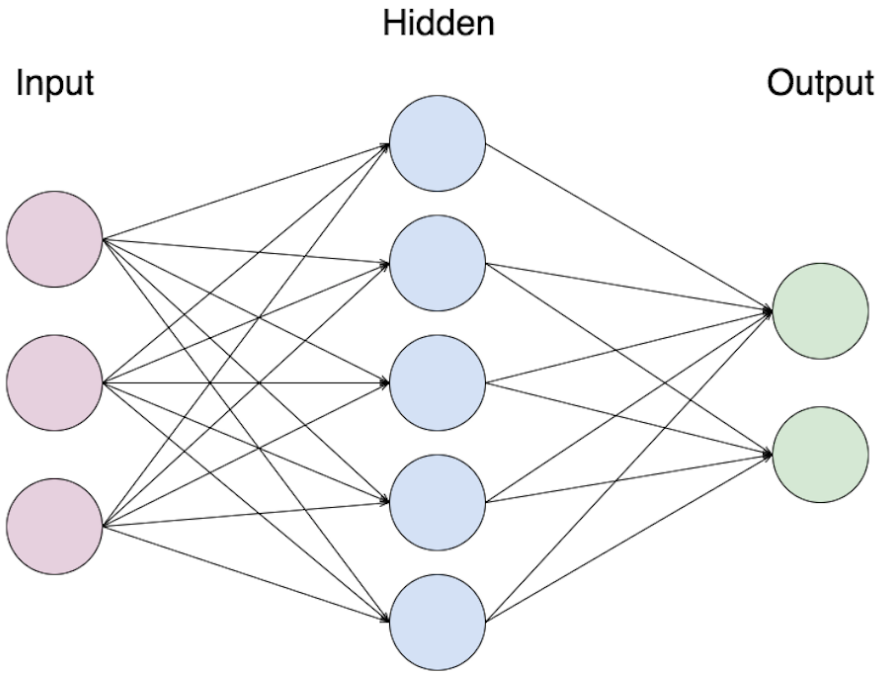


Figura 2.6: modello di rete deep feedforward con uno strato

Questo tipo di reti sono di solito rappresentate dalla composizione di più funzioni, dal momento che l'output di un nodo non puó essere passato in input ad una funzione già eseguita. Nel caso di una rete con L strati avremo che la relazione che lega il vettore di input x con quello di output y risulta essere:

$$y = f(x, \theta) = g_o(\theta_{L-1} g_{L-1}(\cdots \theta_2 g_2(\theta_1 x))) \tag{2.22}$$

Lo scopo della rete è sempre di approssimare $f(x)$ ad una funzione f^* e questo è possibile grazie all'addestramento.

2.4 ImageNet

In Internet oramai sono presenti milioni di milioni di immagini e di video, usati per i modelli e gli algoritmi piú sofisticati e robusti, per aiutare gli utenti ad indicizzare, recuperare, organizzare e interagire con questi dati. Un esempio banale è la ricerca di immagini su Google.
Se si volesse ricercare una razza canina in particolare, come fa l'algoritmo ad individuare un'immagine di un Labrador piuttosto che di un Beagle?
Lo fa tramite categorie e sottocategorie. Ma esattamente come tali dati possono essere utilizzati e organizzati è un problema che non è ancora stato risolto.
Di solito i motori di ricerca possono trovare una determinata immagine solo se il testo inserito corrisponde al testo con cui è stato etichettato. Ma le etichette possono essere inaffidabili, inutili (nel caso di dialetti e nomignoli) o semplicemente inesistenti.
Si vuole quindi cercare un metodo per far imparare alle macchine come riconoscere oggetti simili che non sono stati ancora etichettati, rendendo possibile un notevole aumento dell'accuratezza del riconoscimento.
È stato creato *ImageNet*: un grande database visivo contenente oltre 14 milioni di immagini etichettate, progettato per l'utilizzo nella ricerca di software per il riconoscimento di oggetti visivi.
ImageNet si basa sulla struttura gerarchica fornita da un altro database: *WordNet*. WordNet è un database semantico-lessicale per la lingua inglese elaborato dal linguista George Armitage Miller presso l'Universitá di Princeton, che si propone di organizzare, definire e descrivere i concetti espressi dai vocaboli. L'organizzazione del lessico si avvale di raggruppamenti di termini con significato affine, chiamati *"synset"* (dalla contrazione di synonym set) e del collegamento dei loro significati attraverso diversi tipi di relazioni chiaramente definite. All'interno dei synset le differenze di significato sono numerate e definite.[7]
Un'immagine digitale è formata da diversi quadratini disposti in modo regolare su una griglia di punti equidistanti. Questi quadratini sono detti *pixel (picture elements)*.
I pixel presenti in um'immagine ne determinano la dimensione e la risoluzione; ad esempio un'immagine di dimensione 320×240 pixel indica che sono presenti 240 pixel orizzontali e 320 verticali. Piú i pixel sono numerosi, e quindi piú piccoli e fitti, piú la risoluzione è alta.
In ogni pixel risiede un'informazione espressa in bit riguardante (generalmente) il colore: un pixel da 1 bit puó avere solo 2^1colori (generalmente il bianco e il nero), mentre uno da 1 byte (8bit) puó rappresentare $2^8 = 256$ colori. Il numero di colori possibili è detto anche profonditá. Le rappresentazioni

delle immagini a colori variano a seconda dei campi di colore che si usano, solitamente, ogni campo viene rappresentato da 1 byte e rappresenta il livello di intensitá dei colori fondamentali.
Il modello piú utilizzato è quello RGB, dove per ogni pixel vengono utilizzati 3 byte:

- 1 byte per la componente rossa (R)
- 1 byte per la componente verde (G)
- 1 byte per la componente blu (B)

ma ne esistono anche altri come, ad esempio, CMYK che considera come colori fondamentali: ciano, magenta, giallo e nero.
Tipicamente nel dataset di ImageNet la dimensione media delle immagini è di circa 400×350 pixel.

