

Artificial Neural Networks and Deep Learning

Antonino Elia Mandri
Stefano D'Angelo



POLITECNICO
MILANO 1863

Anno Accademico 2020/2021

Indice

1	Introduzione	4
1.1	Apprendimento automatico	4
1.2	Deep learning	5
1.3	Percettrone	7
1.3.1	Apprendimento Hebbiano	8
1.3.2	Feed Forward Neural Networks	8
1.3.2.1	Percettrone a singolo strato	8
1.3.2.2	Percettrone multistrato	9
1.3.3	Funzioni di attivazione	9
2	Reti Neurali	13
2.1	Teorema di approssimazione universale	14
2.2	Regressione	15
2.2.1	Variabili casuali	15
2.3	Classificazione	16
2.4	Maximum Likelihood Estimation (stimatore di massima verosimiglianza)	16
2.4.1	MLE per regressione	18
2.4.2	MLE per classificazione binaria	20
2.5	Ottimizzazione	22
2.5.1	Discesa del gradiente	22
2.5.2	Algoritmi di ottimizzazioni della discesa del gradiente	25
2.5.2.1	Momentum	28
2.5.2.2	Nesterov accelerated gradient (NAG)	29
2.5.2.3	Adagrad	30
2.5.2.4	Adadelta	31
2.5.2.5	RMSprop	32
2.5.2.6	Adam	32
2.5.2.7	AdaMax	33
2.5.2.8	Nadam	33
2.6	Backpropagation	35
2.6.1	Scomparsa del gradiente	37
2.7	Overfitting	38
2.7.1	Holdout cross validation	39

2.7.2	K-fold cross validation	39
2.7.3	Leave-one-out cross validation	40
2.7.4	Iperparametri	40
2.7.5	Weight decay	41
2.8	Altre tecniche di ottimizzazione	42
2.8.1	Shuffling e Curriculum Learning	42
2.8.2	Preprocessamento dei dati	42
2.8.2.1	Sottrazione della media	42
2.8.2.2	Normalizzazione	43
2.8.3	Batch Normalization	44
2.8.4	Early Stopping	45
2.8.5	Regolarizzazione loss function	46
2.8.6	Dropout	47
2.8.7	Inizializzazione dei pesi	48
2.8.8	Data augmentation	50
2.9	Scelte di non linearità	51
2.9.1	Funzione Sigmoida	51
2.9.2	Tangente iperbolica	53
2.9.3	ReLU	54
2.9.4	Leaky ReLU	56
3	Classificazioni di immagini	57
3.1	Dataset CIFAR-10	59
3.2	Nearest Neighbor Classifier	60
3.3	K-Nearest Neighbor Classifier	61
3.4	Classificatore Lineare (Linear Classifier)	63
3.4.1	Interpretazione di un Classificatore Lineare.	66
3.4.1.1	Interpretazione Geometrica	66
3.4.1.2	Template	67
4	Reti Convoluzionali	69
4.1	Inadeguatezza della struttura fully-connected	69
4.2	Operazione di convoluzione	70
4.3	Strati di una CNN	72
4.3.1	Convolutional Layer	72
4.3.2	Pooling layer	78
4.3.3	Fully Connected layer	79
4.4	Architettura generale di una CNN	80
4.5	Esempi architetture CNN	80
4.6	Data Augmentation	84
4.7	Transfer Learning	86
4.8	Segmentazione di immagini	87
4.8.1	Fully-Convolutional Neural Networks per la segmentazione semantica	88
4.8.2	U-Net	93
4.8.3	Global Averaging Pooling	95

4.9	Localizzazione	100
4.9.1	R-CNN	100
4.9.2	Fast R-CNN	101
4.9.3	Faster R-CNN	102
4.10	Altri esempi di architetture	104
4.10.1	YOLO (You Only Look Once)	104
4.10.2	Mask R-CNN	105
5	Reti Ricorrenti	106
5.1	Modellazione sequenziale	106
5.1.1	Sistemi dinamici lineari	107
5.1.2	Hidden Markov models	107
5.1.3	Recurrent Neural Networks	107
5.2	Vanishing Gradient	110
5.3	Long Short-Term Memories	111
5.4	Modellazione Sequence to Sequence	115
5.5	Macchine neurali di Turing	117
5.6	Trasformatore	126
5.7	Word embedding	128
6	Autoencoder	133
6.1	Struttura	133
6.2	Utilizzo delle reti autoencoder	135
6.2.1	Riduzione dimensionale	135
6.2.2	Eliminazione del rumore	135
6.2.3	Inizializzatore di pesi	136
6.2.4	Autoencoder generativi	137

Capitolo 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'informatica, l'apprendimento automatico è una variante alla programmazione tradizionale nella quale si pre-dispone in una macchina l'abilità di apprendere qualcosa dai dati in maniera autonoma, senza ricevere istruzioni esplicite a riguardo.

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.

Una definizione di cosa sia il machine learning fu data da Mitchell nel 97:

"A computer program is said to learn from experience E with respect to some class of task T and a performance measure P, if its performance at tasks in T, as measured by P, improves because of experience E"

1.2 Deep learning

Senza addentrarci nel complesso e variegato mondo del machine learning, sappiamo in linea generale che si tratta di algoritmi che fanno largo uso della statistica. Funzionano bene su un'ampia varietà di problemi. Tuttavia, tali algoritmi, non sono riusciti a risolvere i problemi centrali dell'IA, come il riconoscimento del linguaggio o il riconoscimento di oggetti e altri ancora. Ciò avviene sostanzialmente a causa dell'alta dimensionalità dei dati da trattare. I meccanismi usati dal machine learning per generalizzare risultano insufficienti per apprendere complesse funzioni multidimensionali. Il **Deep Learning**, la cui traduzione letterale significa apprendimento profondo, è una sottocategoria del Machine Learning e indica quella branca dell'Intelligenza Artificiale che fa riferimento agli algoritmi ispirati alla struttura e alla funzione del cervello chiamate **reti neurali artificiali**. Le architetture di Deep Learning (reti neurali artificiali) sono per esempio state applicate nella computer vision, nel riconoscimento automatico della lingua parlata, nell'elaborazione del linguaggio naturale, nel riconoscimento audio e nella bioinformatica. Potremmo definire il Deep Learning come un sistema che sfrutta una classe di algoritmi di apprendimento automatico che:

- usano vari livelli di unità non lineari a cascata per svolgere compiti di *estrazione di caratteristiche e di trasformazione*. Ciascun livello successivo utilizza l'uscita del livello precedente come input. Gli algoritmi possono essere sia di tipo supervisionato sia non supervisionato e le applicazioni includono l'analisi di pattern (apprendimento non supervisionato) e classificazione (apprendimento supervisionato);
- apprendono multipli livelli di rappresentazione che corrispondono a differenti livelli di astrazione; questi livelli formano una gerarchia di concetti.

La differenza principale consiste in come vengono estratte le caratteristiche utili alla risoluzione del problema. Tali caratteristiche vengono estratte manualmente nel machine learning , mentre vengono estratte autonomamente negli algoritmi di deep learning.

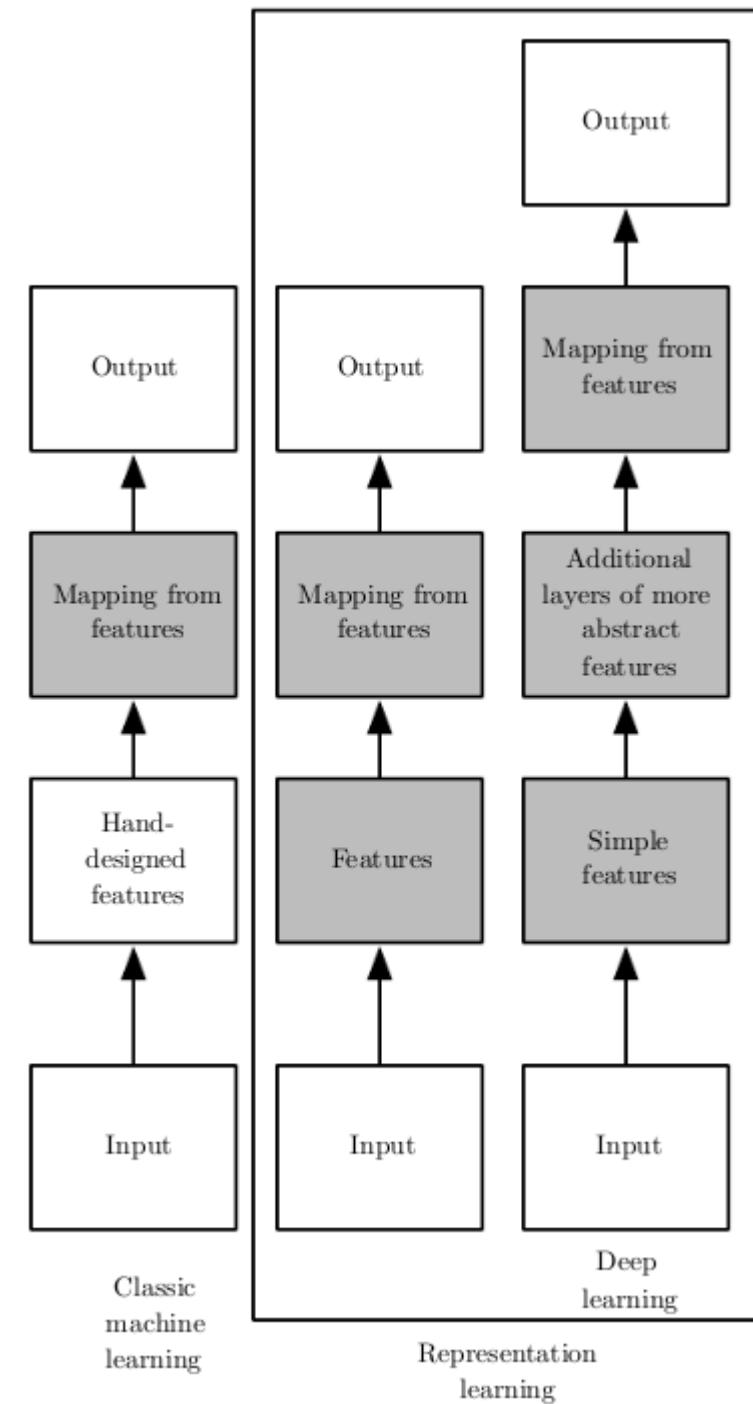


Figura 1.1: Differenze tra machine learning e deep learning. I riquadri evidenziati indicano le parti apprese dai dati

1.3 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**.

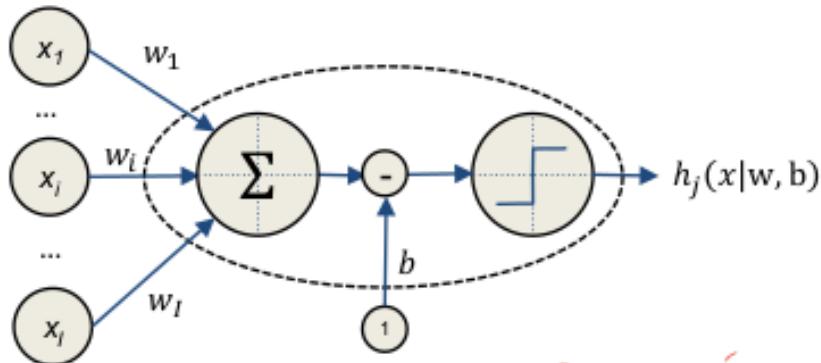


Figura 1.2: percettrone

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

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?

1.3.1 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 tutte 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.

Cercare
esempio

1.3.2 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 in tempi precedenti, per cui l'output è determinato solamente dall'attuale input.

1.3.2.1 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 uno strato in ingresso, seguito direttamente dall'uscita. Ogni unità di ingresso è collegata ad ogni unità di uscita. In pratica questo

tipo di rete 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 è in grado di rappresentare la funzione XOR.

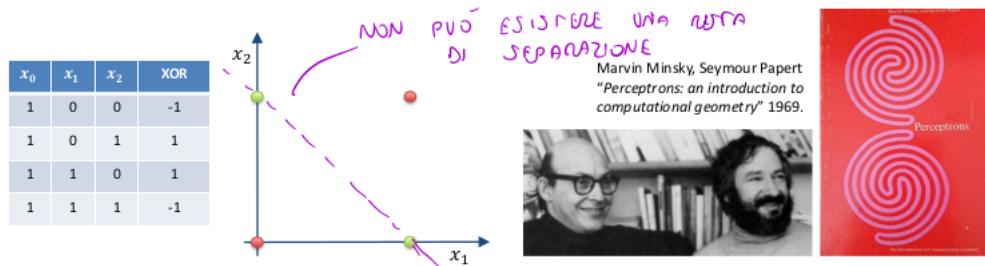


Figura 1.3: Problema XOR percettrone

1.3.2.2 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 progettazioni delle reti neurali è utile introdurre alcuni concetti.

1.3.3 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:

$$f(x) = x^+ = \text{ReLU}(x) = \max(0, x) \quad (1.4)$$

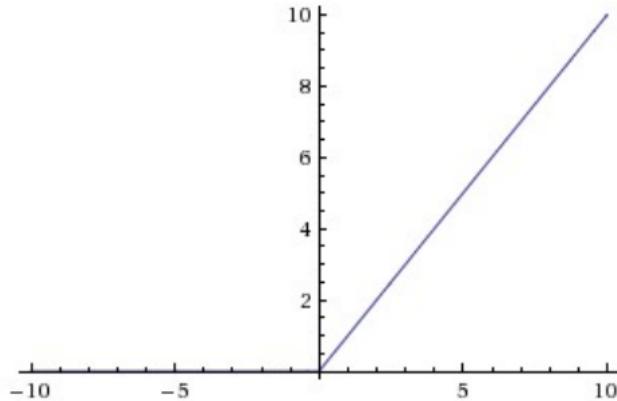


Figura 1.4: grafico ReLU

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.