2.5 Reti Neurali Convoluzionali - CNNs

Le reti neurali convoluzionali (*Convolutional Neural Networks*) sono un tipo particolare di reti neurali a strati che usano, in almeno uno dei loro strati, la convoluzione al posto della classica moltiplicazione tra matrici.

Definizione 6 *Siano $f, g \in L^1(\mathbb{R})$, si definisce convoluzione tra f e g come:*

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = y(t) \tag{2.23}$$

Questa operazione gode di alcune proprietá:

1. **proprietá commutativa**

   $$(f*g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = -\int_{\infty}^{-\infty} f(t-z)g(z)dz = \int_{-\infty}^{\infty} g(z)f(t-z)dz = (g*f)$$

   dove $z = t - \tau \Rightarrow \tau = t - z \Rightarrow d\tau = -dz$.

2. **Linearitá**

   $$(f_1 + f_2) * g = (f_1 * g) + (f_2 * g), \qquad (\lambda f) * g = \lambda(f * g).$$

3. **Integrale** Se $f, g \in L^1(\mathbb{R})$ sono assolutamente integrabili allora

   $$\int_{\mathbb{R}} (f * g)(t)dt = \int_{\mathbb{R}} f(t)dt \cdot \int_{\mathbb{R}} g(t)dt.$$

4. **Cambiamenti di scala**

$$sey = f * g \Rightarrow f(\lambda t) * g(\lambda t) = \frac{1}{|\lambda|} y((\lambda t)$$

5. **Derivate** Se f è regolare a tratti e continua, allora

$$\frac{d}{dt}(f * g)(t) = \frac{d}{dt} f(t) * g(t) = f(t) * \frac{d}{dt} g(t)$$

6. **Supporti** Se supp $f \subset [a, b]$ e supp $g \subset [c, d]$ ⇒supp $(f * g) \subset [a + c, b + d]$.

7. **Segnali causali** Se f,g sono causali ⇒ $f * g$ è causale.

Tramite l'eq2.23, se si conosce la funzione di risposta g(t) e il segnale di entrata f(t), si puó esprimere il segnale di uscita y(t), che nelle reti convoluzionali rappresenta il filtro di convoluzione (*feature map*) e la funzione di risposta è detta *"kernel"*. Si ricorda che per *segnale* si intende una grandezza fisica qualsiasi a cui è associata un'informazione.
Nelle reti neurali i segnali non sono descritti da funzioni continue ma da funzioni discrete, in quanto l'input passato ad un neurone è un valore discreto ben preciso.
Si necessita qundi delle seguenti definizioni:

Definizione 7 *I segnali discreti sono definiti come funzioni di variabili indipendenti che possono assumere solo un insieme finito di valori discreti.*
Un segnale discreto nel tempo x, consiste in una sequenza di numeri indicata con

$$x_n \qquad oppure \qquad x(n), \quad n \in \mathbb{Z}$$

Definizione 8 *Siano $f(n)$ e $g(n)$ due sequenze, si definisce la loro convoluzione discreta come:*

$$f(n) * g(n) = \sum_{k=-\infty}^{\infty} x(k) g(n - k) = y(k) \tag{2.24}$$

Valgono le stesse proprietá prima citate per la convoluzione nel caso continuo.
Nel nostro problema di analisi di immagini, data un'immagine di dimensione M × N pixel l'eq2.24 diventa:

$$(f * g)(x, y) = \sum_{i=0}^{M} \sum_{j=0}^{N} f(x, y) g(x - i, y - j) \tag{2.25}$$

La caratteristica principale per cui le reti convoluzionali sono maggiormente indicate per l'elaborazione di immagini è la loro struttura, basata sull'elaborazione di dati nella forma di array multipli.
Come descritto nel paragrafo 2.4 è facilmente intuibile come ogni dimensione caratterizzante un'immagine puó essere vista come un array multidimensionale, ovvero un tensore.
I neuroni di ciascun livello sono organizzate in griglie o volumi 3D.
L'altezza e la larghezza rappresentano i pixel, mentre la profondità le caratteristiche dell'oggetto, chiamate *feature map*.

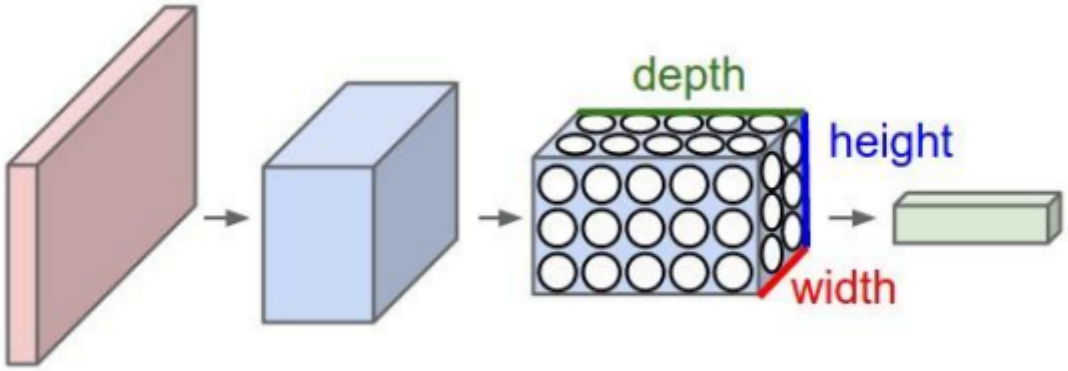


Figura 2.7: Esempio di rappresentazione di un'immagine in formato RGB in una CNN.

2.5.1 Struttura di una CNNs

Alla base di questa tipologia di reti neurali ci sono 3 concetti:

1. connettività locale (*sparse interactions*)
2. condivisione dei parametri
3. alternanza di strati di convoluzione e di *pooling*

La **connettività locale** consiste nello sfruttare le correlazioni spaziali presenti all'interno dei dati di input e viene applicata tra neuroni di strati adiacenti. Questo è possibile applicando un kernel più piccolo dell'input che, tramite l'operazione di convoluzione, darà in output un numero minore di neuroni. Ne consegue una forte riduzione del numero di connessioni tra i

nodi della rete.
Nelle prossime sezioni sarà più chiaro cosa questo voglia dire, ma se si considera, ad esempio, un'immagine e su di essa ci si focalizzi su un singolo pixel, in base alle correlazioni con i pixel circostanti si possono ricavare informazioni locali sull'immagine (ad esempio bordi, colori, forme geometriche, ecc...).
Attraverso la **condivisione dei parametri** si utilizza lo stesso parametro per piú di una funzione in un modello, permettendo alla rete di apprendere un insieme di parametri invece che insiemi separati di parametri per ogni posizione, questo comporta che l'algoritmo deve tenere in memoria un numero minore di parametri.

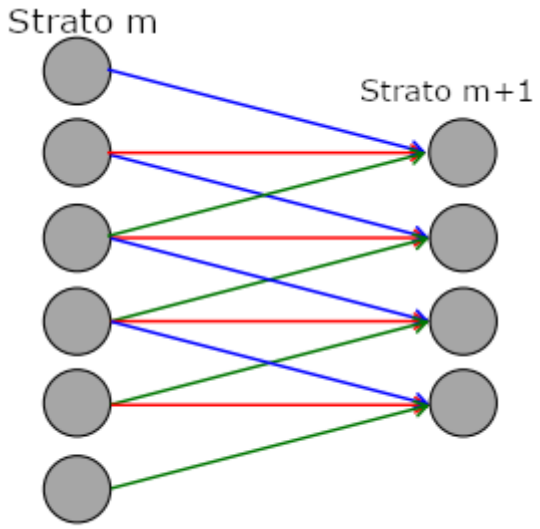


Figura 2.8: Ciascuno dei 4 neuroni a destra è connesso solo a 3 neuroni del livello precedente. I pesi sono condivisi (stesso colore stesso peso).

Nella figura precedente si puó notare che sono presenti in totale 12 connessioni (3× 4) e 3 pesi, mentre in una normale rete neurale equivalente avremmo avuto 24 connessioni (6 × 4) e 24 pesi. In un solo livello sono state utilizzate la metá delle connessioni e $\frac{1}{8}$ dei pesi.

L'architettura di una rete convoluzionale è formata, tipicamente, da tre tipi di strati:

1. convoluzionali;
2. pooling;
3. totalmente connessi.

Uno strato tipico di una rete convoluzionale consiste di tre fasi .Nella prima fase lo strato esegue diverse convoluzioni in parallelo per produrre un set di attivazioni lineari. Nella seconda fase, ogni attivazione lineare viene eseguita attraverso una funzione di attivazione non lineare. Nella terza fase, si esegue una funzione di pooling per modificare ulteriormente l'output dello strato.

Strati convoluzionali

Lo scopo dello strato convoluzionale (*convolutional layer*) è quello di rilevare delle particolari caratteristiche all'interno di un oggetto, grazie ad un'analisi locale. Questo compito è affidato al kernel della convoluzione, detto anche filtro, che si comporta come una finestra bidimensionale composta da pesi che scorre su tutto l'input. Di ogni porzione di input viene calcolata la sua convoluzione con il kernel, producendo cosí un output di dimensioni ridotte rispetto all'input. Questo output viene chiamato *feature map* e racchiude le caratteristiche che il filtro cercava ed è qui che i pesi sono condivisi: i neuroni di una stessa feature map processano porzioni diverse dell' input nello stesso modo.
Il kernel ha due iperparametri, chiamati passo(*stride*) e dimensione(*size*). La size puó essere qualsiasi dimensione di un rettangolo, mentre lo stride è il numero di pixel che si fanno scorrere tra due computazioni della finestra di kernel. Se lo stride ha lunghezza 1, il filtro si sposta di un pixel alla volta, se ha lunghezza 2, i filtri saltano 2 pixel alla volta durante lo scorrimento, producendo un'immagine con dimensione dimezzata.