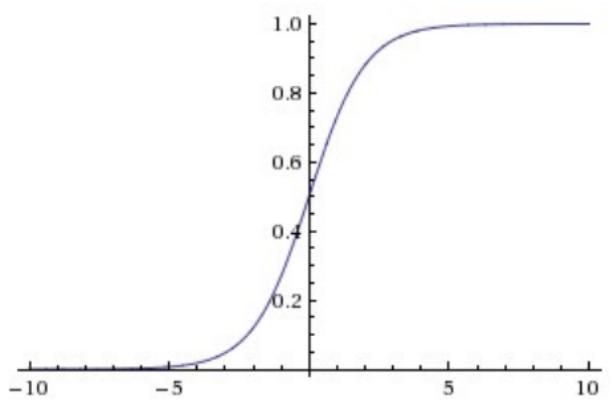


Figura 1.5: grafico sigmoide

Tangente iperbolica:

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

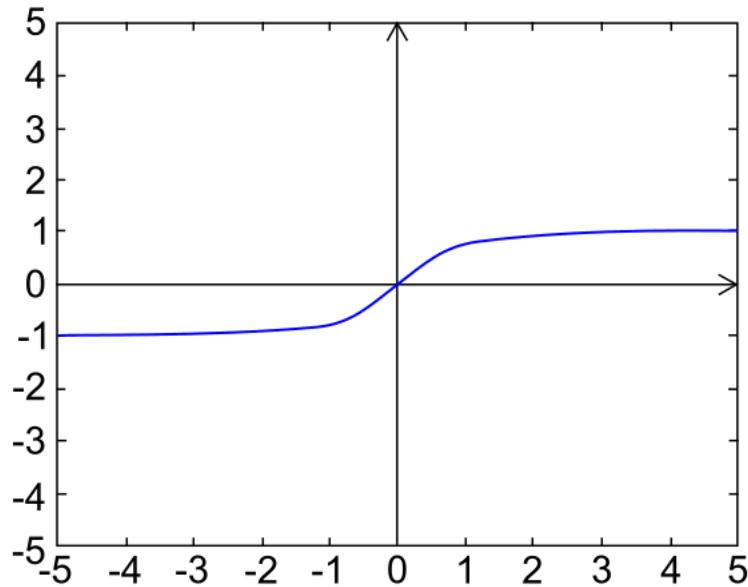


Figura 1.6: grafico tanh

Queste e altre funzioni verranno esaminate più attentamente dopo aver introdotto tecniche di ottimizzazione delle reti neurali.

Softmax In matematica, una funzione softmax, o funzione esponenziale normalizzata, è una generalizzazione di una **funzione logistica** che comprime un vettore $\mathbf{z} \in \mathbb{R}^k$ di valori reali arbitrari in un vettore $\sigma(\mathbf{z})$ di valori reali compresi in un intervallo $(0, 1)$ la cui somma è 1. La funzione è data da:

$$\sigma : \mathbb{R}^K \rightarrow \left\{ \mathbf{z} \in \mathbb{R}^K \mid z_i > 0, \sum_{i=1}^K z_i = 1 \right\}$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

Capitolo 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 in cui le reti neurali sono applicate possono essere classificate nelle seguenti categorie:

- 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 sotto insieme 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 espresiva 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 Re-

LU. 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:

$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon \quad (2.2)$$

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:

$$\int_{\mathbb{R}^n} |F_A(\mathbf{x}) - f(\mathbf{x})| d\mathbf{x} < \varepsilon. \quad (2.3)$$

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 disposizioni N osservazioni (coppie input-output della funzione t). Anche per la classificazioni di immagini è possibile ricondursi a variabili aleatorie.

2.3 Classificazione

La **classificazione statistica** è quell'attività che si serve di un algoritmo statistico al fine di individuare una rappresentazione di alcune caratteristiche di un'entità da classificare (oggetto o nozione), associandole una etichetta classificatoria. La classificazione si divide in due tipologie:

1. **Classificazione in due classi** $\{\Omega_0, \Omega_1\}$ in cui generalmente si utilizza come funzione di attivazione nel layer di output (singolo neurone):
 - (a) tanh codificando le classi nel modo seguente $\{\Omega_0 = -1, \Omega_1 = 1\}$. Ovviamente il codomino della funzione tanh varia con continuità nell'intervallo $(-1,1)$, quindi si prende come valore quello meno distante.
 - (b) Sigmoid codificando le classi nel modo seguente $\{\Omega_0 = 0, \Omega_1 = 1\}$. Analoga considerazione a riguardo del codominio fatta per tanh.
2. **Classificazione in più di due classi** $\{\Omega_0, \Omega_1, \dots, \Omega_n\}$, il layer di output ha tanti neuroni quante sono le classi. Un possibile approccio è codificare le classi usando la codifica onehot dove la classe i -esima viene rappresentata da un vettore di dimensione n in cui ogni componente è 0 ad esclusione dell' i -esimo elemento che vale 1. I neuroni dello strato di output useranno softmax come funzione di attivazione.

2.4 Maximum Likelihood Estimation (stimatore di massima verosimiglianza)

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:

- $\nabla_{\theta}(L) \circ \nabla_{\theta}(l) = 0$
- Controlliamo che il valore di θ_{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 θ 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:

$$\mathbf{x} = x_1, x_2, \dots, x_n \sim N(\mu, \sigma^2) \quad (2.4)$$

con

$$N(\mu, \sigma^2) = p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (2.5)$$

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

$$L = p(x_1, x_2, \dots, x_n | \mu, \sigma^2) = \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}} \quad (2.6)$$

passiamo al logaritmo:

$$l = \log\left(\prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right) = \sum_{i=1}^N \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_i-\mu)^2}{2\sigma^2}}\right) \quad (2.7)$$

$$= N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (x_i - \mu)^2 \quad (2.8)$$

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

$$\frac{\partial l(\mathbf{x}|\mu)}{\partial \mu} = \frac{\partial}{\partial \mu} \left(N \cdot \log\left(\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (x_i - \mu)^2 \right) \quad (2.9)$$

$$= \frac{1}{\sigma^2} \sum_{i=1}^N (x_i - \mu) \quad (2.10)$$

ora uguagliamo a zero la derivata parziale:

$$\frac{1}{\sigma^2} \sum_{i=1}^N (x_i - \mu) = 0 \quad (2.11)$$

$$\sum_{i=1}^N (x_i - \mu) = 0 \quad (2.12)$$

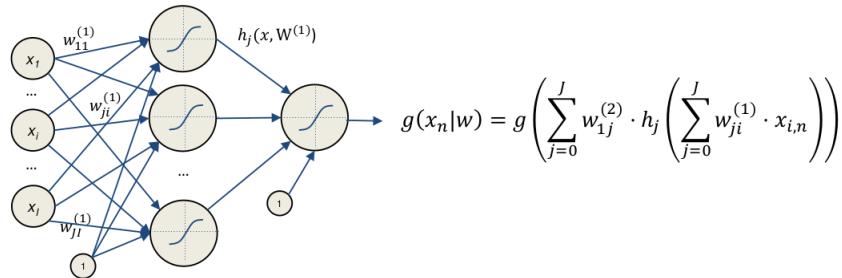
$$\sum_{i=1}^N x_i = \sum_{i=1}^N \mu \quad (2.13)$$

$$\mu_{MLE} = \frac{1}{N} \sum_{i=1}^N x_i. \quad (2.14)$$

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

2.4.1 MLE per regressione

Neural Networks for Regression



Goal: approximate a target function t having N observations

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

Figura 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.

$$t_n = g(x_n|w) + \epsilon_n \quad (2.15)$$

$$\epsilon_n \sim N(0, \sigma^2) \quad (2.16)$$

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

$$\Rightarrow t_n \sim N(g(x_n|w), \sigma^2). \quad (2.17)$$

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

$$p(t|g(x|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-g(x|w))^2}{2\sigma^2}} \quad (2.18)$$

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

$$L(w) = p(t_1, t_2, \dots, t_N | g(x|w), \sigma^2) = \prod_{i=1}^N p(t_i | g(x_i|w), \sigma^2) \quad (2.19)$$

$$= \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i-g(x_i|w))^2}{2\sigma^2}} \quad (2.20)$$

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

$$\operatorname{argmax}_w L(w) = \operatorname{argmax}_w \left(\prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i-g(x_i|w))^2}{2\sigma^2}} \right) \quad (2.21)$$

$$= \operatorname{argmax}_w \left(\sum_{i=1}^N \log \left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i-g(x_i|w))^2}{2\sigma^2}} \right) \right) \quad (2.22)$$

$$= \operatorname{argmax}_w \left(N \cdot \log \left(\frac{1}{\sqrt{2\pi}\sigma} \right) - \frac{1}{2\sigma^2} \sum_{i=1}^N (t_i - g(x_i|w))^2 \right) \quad (2.23)$$

$$= \operatorname{argmin}_w \left(\sum_{i=1}^N (t_i - g(x_i|w))^2 \right). \quad (2.24)$$

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.

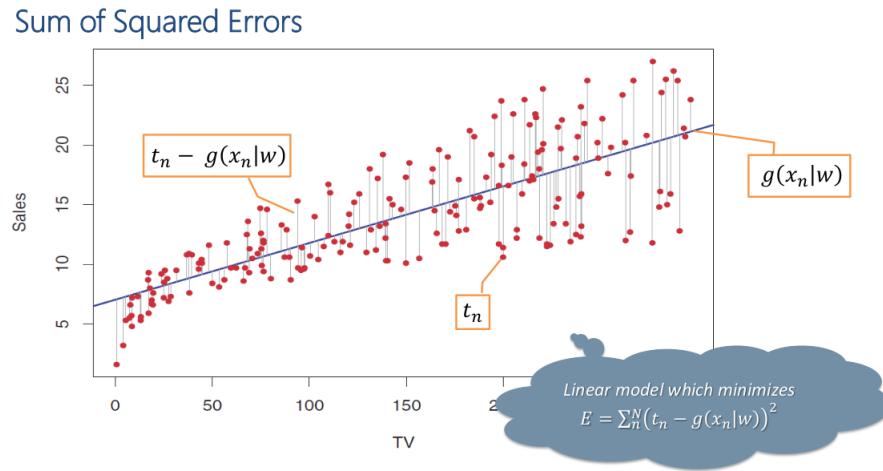
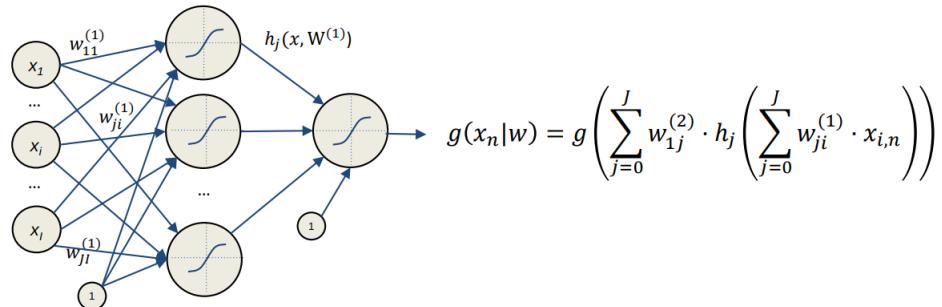


Figura 2.2: SSE

2.4.2 MLE per classificazione binaria

Usiamo MLE per determinare la **loss function** adatta al problema di classificazione binaria, la rete mostrata in figura utilizza la funzione sigmoide nell'ultimo neurone di conseguenza l'output varierà con continuità tra 0 e 1.

Neural Networks for Classification



Goal: approximate a posterior probability t having N observations

$$g(x_n|w) = p(t_n|x_n), \quad t_n \in \{0, 1\} \quad \rightarrow \quad t_n \sim Be(g(x_n|w))$$

Figura 2.3: Rete neurale per classificazione binaria

Adottiamo ancora un approccio probabilistico. Idealmente, in un problema di classificazione binaria, noi vorremmo che ad ogni input x_n corrisponda un solo numero $t_n \in \{0, 1\}$. Una distribuzione di probabilità su due soli valori 0 e 1 è la distribuzione di Bernoulli.

Una variabile aleatoria discreta X ha distribuzione di Bernoulli $\mathcal{B}(p)$ di parametro $p \in [0, 1]$ se e solo se:

$$P(X = 1) = p$$

$$P(X = 0) = 1 - p$$

La conoscenza del parametro p deriva da assunzioni sul fenomeno aleatorio in esame. Ad esempio nel lancio di una moneta non truccata sappiamo che p è uguale a 0,5 dato che assumiamo che le leggi fisiche che governano il moto di una moneta non hanno preferenze tra testa o croce. Nel nostro caso quello che cerchiamo è una probabilità a posteriori del valore assunto da t_n sapendo l'input x_n . L'idea è utilizzare come parametro della distribuzione la conoscenza acquisita dalla rete analizzando l'input.

$$t_n \sim \mathcal{B}(g(x_n|w))$$

Quello che vogliamo fare è massimizzare il numero di classificazioni corrette su tutto l'input. I nostri campioni osservati durante il training sono indipendenti ed identicamente distribuiti:

$$p(t_n|g(x_n|w)) = g(x_n|w)^{t_n} (1 - g(x_n|w))^{1-t_n}$$

Scriviamo la likelihood congiunta:

$$\begin{aligned} L(w) &= p(t_1, t_2, \dots, t_N|g(x|w)) = \prod_{n=1}^N p(t_n|g(x_n|w)) = \\ &= \prod_{n=1}^N g(x_n|w)^{t_n} (1 - g(x_n|w))^{1-t_n} \end{aligned}$$

Massimizzando rispetto a w:

$$\operatorname{argmax}_w \left(\prod_{n=1}^N g(x_n|w)^{t_n} (1 - g(x_n|w))^{1-t_n} \right)$$

Passiamo al logaritmo

$$\operatorname{argmax}_w \left(\sum_{n=1}^N t_n \log(g(x_n|w)) + (1 - t_n) \log(1 - g(x_n|w)) \right)$$

Massimizzare quest'ultima formula è equivalente a minimizzare:

$$\operatorname{argmin}_w \left(- \sum_{n=1}^N t_n \log(g(x_n|w)) + (1 - t_n) \log(1 - g(x_n|w)) \right)$$

In questo modo siamo giunti ad avere una funzione di errore da minimizzare per la classificazione binaria. Tale funzione di errore prende il nome di Cross-entropy.

$$E(w) = \sum_{n=1}^N t_n \log(g(x_n|w)) + (1 - t_n) \log(1 - g(x_n|w))$$

2.5 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 risponso sulla validità dei pesi impostati.

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.

2.5.1 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:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (2.25)$$

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

$$\frac{\partial f(\mathbf{x})}{x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n)}{h} \quad (2.26)$$

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:

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{x_1}, \frac{\partial f(\mathbf{x})}{x_2}, \dots, \frac{\partial f(\mathbf{x})}{x_n} \right) \quad (2.27)$$

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*:

$$\frac{f(x+h) - f(x-h)}{2h} \quad (2.28)$$

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

$$\text{minimizzare } f(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n. \quad (2.29)$$

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \eta_k \mathbf{p}_k \quad (2.30)$$

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.

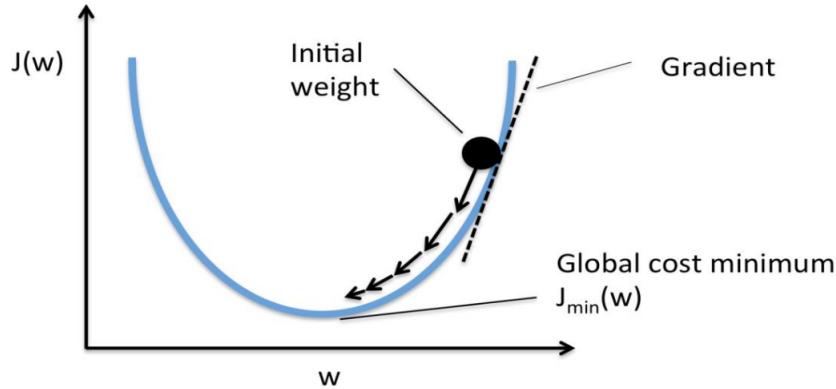


Figura 2.4: Discesa gradiente 1-D

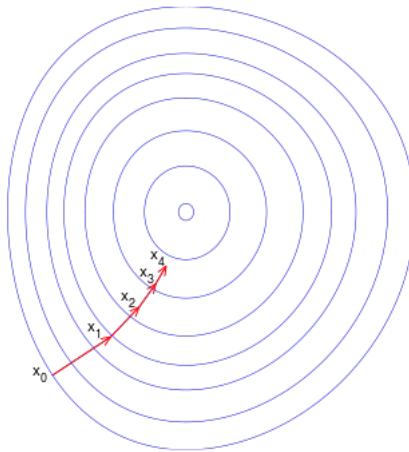


Figura 2.5: Discesa gradiente 2-D

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 η_k

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

$k = k + 1$

end.