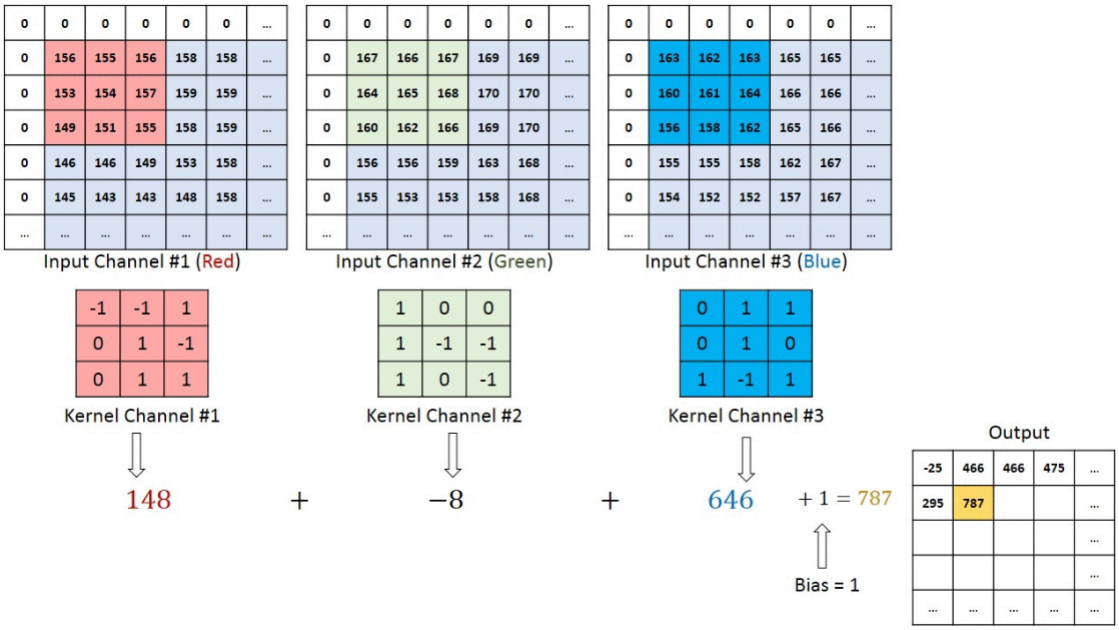


Figura 2.9: Ogni future map ha il suo kernel che scorre per tutta la dimensione dell'input.

Un altro modo per regolare la dimensione delle feature map è tramite l'aggiunta di un bordo al volume di input, inserendo tutti valori nulli in proosimita del bordo. Con il parametro *Padding* si denota lo spessore in pixel del bordo. L'uso di questo parametro ha il vantaggio di mantenere le dimensioni spaziali costanti dopo lo strato convoluzionale, migliorando le prestazioni. Questo perché se gli strati non azzerassero gli input e realizzassero solo convoluzioni valide, allora la dimensione dei volumi si ridurrebbe di una piccola quantità ad ogni convoluzione, in questo modo le informazioni ai bordi si perderebbero troppo rapidamente.
Sia W_{out} la dimensione orizzontale (verticale) delle feature map di output e W_{in} la corrispondente dimensione nell'input. Sia inoltre F la dimensione (orizzontale) del filtro e K il numero di filtri da utilizzare.
Vale la seguente relazione:

$$W_{out} = \frac{W_{in} - F + 2 \cdot Padding}{Stride} + 1 \tag{2.26}$$

e la profondità dell'output sarà pari a K.

Strati di pooling

Lo scopo degli strati di pooling è quello di unire caratteristiche simili (in quanto vicine) in una sola, per ridurre progressivamente la dimensione spaziale della rappresentazione. Di conseguenza si riduce anche la quantitá di parametri, che potebbero comportare un sovradattamento della rete.
Ogni strato appartenente alla profondità dell'input viene suddiviso in piccoli quadrati, chiamati *subsambling*, di dimensione $r \times r$, con $r \in \mathbb{N}$ e per ogni quadrato viene calcolato indipendentemente o il valore massimo (Max) o la media (Avg) dei suoi neuroni.
Max e Avg sono gli operatori di aggregazione maggiormente utilizzati e sono invarianti per piccole tralslazioni. Riassumendo: una funzione di raggruppamento sostituisce l'output della rete in una determinata posizione con una statistica riassuntiva delle uscite vicine.
L'invarianza per traslazione è molto utile in quanto se una caratteristica è interessante in una parte dell'immagine, presumibilmnete lo sarà anche se è posizionata in un altro spazio.
Ad esempio, quando si determina se un'immagine contiene un volto, non è necessario conoscere la posizione degli occhi con "precisione pixel" perfetta, ma soltanto riscontrare la presenza di un occhio sul lato sinistro del viso e uno sul lato destro.
In altri contesti, è piú importante conservare la posizione di una funzione. Ad esempio, se si vuole trovare un angolo definito da due spigoli che si incontrano con un orientamento specifico, bisogna verificare che sia preservata la posizione dei bordi in modo che essi si incontrino.
Passata in input un'immagine di volume $W_{in} \times H_{in} \times D$, lo strato di pooling di dimensione $F \times F$ e stride S produce un volume di dimensioni $W_{out} \times H_{out} \times D$, dove:

$$W_{out} = \frac{W_{in} - F}{S} + 1 \tag{2.27}$$

Alternando ripetutamente queste due tipologie di strati, si arriva ad una configurazione spaziale unita dell'immagine di piccole dimensioni. In quest'ultima prospettiva si passa a strati totalmente connessi, specialmente nell'ultimo strato dove l'output deve necessariamente essere una delle possibili classi.

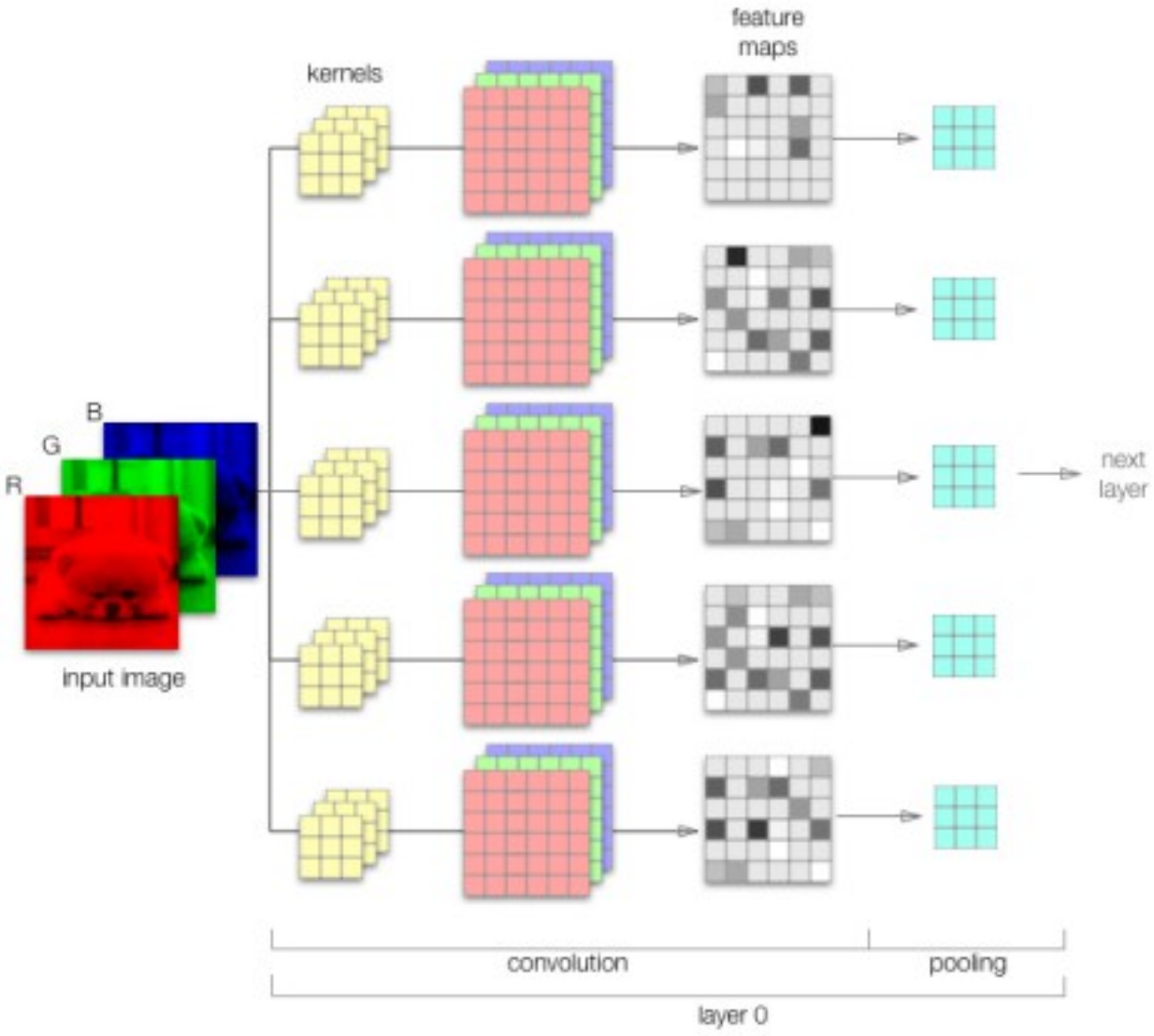


Figura 2.10: Struttura generale di una rete neurale convoluzionale

Capitolo 3

Classificazione

La Classificazione viene usata quando è necessario decidere a quale categoria appartiene un determinato dato.Per esempio, data una foto capire a quale categoria appartiene, in questa tesi vogliamo classificare immagini, piú precisamente: capire a quale tipo di monumento corrispone una determinata immagine.
Questo tipo di algoritmo deve specificare a quale delle k categorie appartiene un input. Crea una funzione $f : \mathbb{R}^n \to \{1, \cdots, k\}$, quando $y = f(x)$, il modello assegna l'input descritto dal vettore x ad una categoria identificata dal codice numerico y.
Esistono altre varianti dell'attivitá di classificazione, ad esempio, dove f genera una distribuzione di probabilitá su classi.
Vediamo ora qualcuno degli algoritmi usati per classificare dati.