2.5.2 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 $\boldsymbol{\theta}$ della funzione di costo sull'intero insieme di dati di allenamento (training dataset) ed esegue la media aritmetica:

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \quad (2.31)$$

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
```

In formule il gradiente è calcolato come:

$$\nabla_{\boldsymbol{\theta}} J(\mathbf{x}|\boldsymbol{\theta}) = \left(\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_2}, \dots, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_d} \right) \quad (2.32)$$

dove:

$$\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_i} = \frac{1}{N} \sum_{n=1}^N \frac{\partial J(x_n|\boldsymbol{\theta})}{\partial \theta_i} \quad (2.33)$$

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

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \cdot \nabla_{\boldsymbol{\theta}} J(x_i, y_i|\boldsymbol{\theta}). \quad (2.34)$$

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.

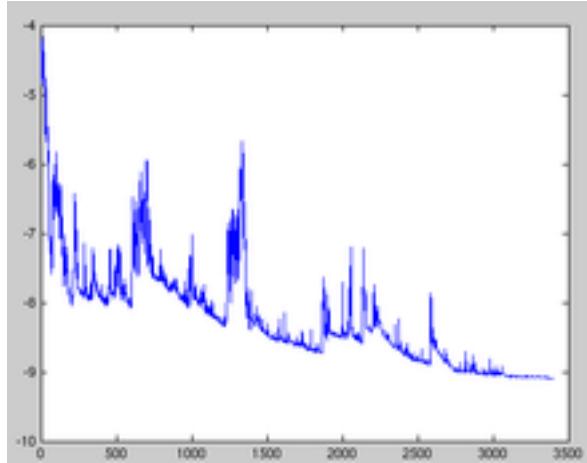


Figura 2.6: Fluttuazioni SGD

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
```

In formule:

$$\nabla_{\boldsymbol{\theta}} J(\mathbf{x}|\boldsymbol{\theta}) = \left(\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_2}, \dots, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_d} \right) \quad (2.35)$$

dove:

$$\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{\partial J(x_n|\boldsymbol{\theta})}{\partial \theta_i} \quad (2.36)$$

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

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \cdot \nabla_{\boldsymbol{\theta}} J(x_{i:i+n}, y_{i:i+n}|\boldsymbol{\theta}). \quad (2.37)$$

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.

```
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
```

In formule:

$$\nabla_{\boldsymbol{\theta}} J(\mathbf{x}|\boldsymbol{\theta}) = \left(\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_1}, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_2}, \dots, \frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_d} \right) \quad (2.38)$$

dove:

$$\frac{\partial J(\mathbf{x}|\boldsymbol{\theta})}{\partial \theta_i} \approx \frac{1}{M} \sum_{n \in \text{minibatch}} \frac{\partial J(x_n|\boldsymbol{\theta})}{\partial \theta_i} \quad (2.39)$$

Minibatch è un sottoinsieme del training set di cardinalità M .

Sfide. Mini-batch gradient descent non garantisce ancora ottime proprietà di convergenza, ma offre 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.

2.5.2.1 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.

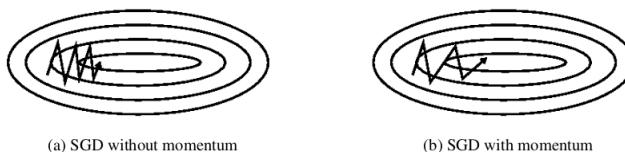


Figura 2.7: Momento

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:

$$\begin{cases} \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \cdot \nabla_{\theta} J(\theta) \\ \theta_t = \theta_{t-1} - \mathbf{v}_t \end{cases} \quad (2.40)$$

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.

2.5.2.2 Nesterov accelerated gradient (NAG).

Questo metodo 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 $\theta_{t-1} - \gamma \mathbf{v}_{t-1}$ 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.

$$\begin{cases} \mathbf{v}_t = \gamma \mathbf{v}_{t-1} + \eta \cdot \nabla_{\theta} J(\theta_{t-1} - \gamma \mathbf{v}_{t-1}) \\ \theta_t = \theta_{t-1} - \mathbf{v}_t \end{cases} \quad (2.41)$$

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

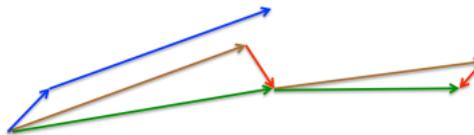


Figura 2.8: NAG update

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.

2.5.2.3 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. Sia:

$$g_t = \nabla_{\theta_t} J(\theta_t) \quad (2.42)$$

inoltre definiamo:

$$G_t = \begin{bmatrix} \sum_{j=0}^t (\frac{\partial j(\theta)}{\partial \theta_1})^2 & 0 & \dots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \sum_{j=0}^t (\frac{\partial j(\theta)}{\partial \theta_i})^2 & \vdots \\ 0 & \dots & \dots & \dots & \sum_{j=0}^t (\frac{\partial j(\theta)}{\partial \theta_d})^2 \end{bmatrix} \quad (2.43)$$

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \quad (2.44)$$

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 decadenza 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.

2.5.2.4 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 fissata 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:

$$E[g^2]_0 = \mathbf{0} \quad (2.45)$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma)g_t^2 \quad (2.46)$$

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:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \quad (2.47)$$

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

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \quad (2.48)$$

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

$$RMS[g]_t = \sqrt{E[g^2]_t + \epsilon} \quad (2.49)$$

dove RMS sta per root mean squared, allora

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \quad (2.50)$$

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:

$$E[\Delta\theta^2]_0 = \mathbf{0} \quad (2.51)$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1 - \gamma)\Delta\theta^2. \quad (2.52)$$

La root mean squared degli aggiornamenti dei parametri è:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}. \quad (2.53)$$

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:

$$\begin{cases} \Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t \\ \theta_{t+1} = \theta_t - \Delta\theta_t \end{cases}. \quad (2.54)$$

2.5.2.5 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:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2 \quad (2.55)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \quad (2.56)$$

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

2.5.2.6 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):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (2.57)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2. \quad (2.58)$$

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:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad (2.59)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (2.60)$$

L'aggiornamento diventa allora:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \quad (2.61)$$

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.

2.5.2.7 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.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)|g_t|^p \quad (2.62)$$

$$= (1 - \beta_2) \sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p. \quad (2.63)$$

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:

$$u_t = \lim_{p \rightarrow \infty} (v_t)^{1/p} = \lim_{p \rightarrow \infty} ((1 - \beta_2) \sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \quad (2.64)$$

$$= \lim_{p \rightarrow \infty} (1 - \beta_2)^{1/p} (\sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \quad (2.65)$$

$$= \lim_{p \rightarrow \infty} (\sum_{i=1}^t \beta_2^{(t-i)} |g_i|^p)^{1/p} \quad (2.66)$$

$$= \max(\beta_2^{t-1} |g_1|, \beta_2^{t-2} |g_2|, \dots, \beta_2 |g_{t-1}|, |g_t|) \quad (2.67)$$

che può essere riscritta ricorsivamente come:

$$u_t = \max(\beta_2 u_{t-1}, |g_t|). \quad (2.68)$$

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

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t \quad (2.69)$$

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

2.5.2.8 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.

$$g_t = \nabla_{\theta} J(\theta_t - \gamma m_{t-1})$$

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

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

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:

$$g_t = \nabla_{\theta} J(\theta_t) \quad (2.70)$$

$$m_t = \gamma m_{t-1} + \eta g_t \quad (2.71)$$

$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t) \quad (2.72)$$

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{m_t}{1 - \beta_1^t} \right) \quad (2.73)$$

$$= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{1 - \beta_1^t} \right) \quad (2.74)$$

$$= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right) \quad (2.75)$$

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

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t} \right) \quad (2.76)$$

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:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} (\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t}) \quad (2.77)$$

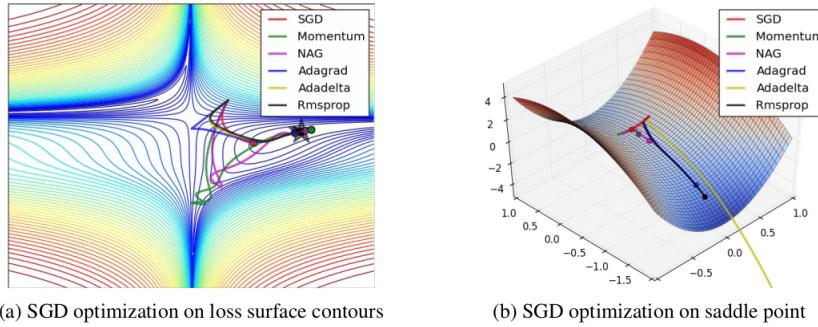


Figura 2.9: Confronto algoritmi

2.6 Backpropagation

In questa sezione svilupperemo una comprensione intuitiva della **backpropagation** che permette di calcolare agilmente il gradiente della funzione di costo (error function o loss function) di una rete neurale, tramite l'applicazione ricorsiva della regola di derivazione di funzioni composte, nota anche come **regola della catena**.

Derivazione funzione composta. Consideriamo due funzioni reali di variabile reale (la regola è valida in generale per tutte le funzioni differenziali, anche a più variabili con tutti gli adattamenti del caso) $f : \mathbb{R} \rightarrow \mathbb{R}$ e $g : \mathbb{R} \rightarrow \mathbb{R}$ e chiamiamole $y = g(x)$ e $z = f(y) = f(g(x))$ la loro composizione. Allora vale:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \quad (2.78)$$

(come se il differenziale dy si semplificasse nella moltiplicazione, con buona pace dei matematici). Sostituendo:

$$\frac{dz}{dx} = \frac{d}{dy} f(y) \cdot \frac{d}{dx} g(x) \quad (2.79)$$

$$= f'(y) \cdot g'(x) \quad (2.80)$$

$$= f'(g(x)) \cdot g(x) \quad (2.81)$$

In parole povere, la derivata della funzione composta $z = f(g(x))$ è data dalla derivata della funzione più esterna, con argomento invariato, moltiplicata per la derivata della funzione più interna.

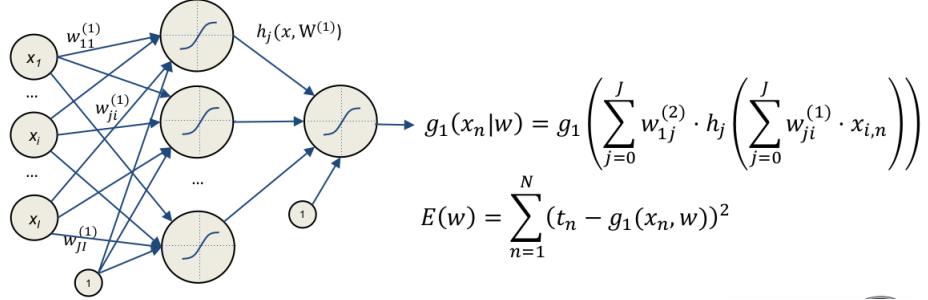


Figura 2.10: Rete neurale multistrato

In Figura 2.10 è mostrata una rete neurale che computa la funzione $g_1 : \mathbb{R}^I \rightarrow \mathbb{R}$ parametrizzata da w . La funzione da minimizzare è $E(w|t_n, x_n)$, notiamo che E è funzione di w e parametrizzata da t_n e x_n . Usando la regola della catena calcoliamo la derivata:

$$\frac{\partial E(w)}{\partial w_{ji}} = -2 \sum_{n=1}^N (t_n - g_1(x_n, w)) \cdot g'_1(x_n, w) \cdot w_{1j}^{(2)} \cdot h'_j \left(\sum_{j=0}^J w_{1j}^{(1)} \cdot x_{i,n} \right) \cdot x_i \quad (2.82)$$

Infatti notiamo:

$$\frac{\partial E(w)}{\partial g_1(x_n, w)} = -2 \sum_{n=1}^N (t_n - g_1(x_n, w)) \quad (2.83)$$

$$\frac{\partial g_1(x_n, w)}{\partial w_{1j}^{(2)} \cdot h_j(\bullet)} = g'_1(x_n, w) \quad (2.84)$$

$$\frac{\partial w_{1j}^{(2)} \cdot h_j(\bullet)}{\partial h_j(\bullet)} = w_{1j}^{(2)} \quad (2.85)$$

$$\frac{\partial h_j(\bullet)}{\partial w_{1j}^{(1)} \cdot x_{i,n}} = h'_j \left(\sum_{j=0}^J w_{1j}^{(1)} \cdot x_{i,n} \right) \quad (2.86)$$

$$\frac{\partial w_{1j}^{(1)} \cdot x_{i,n}}{\partial w_{1j}^{(1)}} = x_i \quad (2.87)$$

Allora:

$$\frac{\partial E(w)}{\partial w_{ji}} = \frac{\partial E(w)}{\partial g_1(x_n, w)} \cdot \frac{\partial g_1(x_n, w)}{\partial w_{1j}^{(2)} \cdot h_j(\bullet)} \cdot \frac{\partial w_{1j}^{(2)} \cdot h_j(\bullet)}{\partial h_j(\bullet)} \cdot \frac{\partial h_j(\bullet)}{\partial w_{1j}^{(1)} \cdot x_{i,n}} \cdot \frac{\partial w_{1j}^{(1)} \cdot x_{i,n}}{\partial w_{1j}^{(1)}}. \quad (2.88)$$

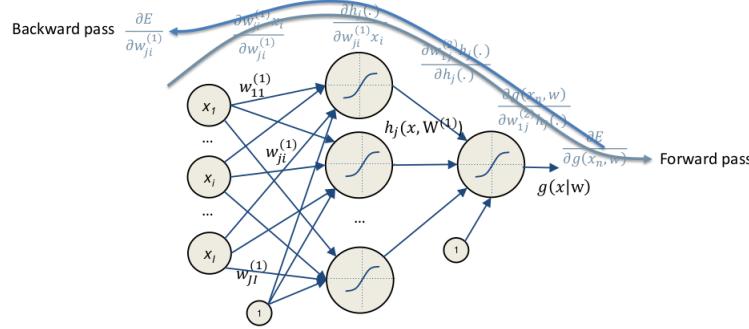


Figura 2.11: Backpropagation

Dal punto di vista pratico, la regola della catena è molto utile, permette facilmente di parallelizzare ed eseguire localmente il calcolo del gradiente e l'aggiornamento dei pesi. Ogni neurone ha la propria *funzione di attivazione* con la propria derivata. Possiamo calcolare le derivate singolarmente e propagare all'indietro semplicemente tramite un prodotto. Per esempio nella rete in Figura 2.10 il neurone di output calcola tutte le derivate parziali della sua funzione di attivazione rispetto ai propri parametri di ingresso, il risultato così ottenuto viene propagato allo strato precedente, in cui ogni neurone ha già calcolato le proprie derivate rispetto ai suoi parametri di input, le moltiplica con la derivata ricevuta dallo strato successivo e propaga all'indietro e così via fino allo strato di input. Inoltre quando un neurone esegue il prodotto tra le sue derivate e quella ricevuta dallo strato successivo, è in possesso del gradiente della funzione costo rispetto a tutti i suoi parametri e quindi procede ad aggiornare i pesi secondo una delle strategie viste nella precedente sezione.

2.6.1 Scomparsa del gradiente

I problemi che si possono incontrare usando la backpropagation sono molteplici e dipendono da vari fattori, come profondità della rete, funzioni di attivazione scelte, inizializzazione dei pesi e altri. Alcuni sono già stati accennati e trattati nei vari algoritmi di discesa del gradiente visti. Il problema della **scomparsa del gradiente** (in lingua inglese **vanishing gradient** problem) è un fenomeno che crea difficoltà nell'addestramento delle reti neurali profonde tramite retropropagazione dell'errore mediante discesa stocastica del gradiente. Come visto, ogni parametro del modello riceve ad ogni iterazione un aggiornamento proporzionale alla derivata parziale della funzione di perdita rispetto al parametro stesso. Una delle principali cause è la presenza di funzioni di attivazione non lineari classiche, come la *tangente iperbolica* o la *funzione logistica*, che hanno gradiente a valori nell'intervallo $(0, 1)$. Poiché nell'algoritmo di retropropagazione i gradienti ai vari livelli vengono moltiplicati tramite la regola della catena, il prodotto di n numeri in $(0, 1)$ decresce esponenzialmente rispetto alla profondità n della rete. Quando invece il gradiente delle funzioni di attivazione

può assumere valori elevati, un problema analogo che può manifestarsi è quello dell’esplosione del gradiente.

2.7 Overfitting

In statistica e in informatica, si parla di **overfitting** (in italiano: adattamento eccessivo, sovradattamento) quando un modello statistico molto complesso si adatta ai dati osservati (il campione) perché ha un numero eccessivo di parametri rispetto al numero di osservazioni. Un modello assurdo e sbagliato può adattarsi perfettamente se è abbastanza complesso rispetto alla quantità di dati disponibili. Si sostiene che l’overfitting sia una violazione del principio del Rasoio di Occam.

Apprendimento automatico Il concetto di overfitting è molto importante anche nell’apprendimento automatico e nel data mining. Di solito un algoritmo di apprendimento viene allenato usando un certo insieme di esempi (il training set appunto), ad esempio situazioni tipo di cui è già noto il risultato che interessa prevedere (output). Si assume che l’algoritmo di apprendimento (il learner) raggiungerà uno stato in cui sarà in grado di predire gli output per tutti gli altri esempi che ancora non ha visionato, cioè si assume che il modello di apprendimento sarà in grado di **generalizzare**. Tuttavia, soprattutto nei casi in cui l’apprendimento è stato effettuato troppo a lungo o dove c’era uno scarso numero di esempi di allenamento, il modello potrebbe adattarsi a caratteristiche che sono specifiche solo del training set, ma che non hanno riscontro nel resto dei casi; perciò, in presenza di overfitting, le prestazioni (cioè la capacità di adattarsi/prevedere) sui dati di allenamento aumenteranno, mentre le prestazioni sui dati non visionati saranno peggiori.

Contromisure Sia nella statistica sia nel apprendimento automatico, per prevenire ed evitare l’overfitting è necessario mettere in atto particolari accorgimenti tecnici, come la convalidazione incrociata e l’arresto anticipato, che indicano quando un ulteriore allenamento non porterebbe a una migliore generalizzazione.

Cross validation La convalidazione incrociata è un accorgimento che non permette direttamente di evitare l’overfitting ma ci permette di riconoscere quando avviene. L’idea è molto semplice e quasi sempre applicabile, a patto di avere un training set abbastanza grande, consiste nel rimuovere dal training set un certo numero di esempi e spostarli nel validation set. La rete si allena ancora normalmente sul training set, che risulterà ridotto, successivamente ad ogni epoca vengono sottoposti gli esempi del validation set che vengono processati dalla rete, quest’ultima durante la fase di validazione non si allena (non aggiorna i pesi) e viene monitorata la funzione di errore che in assenza di overfitting mostrerà un andamento simile alla funzione di errore valutata durante la

fase di allenamento, nel momento in cui la rete inizierà ad adattarsi eccessivamente noteremo che la funzione di errore valutata nel validation set inizierà ad aumentare mentre quando la valutiamo nel training set continuerà a decrescere.

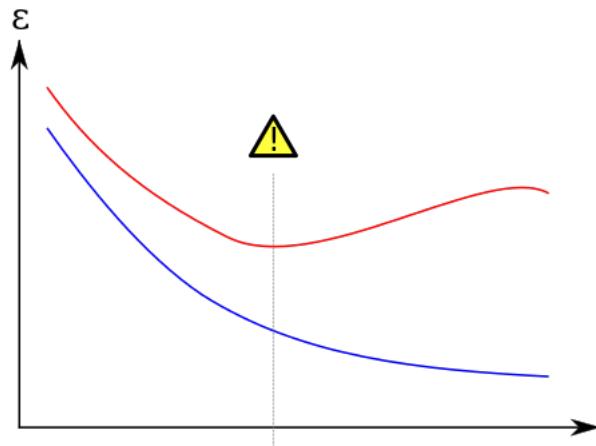


Figura 2.12: Overfitting: La curva blu mostra l'andamento dell'errore nel classificare i dati di training, mentre la curva rossa mostra l'errore nel classificare i dati di test o validazione. Una situazione in cui il secondo aumenta mentre il primo diminuisce è indice della possibile presenza di un caso di overfitting.

Esistono varie implementazioni della cross validation.

2.7.1 Holdout cross validation

È il metodo più semplice per implementare la **validazione incrociata**. Il data set viene diviso staticamente in due parti, **training set** e **validation set** e i due sottoinsiemi vengono usati come descritto prima.

2.7.2 K-fold cross validation

Il data set viene suddiviso in **k** sotto insiemi di egual dimensione (escluso al più l'ultimo), il training del modello viene ripetuto **k** volte ed ogni volta viene usato un sotto insieme come validation set, ogni volta ripartendo dallo stato iniziale. Alla fine delle **k** iterazioni i risultati delle validazioni vengono mediati per ottenere il risultato finale. Il vantaggio di questo metodo è che importa meno come i dati vengono divisi in quanto dopo **k** iterazioni tutto il dataset è stato usato almeno una volta come validation set (meno varianza). Lo svantaggio è l'enorme peso computazionale (training ripetuto **k** volte) per questo è quasi mai usato.

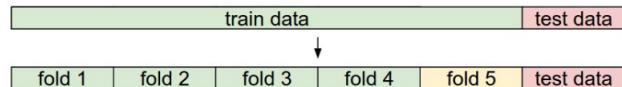


Figura 2.13: K-fold cross validation: in giallo il sottoinsieme di validazione

2.7.3 Leave-one-out cross validation

È una variante di k-fold dove k viene posto uguale a N cardinalità del dataset. Quindi la rete si allena su N-1 esempi e un solo esempio è usato per la validazione.

2.7.4 Iperparametri

Fino ad ora abbiamo visto molti tasselli che compongono una rete neurale ma non come combinarli assieme. Purtroppo ancora non è stata sviluppata una metodologia di progettazione che possa fornire linee guida solide per la creazione di una rete neurale per un determinato compito. Alcuni strumenti matematici possono aiutarci ad esempio come abbiamo fatto per trovare la funzione di errore per la regressione. Per altre grandezze invece va fatto ricorso all'esperienza, ad altri risultati pubblicati riguardanti reti che svolgono un compito simile e tanti, tantissimi tentativi ed errori. Per valutare la bontà di un modello possiamo usare l'**insieme di test** e ancora la cross validation. Quanti e quali sono quindi gli **iperparametri** che il progettista della rete deve scegliere? Molti, per esempio in una rete feedforward bisogna definire:

- **numero dei neuroni** e loro disposizione in strati;
- **funzione di attivazione** dei neuroni;
- **funzione di errore**;
- algoritmo di **discesa del gradiente**;

eventualmente anche:

- **dimensione batch**;
- **dimensione validation set** per la cross validation;
- molti altri a seconda della complessità della rete.

Come scegliere questi iperparametri?

- usare tutte le conoscenze pregresse e ipotesi sulla distribuzione dei dati da processare;
- sfruttare le conoscenze acquisite da altri nella risoluzione di compiti simili;
- usare la creatività!

Anche per la scelta degli iperparametri del modello è di aiuto la cross validation. Infatti possiamo usare (ed è altamente consigliato) la validazione per monitorare l'andamento della loss function e determinare la bontà del modello al variare degli iperparametri. È utile usare i dati di validazione al posto di quelli di test per evitare che le nostre scelte degli iperparametri siano condizionate e sovra adattate al test set, perdendo generalità. Modifichiamo gli iperparametri in base ai risultati sul validation set, se questi sono confermati anche dal test set possiamo ragionevolmente affermare che il modello funzioni. Se il test set non conferma allora abbiamo sovra adattato gli iperparametri al validation set (una qualche forma di overfitting manuale).

2.7.5 Weight decay

La regolarizzazione riguarda il vincolamento della “libertà” di un modello, sulla base di un’assunzione a-priori sul modello stesso, per ridurre l’overfitting. Finora abbiamo massimizzato la verosimiglianza dei dati

$$w_{MLE} = \operatorname{argmax}_w P(D|w) \quad (2.89)$$

questo approccio è detto frequentista. Usando, invece, un approccio **Bayesiano**, possiamo ridurre la libertà del modello. Usiamo, cioè, una **Maximum A-Posteriori verosimiglianza**:

$$w_{MAP} = \operatorname{argmax}_w P(w|D) \approx \operatorname{argmax}_w P(D|w) \cdot P(w) \quad (2.90)$$

dove $P(w)$ è la probabilità a-priori, che definisce lo spazio di ricerca dell’algoritmo. Si è osservato che piccoli pesi migliorano la capacità di generalizzazione delle reti neurali. Weight decay in pratica aggiunge una penalità proporzionale alla norma L2 dei pesi alla loss function, con il fine di imprimere una direzione al gradiente che riduca la norma dei pesi. Oltre all’evidenza empirica in taluni casi weight decay è supportato dalla teoria matematica usando **Maximum A-Posteriori verosimiglianza**. Analizziamo il caso già visto della regressione, ma utilizzando MAP e supponiamo che i pesi vengano inizializzati con una distribuzione $P(w) \approx N(0, \sigma_w)$ che come vedremo più avanti è assolutamente ragionevole come ipotesi.

$$\begin{aligned} \operatorname{argmax}_w P(D|w) \cdot P(w) &= \prod_{i=1}^N \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_i - g(x_i|w))^2}{2\sigma^2}} * \prod_{q=1}^Q \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(w_q)^2}{2\sigma^2}} \\ &= \operatorname{argmin}_w \left(\sum_{i=1}^N (t_i - g(x_i|w))^2 \right) + \sum_{q=1}^Q \frac{(w_q)^2}{2\sigma^2} \\ &= \operatorname{argmin}_w \left(\sum_{i=1}^N (t_i - g(x_i|w))^2 \right) + \gamma \sum_{q=1}^Q (w_q)^2 \end{aligned} \quad (2.91)$$

Come scegliere il parametro gamma? Una possibile strategia è tramite cross-validation.

2.8 Altre tecniche di ottimizzazione

2.8.1 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**.

2.8.2 Preprocessamento dei dati

Ci sono tre operazioni principali che possono essere eseguite sui dati di input prima di essere processati dalla rete (sia durante il training che durante il test). Ricordiamo che per noi l'input è in linea generale un vettore, quindi può essere visto come un punto nello spazio e l'insieme dei dati, rappresentato da una matrice $X \in \mathbb{R}^{N \times D}$ dove N è il numero dei dati, D la loro dimensione.

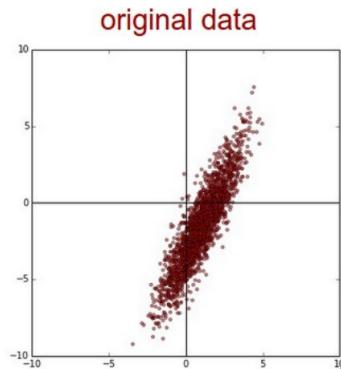


Figura 2.14: Dati originali

2.8.2.1 Sottrazione della media

Questa operazione ha il compito di centrare la media dell'insieme dei dati in zero. Ad ogni coordinata di tutti i vettori viene sottratta la media rispettiva. In formule:

sia \mathbf{x}_i l' i -esimo vettore appartente a X di dimensione D , $\mathbf{x}_i = (x_{i,1}, \dots, x_{i,D})$

$$\mathbf{y}_i = \mathbf{x}_i - \hat{\mathbf{x}} \quad (2.92)$$

dove

$$\hat{\mathbf{x}} = \left(\frac{1}{N} \sum_{i=1}^N x_{i,1}, \frac{1}{N} \sum_{i=1}^N x_{i,2}, \dots, \frac{1}{N} \sum_{i=1}^N x_{i,D} \right) \quad (2.93)$$

2.8.2.2 Normalizzazione

Ci riferiamo alla **normalizzazione** dei dati non in senso geometrico classico, che consiste nel dividere ogni vettore per la sua norma, ma come l'operazione di riscalamento di ogni dimensione in modo tale che ognuna di esse abbia circa la stessa scala. Per far ciò possiamo operare in due modi:

1. Dividiamo ogni dimensione di ogni vettore per la deviazione standard di quella dimensione (analogo a quanto fatto per la media). In formule:

$$\mathbf{y}_i = \frac{\mathbf{x}_i}{\sigma} \quad (2.94)$$

dove

$$\sigma = (\sqrt{\frac{1}{N} \sum (x_{i,1} - \hat{x}_1)^2}, \sqrt{\frac{1}{N} \sum (x_{i,2} - \hat{x}_2)^2}, \dots, \sqrt{\frac{1}{N} \sum (x_{i,D} - \hat{x}_D)^2}) \quad (2.95)$$

2. Dividiamo ogni dimensione di ogni vettore per la differenza tra la coordinata massima e minima rispettiva. In formule:

$$\mathbf{y}_i = \frac{\mathbf{x}_i}{\Delta} \quad (2.96)$$

dove

$$\Delta = (\max_i(x_{i,1}), \max_i(x_{i,2}), \dots, \max_i(x_{i,D})) \quad (2.97)$$

in questo modo l'input normalizzato avrà ogni componente compresa nell'intervallo $[-1, 1]$

Generalmente **sottrazione della media** e **normalizzazione** vengono effettuate in coppia. Ha senso normalizzare se si ha ragione di credere che l'input abbia features in scale differenti ma equamente importanti.