3.1 K-Nearest Neighbor

Questo è un algoritmo non parametrico, ovvero il numero di parametri cresce con la quantitá di dati di addestramento. Dato un set di dati di addestramento $(x_1, x_2, \cdots, x_n)$, corrispondenti ai risultati $(y_1, y_2, \cdots, y_n)$, questo metodo ,dato un nuovo punto z, cerca di prevedere la sua classe di appartenenza osservando, tra l'insieme di punti adiacenti, quelli a lui più vicini. Il numero di punti adiacenti da considerare dipende dal parametro k: si osservano le classi a cui appartengono i k punti piú vicini e la classe piú ricorrente sarà assegnata al punto z.
Si supponi di avere un set di dati che comprende N_k punti appartenenti alla classe C_k con N punti in totale, ovvero: $\sum_k N_k = N$. Se si vuole classificare un nuovo punto x, si disegna una sfera centrata in x contenente K punti qualsiasi, indipendentemente da come siano classificati.

Si vuole ora calcolare la funzione di probabilità di x che (essendo in un caso di classificazione i valori sono discreti) sarà la densità discreta:

$$P = \int_R p(x)dx \tag{3.1}$$

Si supponeni ora di aver raccolto un set di dati comprendente N osservazioni, ognuna con probabilitá uniforme $p(x)$ di essere all'interno della regione R, quindi la probabiltá di avere K punti all'interno di R sará data dalla distribuzione binomiale:

$$P(X = K) = \frac{N!}{K!(N-K)!}P^K(1-P)^N - K, \tag{3.2}$$

dove il valore atteso e la varianza sono date da:

$$\mathbb{E}[K] = NP \qquad\qquad Var[K] = NP(1-P) \tag{3.3}$$

Per N grandi, si applica il Teorema di De Moivre-Laplace alla eq.3.2 e si nota che la distribuzione binomiale si comporta come una distribuzione normale, con stessa media e varianza della binomiale, perciò possiamo assumere:

$$K \simeq NP \tag{3.4}$$

Ora se la regione R è sufficientemente piccola e la densità di probabilità $p(x)$ è approssimativamente costante in R, dall'eq 3.1 si ottiene:

$$P \simeq p(x)V \tag{3.5}$$

dove V è il volume della sfera R.
Combinando le eq. 3.4 e 3.5 otteniamo:

$$p(x) = \frac{K}{NV} \tag{3.6}$$

che fornisce le seguenti stime:

$$p(x|C_k) = \frac{K_k}{N_k V} \tag{3.7}$$

$$p(x) = \frac{K}{NV} \tag{3.8}$$

$$p(C_k) = \frac{N_k}{N} \tag{3.9}$$

quindi la formula del Teorema di Bayes diventa:

$$p(C_k|rx) = \frac{p(x|C_k)p(C_k)}{p(x)} = \frac{K_k}{K} \tag{3.10}$$

Per minimizzare la probabilità di errore di classificazione, bisogna assegnare al nuovo punto x la classe con probabilità più alta, ovvero quando il valore di $\frac{K_k}{K}$ è massimo.
L'obiettivo del metodo è quindi chiaro: per classificare un nuovo punto, si identificano i K punti più vicini all'insieme di allenamento e si assegna al nuovo punto la classe che ha il maggior numero di rappresentanti tra i K punti considerati.
Rimane solo la scelta della metrica da usare per calcolare la distanza tra i punti, solitamente viene usata la distanza Euclidea, ma questo dipende dal problema e dalla tipologia del dato da analizzare.
Lo svantaggio di questo algortimo è chiaro: il numero di distanze da calcolare aumenta con l'aumentare dell'insieme di addestramento. Oltre a rallentare il tempo di calcolo, si usa anche una considerevole quantità di memoria. Per ovviare a questo: si divide l'insieme di allenamento in sottoinsiemi di cardinalità n, dove n di solito è un divisore della cardinalità dell'insieme totale e prende il nome di *batch size*.
Diminuendo il numero di dati per l'addestramento, i calcoli e la memoria necessari diminuiscono notevolmente, in quanto i paragoni e i dati significativi da mantenere sono minori.

Capitolo 4

TensorFlow

TensorFlow (TF) è una libreria software open source, sviluppata da Google, utilizzata per implementare l'apprendimento automatico e i sistemi di deep learning. Essa fornisce API native in linguaggio: Python, C/C++, Java, Go, e RUST.
In questa tesi il linguaggio di programmazione usato è Python.
In generale un algoritmo scritto in TF rispetta la seguente struttura:

1. Importare ed analizzare l'insieme di dati;
2. Creare colonne di caratteristiche per descrivere i dati;
3. Selezionare il tipo di modello;
4. Provare il modello;
5. Valutare l'efficacia del modello;
6. Lasciare che il modello addestrato faccia previsioni (test).

Questa struttura è in linea con la descrizione di un generico algoritmo di apprendimento, descritto in (1.1). La vera innovazione di TF risiede nella descrizione del modello, poiché lo fa costruendo un grafico computazionale: *Data Flow Graph*. In questo grafico ogni nodo rappresenta l'istanza di un'operazione matematica, mentre ogni spigolo rappresenta un tensore, su cui vengono eseguite le operazioni.

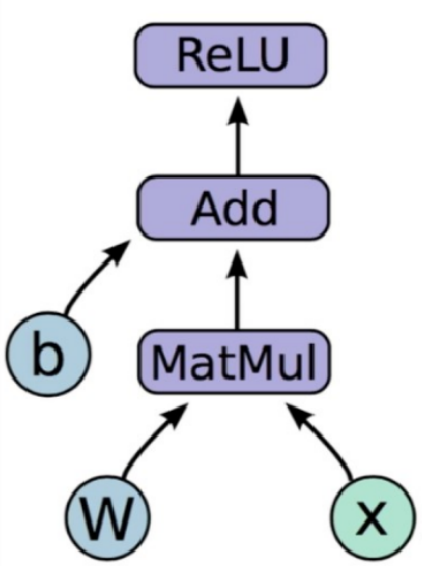


Figura 4.1: Data Flow Graph del calcolo di $ReLu(Wx + b)$.

Definizione 9 *Un **tensore** in TF è una matrice n-dimensionale di tipi di dati di base (es: float32, int32,string, ecc..). Viene chiamato tf.Tensor ed è descritto da tre parametri:*

1. *grado (rank);*
2. *corpo (shape);*
3. *tipo (type).*

Il **grado** di un oggetto tf.Tensor è il suo numero di dimensioni.
Il **corpo** di un tf.Tensore è il numero di elementi in ogni dimensione. TF automaticamente deduce il corpo durante la costruzione del grafico.
Il **tipo** è il tipo di dato a cui appartongono gli elementi del tensore.
I principali tipi di tensori sono:

- Variabili (*tf.Variable*): i parametri dell'algoritmo che verranno cambiati per ottimizzare l'algoritmo;
- Costanti (*tf.constant*);
- Segnaposto(*tf.placeholder*): consentono di inserire dati e di creare operazioni per costruire il grafico computazionale, possono dipendere da altri dati ad esempio il risultato previsto di un calcolo. Possono essere usati più volte e non dare lo stesso risultato;

- Tensore sparso (*tf.SparseTensor*).

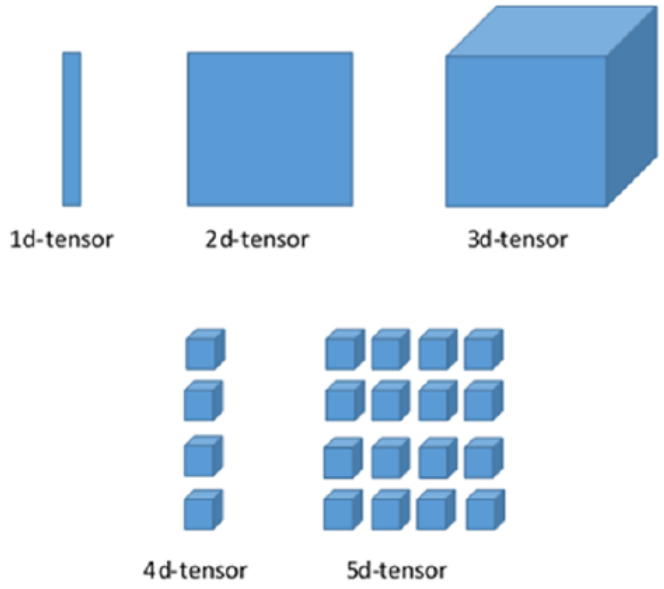


Figura 4.2: Strutture di tensori n-dimensionali, per alcune n.

Classificazione

Di seguito viene riportato un esempio di codice per la classificazione di numeri scritti a mano estratti dal dataset di MNIST. L'algoritmo utilizzato è dei K-nearest neighbour,in due versioni: la prima non utilizza il batch size mentre la seconda si.

Classificazione con k=5

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image
from tensorflow.examples.tutorials.mnist import input_data
sess=tf.Session()
mnist=input_data.read_data_sets("MNIST_data/",one_hot=True)
train_size=2000
test_size=200
train_set=np.random.choice(len(mnist.train.images),train_size,
    replace=False)
test_set=np.random.choice(len(mnist.test.images),test_size,
    replace=False)
x_vals_train = mnist.train.images[train_set]
x_vals_test = mnist.test.images[test_set]
y_vals_train = mnist.train.labels[train_set]
y_vals_test = mnist.test.labels[test_set]
k = 5
x_data_train=tf.placeholder(shape=[None,784],dtype=tf.float32)
x_data_test=tf.placeholder(shape=[None,784],dtype=tf.float32)
y_target_train=tf.placeholder(shape=[None,10],dtype=tf.float32)
y_target_test = tf.placeholder(shape=[None, 10],dtype=tf.float32)
distance=tf.reduce_sum(tf.abs(tf.subtract(x_data_train,
      tf.expand_dims(x_data_test,1))),2)
neighbour_xvals,neighbour_indices=tf.nn.top_k(
      tf.negative(distance),k=k)
predict_indices=tf.gather(y_target_train, neighbour_indices)
count_of_predict=tf.reduce_sum(predict_indices,reduction_indices=1)
prediction=tf.argmax(count_of_predict,axis=1)
test_output=[]
ok=0
init=tf.global_variables_initializer()
sess.run(init)
```