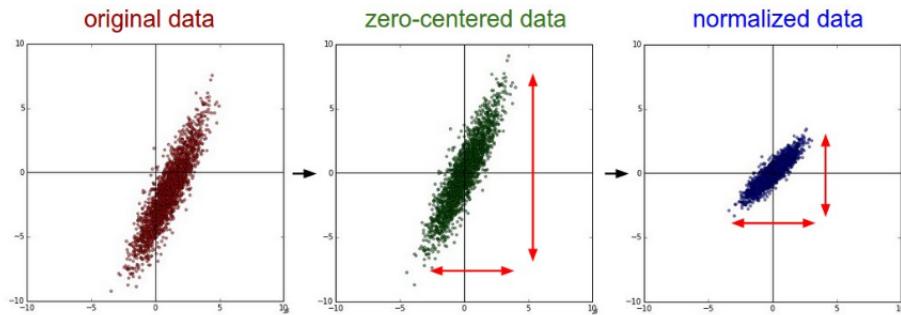


Figura 2.15: Preprocessamento completo

Attenzione: un punto importante da sottolineare nelle preelaborazioni è che qualsiasi statistica (es. media, varianza etc) deve essere calcolata solo sui

dati di addestramento e quindi applicata ai tutti i dati (allenamento, validazione, test). Ad esempio calcolare la media su tutti i dati e poi suddividere in allenamento, validazione e test sarebbe un errore. Invece le statistiche vanno calcolate solo sui dati di allenamento e poi applicate a tutti i dati.

2.8.3 Batch Normalization

Con l'aumento del numero di strati delle reti neurali permesso dalla sempre crescente potenza di calcolo dei computer odierni sono sorti vari problemi soprattutto durante l'addestramento, tra cui l'esplosione e la *scomparsa del gradiente*. Una delle possibili soluzioni a questi problemi consiste nel layer di **batch normalization**. Come già detto precedentemente, solitamente l'input di una rete viene *normalizzato* (cioè tutti gli input vengono riscalati per avere valori compresi in un range scelto, solitamente $[0,1]$) o *standardizzato* (cioè ai valori degli input gli viene sottratta la media e vengono divisi per la deviazione standard del dataset, per avere dei dati distribuiti con media 0 e deviazione standard 1). Ciò aiuta l'addestramento in quanto riduce il range dinamico dei dati in input a un range fisso, permettendo alla rete di estrarre feature più robuste e più velocemente. Tuttavia se la rete ha un elevato numero di layer, a seconda dei valori dei pesi l'output dei layer potrebbe tornare ad avere range dinamici ampi. Per ovviare a questo problema, si interpone un layer di batch normalization dopo il layer della rete da normalizzare. In questo modo non solo l'input della rete ma anche l'output dei vari layer viene standardizzato. Ogni layer di batch normalization ha **due pesi** (*per ogni batch*), un **fattore di scala** e un **bias**, che modificano l'output standardizzato permettendo di cambiarne media e deviazione standard. Questi pesi possono venire aggiustati durante l'addestramento. Il termine **batch** nel nome deriva dal gruppo di dati su cui viene effettuata la normalizzazione nel layer, che in questo caso è appunto un batch utilizzato durante l'addestramento con **Stochastic Gradient Descent (SGD)**. Per implementare la batch normalization:

si trova il valore medio per il batch corrente $B = \{x_1, \dots, x_m\}$, ovvero il valore medio prodotto da un particolare sub-strato della rete, prima di passare dalla funzione di attivazione non-lineare, quindi

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i$$

si trova la varianza per il batch corrente

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

si normalizza il valore con l'equazione

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Quindi, ad ogni valore si sottrae la media e si divide per una deviazione standard a cui si aggiunge il valore ϵ . Il valore ϵ è una costante positiva, ad esempio 0.001, che conferisce maggiore stabilità numerica (evita divisioni per zero) e incrementa, di poco, la varianza per ogni batch. Incrementare la varianza ha lo scopo di tenere in considerazione che la varianza della popolazione è maggiore di ogni campione preso dalla popolazione stessa. Il valore normalizzato viene poi moltiplicato per Gamma a sommato al parametro Beta, entrambi parametri che la rete apprenderà in fase di training:

$$y_i = \gamma \hat{x}_i + \beta \quad (2.98)$$

y_i è il valore normalizzato prodotto da ogni sub-strato della rete che passerà attraverso la funzione di attivazione, quali Sigmoid, Relu, Tanh ecc ecc. Durante il training il gradiente dovrà propagarsi a ritroso (backpropagation) attraverso questa trasformazione, affinché Beta e Gamma ricevano il segnale di errore e vengano ottimizzati. La fase di inferenza, rispetto alla fase di training, presenta alcune differenze: la rete, infatti, non viene esposta ad un batch di input ma ad un solo valore di input, per il quale dovrà produrre un output. Se utilizzassimo la medesima tecnica applicata durante il training dovremmo calcolare media e varianza su un singolo valore e non si produrrebbe quindi nessun risultato sensato. Per superare questo problema, la rete, durante la fase di testing, normalizza i valori di input utilizzando media e varianza stimati durante la fase di training. La Batch normalization si può utilizzare su **reti feed forward**, come su **reti convoluzionali** e **reti ricorrenti**. Per le reti ricorrenti, *lstm*, *gru* o *vanilla*, media e varianza vengono calcolate per ogni step di tempo anziché per ogni strato. Per le reti convoluzionali media e varianza vengono calcolate per ogni filtro. I vantaggi introdotti dalla batch normalization sono molteplici:

- Training della rete più veloce
- Si possono utilizzare tassi di apprendimento più alti
- L'inizializzazione dei pesi della rete può essere fatta con meno cautela
- Le funzioni di attivazione quali Sigmoid e Relu sono utilizzabili anche con reti maggiormente profonde
- Fornisce una sorta di regolarizzazione aggiuntiva e potrebbe ridurre la necessità di Dropout
- Miglioramento delle Performance in generale

2.8.4 Early Stopping

Consiste nel monitorare l'errore durante la fase di validazione. 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 un 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.

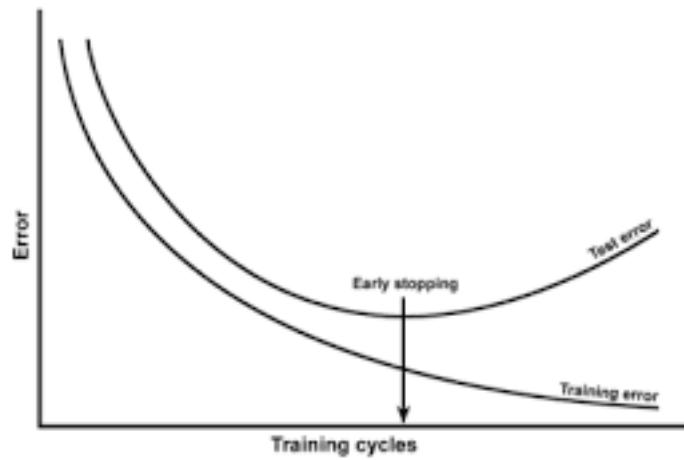


Figura 2.16: Early stopping

2.8.5 Regolarizzazione loss function

Le reti neurali, durante il loro processo di apprendimento, sfruttano, come abbiamo visto, la funzione costo per decidere come sistemare i propri parametri, ovvero pesi e bias. Abbiamo visto che c'è anche un grande problema, che è quello dell'overfitting. Sono state sviluppate alcune tecniche che, operando sulla funzione costo, aiutano a ridurre gli effetti del sovraallenamento di una rete neurale. Tali tecniche prendono il nome di tecniche di **regolarizzazione**. Generalmente, tali tecniche prevedono l'aggiunta di un fattore, dipendente dai pesi, dopo l'espressione della funzione costo. Tale fattore ha lo scopo di introdurre un termine di penalità sulla norma di w che ha l'effetto di restringere l'insieme entro cui vengono scelti i parametri. Ciò equivale, essenzialmente, ad imporre condizioni di regolarità sulla classe di funzioni realizzata dalla rete. Sia $E_p(w)$ la funzione costo (o loss function, error function) non regolarizzata, la funzione costo regolarizzata assume la forma:

$$E(w) = E_p(w) + \gamma \|w\| \quad (2.99)$$

dove $\|\bullet\|$ denota la norma euclidea L_2 . Per completezza aggiungiamo che è possibile usare anche altri tipi di norma, ma comunemente le più usate sono la

norma L_1, L_2 . Il termine $\gamma > 0$ è anch'esso un iperparametro. La regolarizzazione può essere vista come un *compromesso* tra trovare pesi piccoli e minimizzare la funzione costo. Il parametro γ viene detto **tasso di regolarizzazione** e serve per determinare il bilanciamento di tale compromesso: quando γ è piccolo, allora preferiamo minimizzare la funzione costo, mentre quando è grande, cerchiamo di trovare pesi piccoli. La regolarizzazione aiuta le reti neurali a generalizzare meglio, in quanto una rete con pesi piccoli non varia il proprio comportamento se cambiano alcuni dei dati di input. Questo le rende particolarmente difficile memorizzare le peculiarità dei dati, mentre la aiuta ad apprendere meglio quelli che sono i modelli e gli schemi dei dati di allenamento. Il principale problema della regolarizzazione è che non si è ancora capito esattamente il perché essa aiuti a migliorare le prestazioni di una rete neurale, ma abbiamo a disposizione solo evidenze pratiche di questo fatto. Nonostante questo, la regolarizzazione è ampiamente utilizzata e ci aiuta a migliorare le prestazioni delle nostre reti neurali.

2.8.6 Dropout

La tecnica di **dropout** invece funziona diversamente, in quanto modifica non la funzione di costo della rete, ma la rete stessa. Abbiamo visto il principio di funzionamento di una rete neurale e come essa riesca ad allenarsi. Ecco, questa tecnica prevede di applicare il solito procedimento togliendo prima una certa percentuale di neuroni in ogni hidden layer! Per ogni epoca di allenamento si sceglie casualmente con una data probabilità (iperparametro) quali neuroni tenere e quali scartare e si allena la rete così ottenuta. Si ripete quindi il procedimento, tenendo e scartando neuroni diversi ad ogni epoca: una volta che si ritiene che la rete sia pronta, si prende la rete originale e si aggiustano i pesi uscenti dai neuroni nascosti: abbiamo ottenuto una rete pronta a svolgere il proprio compito. In poche parole, è come se usassimo tante reti diverse e poi prendessimo come risultato la media di tutti i risultati di queste reti. Va tenuto ben presente che questo procedimento è applicato solo in fase di allenamento: durante il funzionamento vero e proprio, la rete è considerata nella sua interezza.

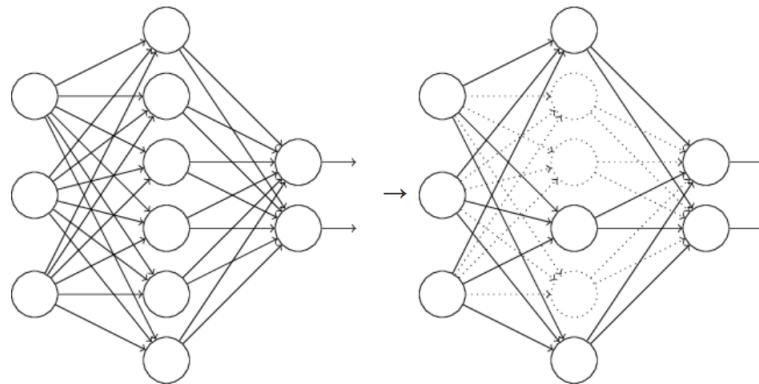


Figura 2.17: Dropout: ad ogni epoca, poi, faremo in modo di avere una rete diversa ogni volta, considerando neuroni diversi.

2.8.7 Inizializzazione dei pesi

Prima di partire con l'allenamento i pesi vanno inizializzati. Osserviamo che non sappiamo quale sarà il valore finale di ogni peso al termine dell'allenamento, ma se ad esempio normalizziamo i dati di input è ragionevole supporre che circa la metà dei pesi sarà positiva e la metà negativa, quindi con media tendente a zero.

- Una prima idea di **inizializzazione** è quella di porre tutti i pesi a **zero**. Pessima idea, infatti avendo tutti i pesi uguali, ogni neurone calcolerà lo stesso output e lo stesso gradiente, quindi i pesi subiranno tutti gli stessi aggiornamenti e i pesi finali resteranno tutti uguali tra loro. Questo è un problema, supponiamo che la nostra rete sia un classificatore di immagini, avere tutti i pesi uguali significa che ogni pixel di ogni immagine abbia la stessa importanza degli altri, quindi la rete non impara ad estrarre e riconoscere nessuna feature.
- **Inizializzazione con numeri grandi:** anche questa è una pessima idea, a seconda della funzione di attivazione del neurone abbiamo diversi comportamenti ma tutti problematici. Come detto prima, i pesi grandi peggiorano la capacità di generalizzazione della rete, inoltre se usassimo funzioni di attivazione come softmax o tangente iperbolica potremmo essere fin da subito nelle zone di saturazione, zone in cui il gradiente tende a zero bloccando di fatto l'apprendimento.
- **Inizializzare i pesi intorno allo zero:** abbandonata l'idea di inizializzare con numeri grandi, vogliamo che i pesi siano molto vicini allo zero, non identicamente nulli e distribuiti in modo che circa la metà sia positiva e metà sia negativa. Una pratica comune è inizializzare i pesi campionandoli da una distribuzione gaussiana a media nulla e varianza piccola

(0.01). Questo tipo di inizializzazione può andare bene se la rete neurale dispone di pochi strati nascosti, ma in reti neurali più profonde le attivazioni diventano sempre più piccole mano a mano che si processano i vari strati, fino ridursi a quantità praticamente nulle. Questo diventa un problema in quanto durante la fase di back propagation il gradiente accumulato continua ad essere moltiplicato per quantità piccolissime che portano alla sua dissolvenza.

- **Inizializzazione di Xavier:** l'idea è quindi quella di avere una distribuzione delle attivazioni tale che la rete neurale sia in grado di apprendere in maniera efficiente. In quest'ottica, Xavier (2010) ha proposto una inizializzazione dei pesi secondo una normale a media nulla con deviazione standard tale che la varianza delle attivazioni $a^{(k)}$ risulti essere unitaria. Sotto l'assunzione di attivazioni lineari e simmetriche (plausibile dal momento che anche la tangente iperbolica ha proprio questo comportamento intorno allo zero) questo si traduce nel rendere unitaria la varianza degli input $z^{(k)}$. L'inizializzazione di Xavier va eseguita neurone per neurone a partire dallo strato iniziale, per questo è necessario preprocessare l'input della rete come descritto sopra (input a media nulla e varianza unitaria) quando vogliamo utilizzare questa inizializzazione. Supponiamo di avere un input X di dimensione n e lo strato, composto da n neuroni sia solamente connesso tramite pesi casuali W di dimensione $n \times n$, l'output Y dell'i-esimo neurone, di dimensione n anch'esso sarà:

$$Y_i = X_1 W_{i,1} + X_2 W_{i,2} + \dots + X_n W_{i,n} \quad (2.100)$$

dove $i \in [1, n]$ e indica l'i-esimo neurone e W_i il vettore dei pesi associati. La varianza del prodotto di due variabili aleatorie è:

$$\text{Var}(W_i X_i) = E[X_i]^2 \text{Var}(W_i) + E[W_i]^2 \text{Var}(X_i) + \text{Var}(W_i) \text{Var}(X_i)$$

Per ipotesi sia l'input sia i pesi sono a media nulla, semplificando:

$$\text{Var}(W_i X_i) = \text{Var}(W_i) \text{Var}(X_i)$$

Inoltre assumiamo che X_i, W_i siano indipendenti ed identicamente distribuite (iid), allora:

$$\begin{aligned} \text{Var}(Y_i) &= \text{Var}(X_1 W_{i,1} + X_2 W_{i,2} + \dots + X_n W_{i,n}) \\ &= n \text{Var}(W_i) \text{Var}(X_i) \end{aligned} \quad (2.101)$$

Assumendo $\text{Var}(X_1) = 1$ perché l'input può essere o i dati di allenamento già normalizzati o essere l'output dello strato precedente in cui i pesi sono già stati inizializzati correttamente. Imponiamo quindi:

$$n \text{Var}(W_i) = 1$$

$$\text{Var}(W_i) = \frac{1}{n} = \frac{1}{n_{in,i}}$$

Quindi in definitiva Xavier propone di inizializzare i pesi campionandoli nel modo seguente:

$$W_i \sim N(0, \frac{1}{n_{in,i}}) \quad (2.102)$$

Fino ad adesso abbiamo assunto l'ipotesi che la funzione di attivazione sia lineare e simmetrica intorno allo 0. Usando la funzione ReLU cade l'ipotesi di simmetria e la distribuzione va adattata. A tale scopo recentemente è stata proposta l'inizializzazione:

$$W_i \sim N(0, \frac{2}{n_{in,i}}) \quad (2.103)$$

Il fattore di correzione 2 ha senso: la ReLU dimezza di fatto gli input, e pertanto bisogna raddoppiare la varianza dei pesi per mantenere la stessa varianza delle attivazioni.

- **Glorot & Bengio** seguendo un simile ragionamento a quello di Xavier ma applicato alla backpropagation, quindi procedendo a ritroso dall'ultimo strato al primo, invertendo anche il ruolo di input e output, hanno trovato che deve valere:

$$n_{in,i}Var(W_i) = 1$$

$$n_{out,i}Var(W_i) = 1$$

Imporre le due condizioni simultaneamente risulta essere troppo restrittivo, imporrebbe infatti $n_{in,i} = n_{out,i}$ che nel caso di una rete feedforward totalmente connessa significa che tutti gli strati hanno la medesima ampiezza e anche in altre tipologie di reti è una forte imposizione sull'architettura. Si è raggiunto un compromesso tra i due vincoli usando una distribuzione

$$W_i \sim N(0, \frac{2}{n_{in,i} + n_{out,i}}) \quad (2.104)$$

2.8.8 Data augmentation

Si può anche pensare di **ampliare** artificialmente **i dati di allenamento**, in quanto ottenere nuovi dati per allenare la rete è sempre una buona idea. Il problema è che non sempre è possibile, oppure è troppo costoso ottenerne di nuovi. Quindi se ne generano di nuovi a partire da quelli che abbiamo già a disposizione. Ad esempio, se la nostra rete dovesse riconoscere delle cifre scritte a mano, potremmo applicare delle piccole rotazioni o delle lievi dilatazioni o restrizioni ai dati che abbiamo già in possesso, creando delle immagini nuove da fornire alla nostra rete neurale. In generale, si cerca di espandere il set di allenamento cercando di riprodurre quelle che sono le variazioni che di solito hanno nella pratica.



Figura 2.18: Data augmentation: Nonostante la differenza sia minima, per l'analisi svolta dalla rete sono due immagini significativamente diverse.

2.9 Scelte di non linearità

Possiamo ora analizzare più nel dettaglio funzioni di attivazioni accennate nel capitolo 1 avendo ora maggiori conoscenze sull'ottimizzazione.

2.9.1 Funzione Sigmoide

Tale soluzione è stata progressivamente accantonata negli ultimi anni per via di alcune problematiche che comporta a livello pratico.

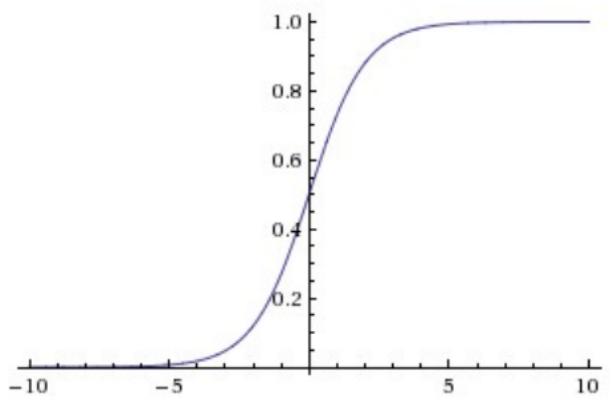


Figura 2.19: Grafico Sigmoide

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

La prima e più importante è quella relativa alla dissolvenza del gradiente in seguito alla *saturazione* dei neuroni, ossia quei neuroni che presentano valori di

output agli estremi del codominio della funzione di attivazione, in questo caso $(0, 1)$. È facile infatti notare che

$$\lim_{x \rightarrow +\infty} f(x) = 1$$

$$\lim_{x \rightarrow -\infty} f(x) = 0$$

Tale saturazione diventa problematica durante le fasi di back propagation, in quanto il gradiente locale assume valori prossimi allo zero, che per la *regola della catena* vanno a moltiplicare tutti i gradienti calcolati in precedenza e quelli successivi, conducendo così all'annullamento del gradiente globale.

$$f'(x) = f(x)(1 - f(x))$$

$$\lim_{x \rightarrow +\infty} f'(x) = 0$$

$$\lim_{x \rightarrow -\infty} f'(x) = 0$$

In pratica, si ha un flusso utile del gradiente solo per valori di input che rimangono all'interno di una zona di sicurezza, cioè nei dintorni dello zero. Il secondo problema deriva invece dal fatto che gli output della funzione sigmoide non sono centrati intorno allo zero, e di conseguenza gli strati processati successivamente riceveranno anch'essi valori con una distribuzione non centrata sullo zero. Questo influisce in maniera significativa sulla discesa del gradiente, in quanto gli input in ingresso ai neuroni saranno sempre positivi, e pertanto il gradiente dei pesi associati diventerà, durante la fase di back propagation, sempre positivo o sempre negativo. Tale risultato si traduce in una dinamica a zig-zag negli aggiornamenti dei pesi che rallenta in maniera significativa il processo di convergenza.

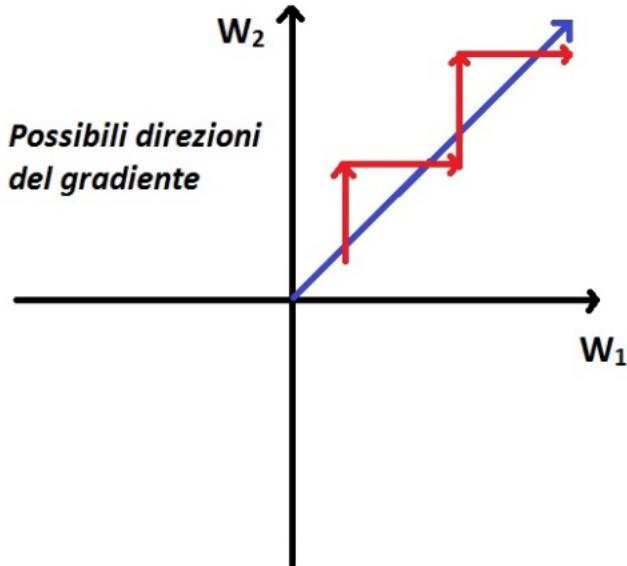


Figura 2.20: Esempio di dinamica a zig-zag nell’aggiornamento di due pesi W_1 e W_2 . La freccia blu indica l’ipotetico vettore ottimale per la discesa del gradiente, mentre le frecce rosse i passi di aggiornamento compiuti: gradienti tutti dello stesso segno comportano due sole possibili direzioni di aggiornamento.

È importante comunque notare che una volta che i gradienti delle singole osservazioni vengono sommati all’interno dello stesso batch di dati l’aggiornamento finale dei pesi può avere segni diversi, permettendo quindi di muoversi lungo un insieme più ampio di direzioni.

Il terzo e ultimo difetto della funzione sigmoide è che l’operazione $\exp(\cdot)$ al denominatore è molto costosa dal punto di vista computazionale, soprattutto rispetto alle alternative che verranno presentate di seguito.

2.9.2 Tangente iperbolica

Il problema degli output non centrati sullo zero della sigmoide può essere risolto ricorrendo all’utilizzo della tangente iperbolica, la quale presenta codominio $(-1, 1)$ centrato sull’origine degli assi.

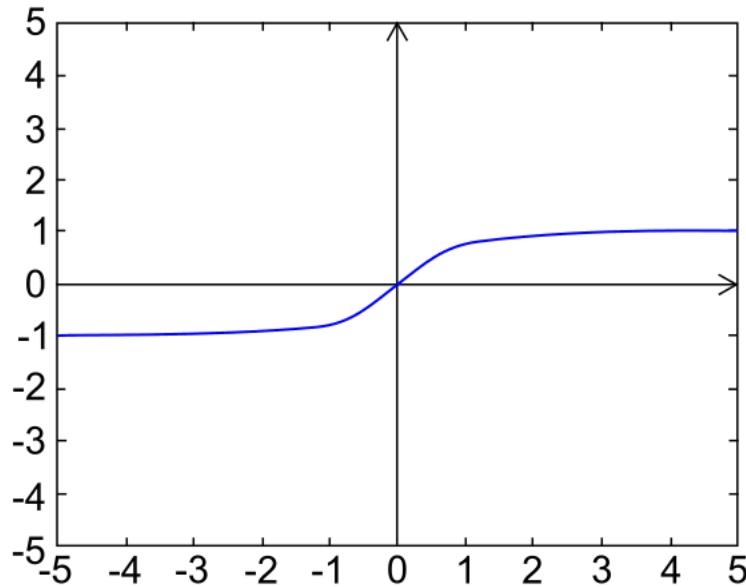


Figura 2.21: Grafico tangente iperbolica

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

Tuttavia, rimane il problema della *saturazione* dei neuroni, anzi viene addirittura accentuato, dal momento che la zona di sicurezza risulta ancora più ristretta. Rimane anche il problema della complessità computazionale della funzione esponenziale.

2.9.3 ReLU

La **Rectifier Linear Unit** (ReLU) è diventata popolare negli ultimi anni per via dell'incremento prestazionale che offre nel processo di convergenza: velocizza infatti di circa 6 volte la discesa del gradiente rispetto alle alternative viste finora.

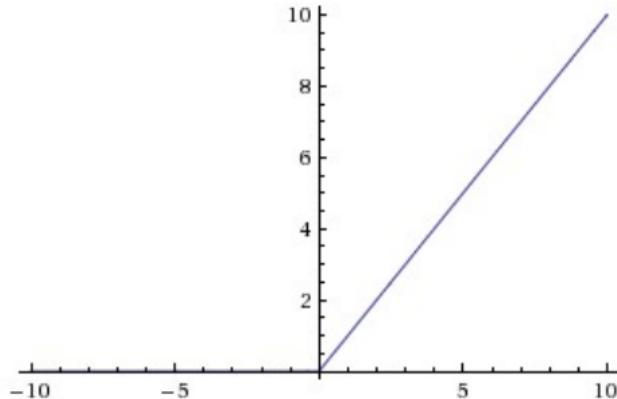


Figura 2.22: Grafico ReLU

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

Questo risultato è da attribuire in larga parte al fatto che la ReLU risolve il *problema della dissolvenza* del gradiente, non andando a saturare i neuroni. Durante la fase di back propagation infatti, se il gradiente calcolato fino a quel punto è positivo questo viene semplicemente lasciato passare, perché la derivata locale per il quale viene moltiplicato è pari ad uno.

$$f'(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

NB durante la back propagation l'input x è il gradiente proveniente dagli strati successivi.

Eventuali problemi sorgono invece quando il gradiente accumulato ha segno negativo, in quanto questo viene azzerato (la derivata locale è nulla lungo tutto il semiasse negativo) con la conseguenza che i pesi non vengono aggiornati. Fortunatamente questo problema può essere alleviato attraverso l'utilizzo di un algoritmo SGD: considerando più dati alla volta c'è infatti la speranza che non tutti gli input del batch provochino l'azzeramento del gradiente, tenendo così in vita il processo di apprendimento del neurone. Al contrario, se per ogni osservazione la ReLU riceve valori negativi, allora il neurone "muore", e non c'è speranza che i pesi vengano aggiornati. Valori elevati del learning rate amplificano questo problema, dal momento che cambiamenti più consistenti dei pesi si traducono in una maggiore probabilità che questi affondino nella "zona morta".

2.9.4 Leaky ReLU

La **leaky ReLU** è un tentativo di risolvere il problema della disattivazione dei neuroni comportato dalla ReLU classica, e consiste nell'introdurre una piccola **pendenza negativa** (di circa 0.01) α nella regione dove la ReLU è nulla, dove α è costante.

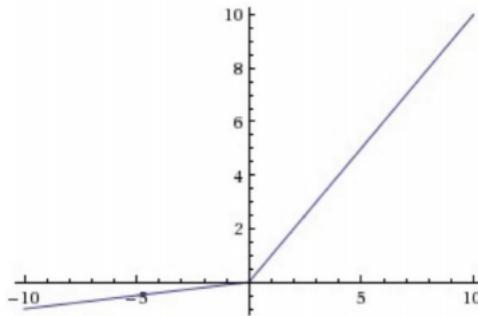


Figura 2.23: Grafico Leaky ReLU

$$f(x) = \max(\alpha x, x)$$

In alcune varianti, α può essere un parametro da stimare, al pari degli altri pesi della rete (si parla di **Parametric ReLU**), oppure una variabile casuale: è il caso della **Randomized ReLU**, dove ad ogni iterazione la pendenza della parte negativa della funzione viene scelta casualmente all'interno di un range prefissato. In alcuni recenti esperimenti, Bing Xu et al. (2015) hanno mostrato come le varianti della ReLU classica siano in grado di aumentare le performance finali del modello in termini di accuratezza, prima su tutte la RReLU, che grazie alla sua natura casuale sembra particolarmente portata alla riduzione del sovraccarico.