```
for i in range(len(x_vals_test)):
    predictions=sess.run(prediction,feed_dict={x_data_train:
        x_vals_train,x_data_test:x_vals_test,y_target_train:
        y_vals_train, y_target_test: y_vals_test})
    test_output.extend(predictions)
    print("test n:",i, " Classe predetta:",test_output[i],
        "Classe vera:",np.argmax(y_vals_test[i]),"\n\r")
    if test_output[i]== np.argmax(y_vals_test[i]):
        ok=ok+1
accuracy = ok/len(x_vals_test)
print("Accuracy on test set:",accuracy)
```

Usando un insieme di addestramento di cardinalità 2000, e uno di test di cardinalità 200, si è ottenuta un'accuratezza dell'89% e il tempo impiegato di esecuzione è di 19.398 secondi.

Versione con batch size=10

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image
from tensorflow.examples.tutorials.mnist import input_data
sess=tf.Session()
mnist=input_data.read_data_sets("MNIST_data/",one_hot=True)
train_size=2000
test_size=200
train_set=np.random.choice(len(mnist.train.images),
        train_size,replace=False)
test_set=np.random.choice(len(mnist.test.images),
       test_size,replace=False)
x_vals_train=mnist.train.images[train_set]
x_vals_test=mnist.test.images[test_set]
y_vals_train=mnist.train.labels[train_set]
y_vals_test=mnist.test.labels[test_set]
k=5
x_data_train=tf.placeholder(shape=[None,784],dtype=tf.float32)
x_data_test=tf.placeholder(shape=[None,784],dtype=tf.float32)
y_target_train=tf.placeholder(shape=[None,10],dtype=tf.float32)
y_target_test=tf.placeholder(shape=[None,10],dtype=tf.float32)
distance=tf.reduce_sum(tf.abs(tf.subtract(x_data_train,
    tf.expand_dims(x_data_test,1))),2)
neighbour_xvals,neighbour_indices=tf.nn.top_k(
```

```
        tf.negative(distance),k=k)
predict_indices=tf.gather(y_target_train,neighbour_indices)
count_of_predict=tf.reduce_sum(predict_indices,reduction_indices=1)
prediction=tf.argmax(count_of_predict,axis=1)
test_output=[]
actual_vals = []
batch_size=10
ok=0
init=tf.global_variables_initializer()
sess.run(init)
for i in range(int(np.ceil(len(x_vals_test)/batch_size))):
  min_index=i*batch_size
  max_index=min((i+1)*batch_size,len(x_vals_train))
  x_batch=x_vals_test[min_index:max_index]
  y_batch=y_vals_test[min_index:max_index]
  predictions=sess.run(prediction,feed_dict={x_data_train:
          x_vals_train,x_data_test:x_batch,
          y_target_train:y_vals_train,
          y_target_test: y_batch})
  test_output.extend(predictions)
  actual_vals.extend(np.argmax(y_batch, axis=1))
for i in range(len(x_vals_test)):
  print("test n:",i,"Classe predetta:",test_output[i],
        "Classe vera:", actual_vals[i],"\n\r")
  if test_output[i]==np.argmax(y_vals_test[i]):
    ok=ok+1
accuracy=ok/len(x_vals_test)
print("Accuracy on test set:",accuracy)
```

Utilizzando insiemi di addestramento e di test con le stesse cardinalità dell'implementazione precedente, ma inserendo il parametro di batch size, si ottiene una velocità di esecuzione maggiore: $15,535$ secondi ($\sim$ 4s in meno). In accordo con la spiegazione dell'algoritmo in 3.1, ci si aspettava un risultato di questo tipo. Purtroppo guadagnando velocità di esecuzione si è perso qualcosa nell'accuratezza: da 89% a 87%. Vale la pena sottolineare che, come detto più volte, non esiste un algoritmo migliore di un altro, ma il più adatto in base al problema da affrontare. Infatti se si aumenta il numero di esempi per l'insieme di addestramento e si inserisce il parametro di batch size, l'algoritmo sale di accuratezza pur mantenendo una velocità di esecuzione bassa.

Bibliografia

[1] Arthur Samuel. https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Is_Machine_Learning?lang=en, 1959.

[2] Shehzad Noor Taus Priyo. https://towardsdatascience.com/intro-to-deep-learning-d5caceedcf85, 2017.

[3] David H. Wolper and William G. Macready. No free lunch theorems for optimization, 1996.

[4] Roberto Ferretti. Appunti del corso di analisi numerica.

[5] Ryan Tibshirani. Error and validation.

[6] Frank L Lewis. *Reinforcement learning and approximate dynamic programming for feedback control*. Wiley, 2013.

[7] Sito Ufficiale dell'Universitá di Princetone. What is wordnet? https://wordnet.princeton.edu/.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[9] K.P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press, 2012.

[10] Kai Li Jia Deng Wei Dong Richard Socher, Li-Jia Li and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *Dept. of Computer Science, Princeton University, USA*, 2010.