Capitolo 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.

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.

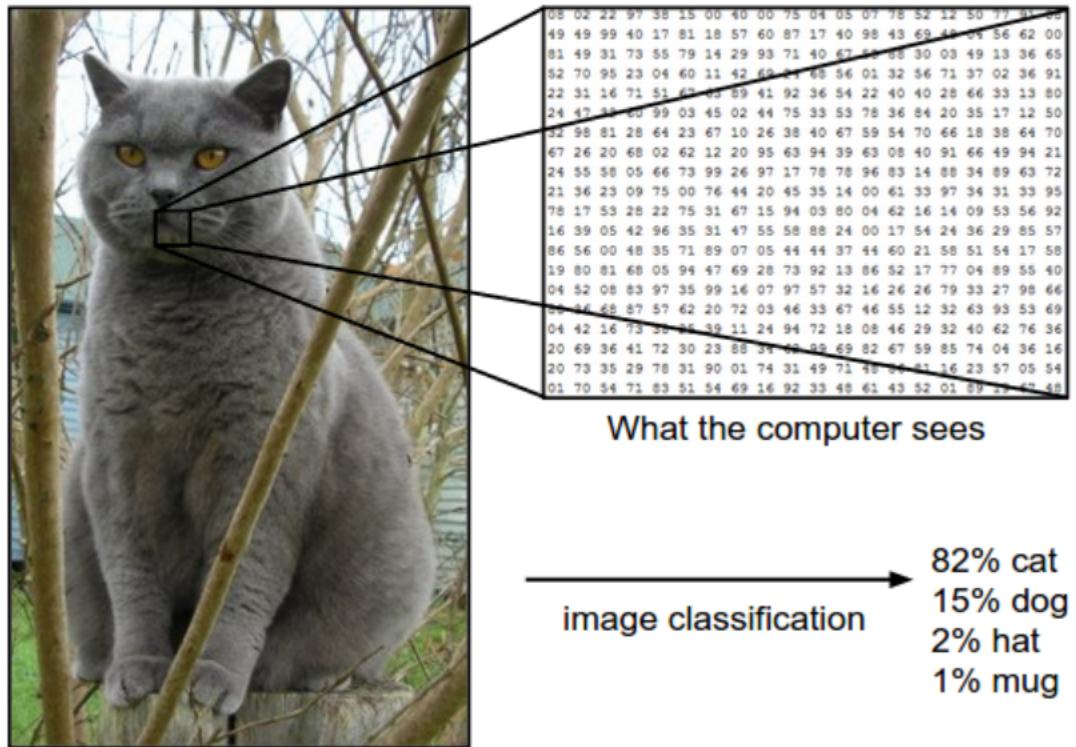


Figura 3.1: Classificazione di un'immagine

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;

- 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.

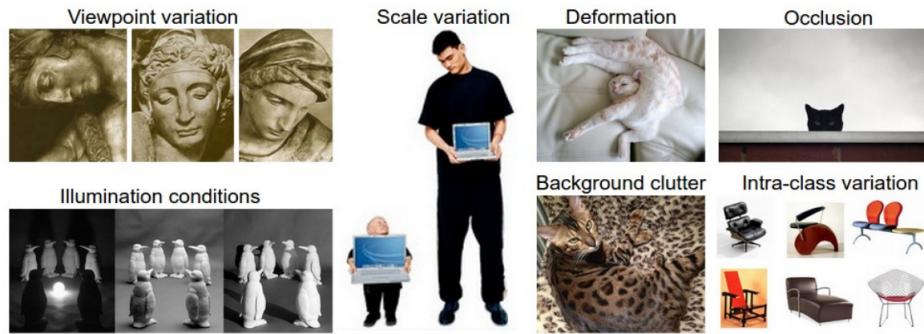


Figura 3.2: Problemi nella classificazione di immagini

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 $\mathbb{R}^{32 \times 32 \times 3}$.

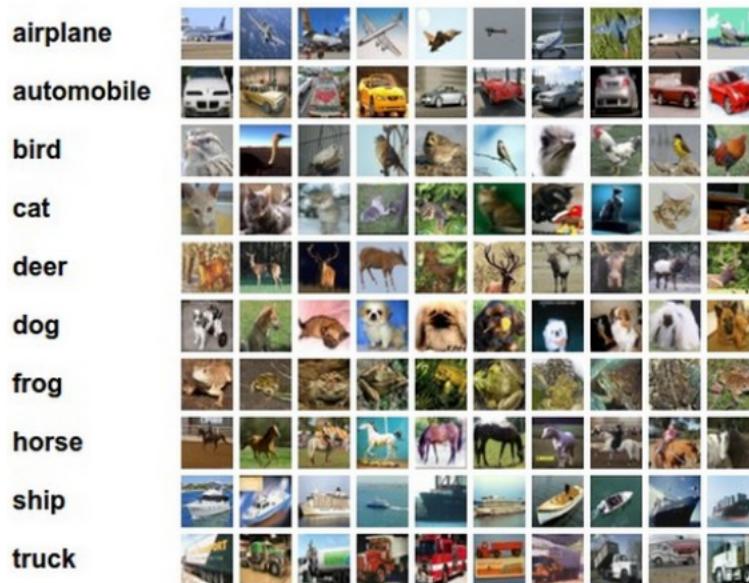


Figura 3.3: CIFAR-10 dataset

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:**

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p| \quad (3.1)$$

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

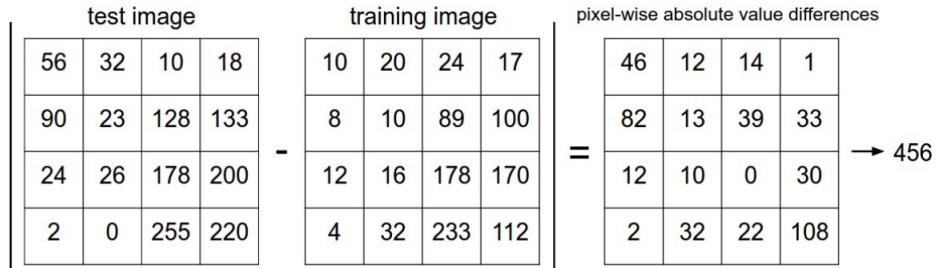


Figura 3.4: L1 distance

2. **L2 distance:**

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2} \quad (3.2)$$

è la distanza euclidea classica.

3. **L_k distance:**

$$d_k(I_1, I_2) = \left(\sum_p (|I_1 - I_2|)^k \right)^{\frac{1}{k}} \quad (3.3)$$

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 Neighbor Classifier* precedente. Intuitivamente un alto valore di k ha un effetto "levigante" sui confini decisionali e rende il classificatore più resistente ai valori anomali.

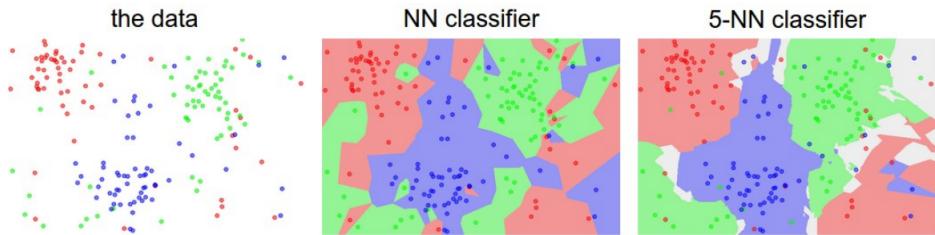


Figura 3.5: confronto NN classifier e 5-NN classifier

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.

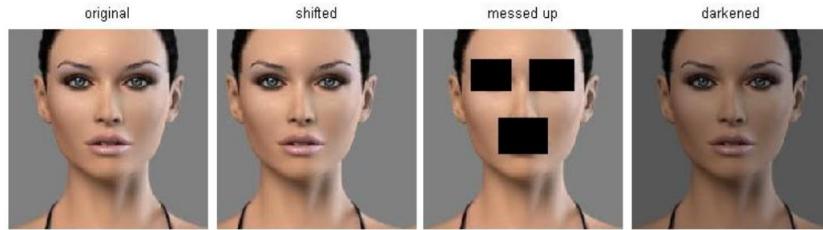


Figura 3.6: Distanza tra immagini, le 3 immagini a sinistra hanno tutte la stessa distanza dall'originale

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

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 incorrette. Più formalmente:

$$\mathbf{F} : \mathbb{R}^d \rightarrow \mathbb{R}^L \quad (3.4)$$

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:

$$\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b \quad (3.5)$$

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.

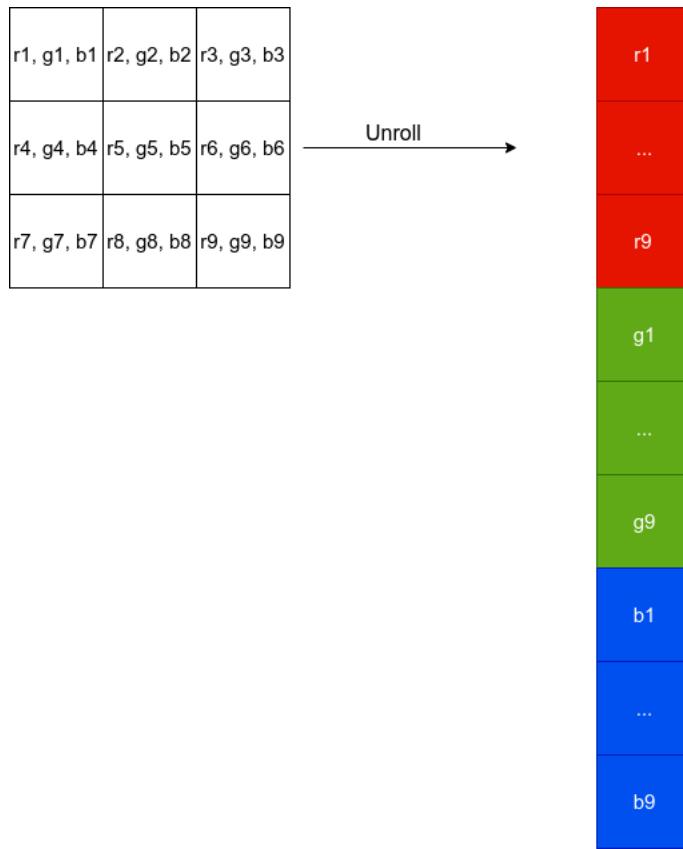


Figura 3.7: Unroll immagine

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

\mathbf{W}

Figura 3.8: Linear Classifier

Il **classificatore lineare** assegna ad un'immagine di input la classe corri-

spondente al più alto punteggio:

$$\hat{y}_j = \arg \max_{i=1, \dots, L} [s_j]_i. \quad (3.6)$$

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 Neighbor 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**).

$$\mathcal{L}(\mathbf{x}, y_i | W, b) \quad (3.7)$$

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:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \quad (3.8)$$

allora la (9) diventa:

$$\mathbf{F}(\mathbf{x} | W, b) = W\mathbf{x} + b = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad (3.9)$$

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

$$\begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & b_1 \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} & b_n \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad (3.10)$$

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:

$$\mathbf{F}(\mathbf{x} | W) = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & w_{1,n+1} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & w_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} & w_{m,n+1} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \\ 1 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix} \quad (3.11)$$

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

Poiché 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:

$$d(\Sigma, \mathbf{P}) = \frac{|\langle \mathbf{a}, \mathbf{P} \rangle + b|}{\|\mathbf{a}\|} \quad (3.12)$$

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.

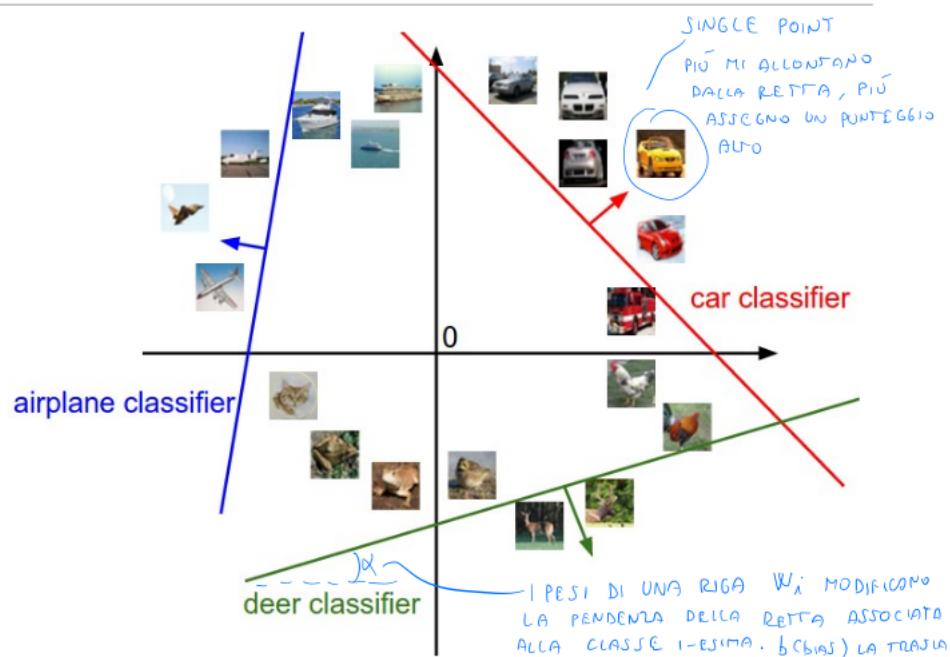


Figura 3.9: Interpretazione geomtrica nello spazio bidimensionale

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.

Capitolo 4

Reti Convoluzionali

Le **reti neurali convoluzionali** (**Convolutional Neural Networks**, CNN) sono di fatto delle reti neurali artificiali. Esattamente come queste ultime infatti, anche le reti neurali convoluzionali sono costituite da neuroni collegati fra loro tramite dei rami pesati (weight); i parametri allenabili delle reti sono sempre quindi i **weight** ed i **bias**. Tutto quanto detto in precedenza sull’allenamento di una rete neurale, cioè forward/backward propagation e aggiornamento dei pesi, vale anche in questo contesto; inoltre un’intera rete neurale convoluzionale utilizza sempre una singola funzione di costo differenziabile. Tuttavia le reti neurali convoluzionali fanno la specifica assunzione che il loro input abbia una precisa struttura di dati, come ad esempio un’immagine nel nostro caso, e ciò permette ad esse di assumere delle specifiche proprietà nella loro architettura al fine di elaborare al meglio tali dati. Ad esempio di poter effettuare delle forward propagation più efficienti in modo da ridurre l’ammontare di parametri della rete. Il nome Convolutional Neural Networks deriva dal fatto che tali reti utilizzano un’operazione matematica lineare chiamata **convoluzione**. Gli strati composti da operazioni di convoluzione prendono il nome di **Convolutional Layers**, ma non sono gli unici strati che compongono una CNN: la tipica architettura prevede infatti l’alternarsi di **Convolutional Layers**, **Pooling Layers** e **Fully Connected Layers**.

4.1 Inadeguatezza della struttura fully-connected

Come visto nel capitolo precedente, le reti neurali tradizionali ricevono in input un singolo vettore, e lo trasformano attraverso una serie di strati nascosti, dove ogni neurone è connesso ad ogni singolo neurone sia dello strato precedente che di quello successivo (ovvero "*fully-connected*") e funziona quindi in maniera completamente indipendente, dal momento che non vi è alcuna condivisione delle connessioni con i nodi circostanti. Nel caso l’input sia costituito da immagini di dimensioni ridotte, ad esempio $32 \times 32 \times 3$ (32 altezza, 32 larghezza, 3 canali colore), un singolo neurone connesso in questa maniera comporterebbe

un numero totale di $32 \times 32 \times 3 = 3072$ pesi, una quantità abbastanza grande ma ancora trattabile. Le cose si complicano però quando le dimensioni si fanno importanti: salire ad appena 256 pixel per lato comporterebbe un carico di $256 \times 256 \times 3 = 196'608$ pesi per singolo neurone, ovvero quasi 2 milioni di parametri per una semplice rete con un singolo strato nascosto da dieci neuroni. L'architettura fully-connected risulta perciò troppo esosa in questo contesto, comportando una quantità enorme di parametri che condurrebbe velocemente a casi di sovradattamento. Inoltre una rete FC è del tutto generica e non ha nessun tipo di accorgimento per l'elaborazione di immagini che presentano una struttura ben determinata. Le Convolutional Neural Networks prendono invece vantaggio dall'assunzione che gli input hanno proprio una struttura di questo tipo, e questo permette loro la costruzione di un'architettura su misura attraverso la formalizzazione di tre fondamentali proprietà: **l'interazione sparsa (sparse interaction)**, **l'invarianza rispetto a traslazioni (invariant to translation)**, e la **condivisione dei parametri (weight sharing)**. Il risultato è una rete più efficace e allo stesso tempo parsimoniosa in termini di parametri.

4.2 Operazione di convoluzione

In problemi discreti l'operazione di convoluzione non è altro che la somma degli elementi del prodotto di Hadamard fra un set di parametri (che prende il nome di filtro o kernel) e una porzione dell'input di pari dimensioni. L'operazione di convoluzione viene quindi ripetuta spostando il filtro lungo tutta la supercie dell'input, sia in altezza che in larghezza. Questo produce quella che viene chiamata mappa di attivazioni (o **features map**), la quale costituisce di fatto il primo strato nascosto della rete.

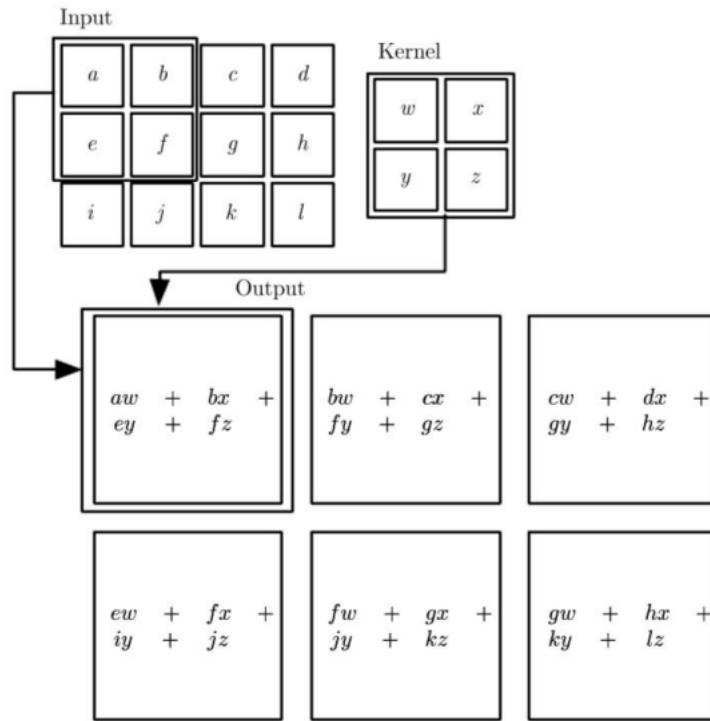


Figura 4.1: Operazione di convoluzione nel caso bidimensionale: un filtro di dimensione 2×2 viene moltiplicato elemento per elemento per una porzione dell'input di uguali dimensioni. L'output dell'operazione è costituito dalla somma di tali prodotti. L'operazione di convoluzione viene infine ripetuta spostando il filtro lungo le due dimensioni dell'input.

Nel caso in cui gli input siano rappresentati da volumi tridimensionali la procedura rimane la stessa, ma è importante notare che il filtro, pur mantendo una ridotta estensione spaziale (larghezza e altezza), viene esteso in profondità in misura uguale all'input.

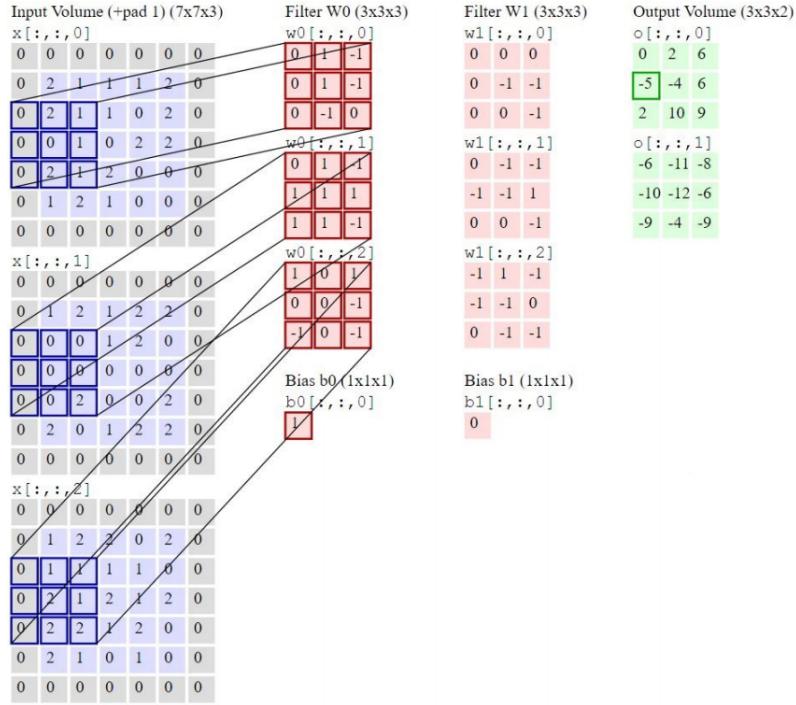


Figura 4.2: Operazione di convoluzione di due differenti filtri (W0 e W1) di dimensione $3 \times 3 \times 3$ su un volume di input $7 \times 7 \times 3$.

È importante notare che la singola operazione di convoluzione produce sempre uno scalare, indipendentemente da quali siano le dimensioni del volume di input, sia esso bidimensionale o tridimensionale. Di conseguenza, una volta che il filtro viene fatto convolare lungo tutta la supercie dell'input, si ottiene sempre una mappa di attivazioni bidimensionale.

4.3 Strati di una CNN

4.3.1 Convolutional Layer

Come si può intuire questo è il principale tipo di layer: l'utilizzo di uno o più di questi layer in una rete neurale convoluzionale è indispensabile. I **parametri di un convolutional layer** in pratica riguardano un insieme di filtri allenabili. Ciascun **filtro** è spazialmente ridotto, lungo le dimensioni di larghezza ed altezza, ma si estende per l'intera profondità del volume di input a cui viene applicato. Durante la forward propagation si trasla, o più precisamente si convolve, ciascun filtro lungo la larghezza e l'altezza del volume di input, producendo una **activation map** (o **feature map**) bidimensionale per quel filtro. Intuitivamente, la rete avrà come obiettivo quello di apprendere (tramite **backpropagation**) i parametri dei filtri.

kpropagation e discesa del gradiente) dei filtri che si attivano in presenza di un qualche specifico tipo di feature in una determinata regione spaziale dell'input. L'accodamento di tutte queste activation map, per tutti i filtri, lungo la dimensione della profondità forma il volume di output di un layer convoluzionale. Ciascun elemento di questo volume può essere interpretato come l'output di un neurone (con la sua funzione di attivazione) che osserva solo un piccola regione dell'input e che condivide i suoi parametri con gli altri neuroni nella stessa activation map, dato che questi valori provengono tutti dall'applicazione dello stesso filtro. In altre parole, ragionando dalla prospettiva opposta, la mappa di attivazioni è formata da neuroni connessi localmente allo strato di input attraverso i parametri del filtro che li ha generati. Attualmente le cose potrebbero non apparire ancora del tutto chiare. Vediamo dunque più dettagliatamente i concetti appena espressi, suddividendoli per meglio comprendere l'intero processo.

Proprietà di connettività locale Era già stato menzionato in precedenza il fatto che, avendo a disposizione degli input di considerevoli dimensioni come appunto delle immagini, connettere ciascun neurone di un layer con tutti i neuroni del layer precedente (o del volume di input) nella pratica non è conveniente. Infatti qui ciascun neurone è connesso solo ad una piccola regione locale del volume di input. L'estensione spaziale (sempre larghezza e altezza) di questa regione è un parametro del layer convoluzionale e viene detto **receptive field** del neurone. È bene ricordare quanto già detto in precedenza: l'estensione lungo l'asse della profondità della regione considerata è sempre uguale alla profondità del volume di input. Ciò significa che rispetto alla profondità non si esclude nulla dell'input per ciascuna regione locale, limitata invece in larghezza ed altezza.

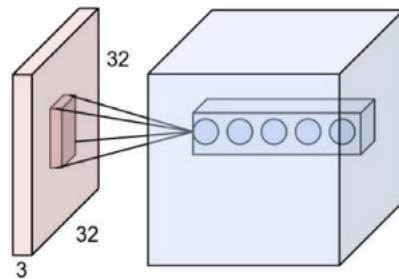


Figura 4.3: Layer convoluzionale applicato ad un'immagine di CIFAR-10, si noti come un singolo neurone non sia connesso a tutto l'input ma solo ad una parte di dimensione uguale al filtro applicato

Un semplice esempio può chiarire questi concetti: prendiamo la solita immagine di CIFAR-10, quindi un volume $[32 \times 32 \times 3]$; se il receptive field è di dimensioni 5×5 , allora ciascun neurone del layer convoluzionale avrà dei weight

associati ad una regione locale $5 \times 5 \times 3$ del volume di input, per un totale di $5 * 5 * 3 = 75$ pesi, numero decisamente inferiore rispetto a quanto visto nella sezione precedente (3072 pesi) per un neurone in un'architettura fully connected.

Output dello strato Fin qui si è parlato della connettività di ciascun neurone di un layer convoluzionale rispetto al volume di input. In questa sottosezione si parlerà invece della quantità di neuroni presenti nel volume di output di un layer convoluzionale e di come essi sono organizzati. In questo caso vi sono 3 iperparametri:

- **Profondità (depth):** corrisponde al numero di filtri N_F che compongono lo strato, ognuno in cerca di caratteristiche differenti nel volume di input. Il set di neuroni (appartenenti a filtri diversi) connessi alla stessa regione di input prende invece il nome di colonna profondità (depth column).
- **Stride:** specifica il numero di pixel di cui si vuole traslare il filtro ad ogni spostamento. Poiché come detto eseguendo la convoluzione tra filtro e parte dell'input otteniamo un singolo scalare, questo parametro permette di specificare in che maniera allocare le depth column in tutto lo spazio, cioè larghezza e altezza, dell'input. Quando lo stride è pari ad uno significa che stiamo muovendo il filtro un pixel alla volta, e di conseguenza viene scannerizzata ogni possibile posizione dell'input. Valori più alti muovono il filtro con salti maggiori, e pertanto viene generato un output di dimensioni minori.
- **Zero padding:** a volte può risultare conveniente aggiungere un bordo di zeri al volume di input, in modo così da controllare le dimensioni dell'output ed evitare incongruenze durante le operazioni. Lo spessore di questo bordo è determinato dall'iperparametro di zero-padding, ed è spesso utilizzato per far combaciare la dimensione dell'input con quella dell'output.

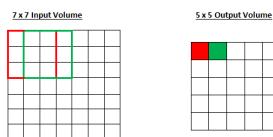


Figura 4.4: Stride uguale a 1

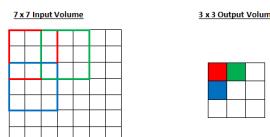


Figura 4.5: Stride uguale a 2

0	0	0	0	0	0
0	35	19	25	6	0
0	13	22	16	53	0
0	4	3	7	10	0
0	9	8	1	3	0
0	0	0	0	0	0

Figura 4.6: Padding uguale a 1

Larghezza e altezza del volume di output sono determinati in funzione del volume di input, dimensione del campo recettivo (dimensione del filtro), dello stride, quantità di zero-padding. La profondità invece dipende dal numero di filtri N_F applicati. Siano F la dimensione di un lato di un filtro (i filtri son quadrati), P la quantità di padding applicata, S il valore di stride allora, dato in input un volume di dimensione $[W_1 \times H_1 \times D_1]$ lo strato convoluzionale produrrà un output di dimensione $[W_2 \times H_2 \times D_2]$ dove:

$$\begin{cases} W_2 = \frac{(W_1 - F + 2P)}{S} + 1 \\ H_2 = \frac{(H_1 - F + 2P)}{S} + 1 \\ D_2 = N_F \end{cases} \quad (4.1)$$

Chiaramente, i valori di stride e zero-padding devono essere scelti in modo tale che W_2 e H_2 siano valori interi. Ad esempio, l'architettura AlexNet di Krizhevsky et al. (2012) accetta in input immagini di dimensione $[227 \times 227 \times 3]$ e utilizza nel suo primo convolutional layer 96 filtri di dimensione 11, stride pari a 4 e nessun zero-padding. Dal momento che $(227 - 11)/4 + 1 = 55$ il volume finale dell'output del primo strato avrà dimensione $[55 \times 55 \times 96]$. Ognuno dei $55 \times 55 \times 96$ neuroni di questo volume è connesso ad una regione di dimensione $[11 \times 11 \times 3]$ del volume di input, e tutti i 96 neuroni appartenenti alla stessa colonna profondità sono connessi alla stessa regione $[11 \times 11 \times 3]$, ma ovviamente attraverso set diversi di pesi. **Osservazione:** è importante che durante la convoluzione i filtri coprano tutto l'input, non è una buona idea quindi adottare un valore di stride superiore alla larghezza dei filtri.

Condivisione dei parametri e invarianza alla traslazione Nell'esempio sopra riportato, ognuno dei $55 \times 55 \times 96 = 290400$ neuroni del primo convolutional layer possiede $11 \times 11 \times 3 = 363$ pesi, più uno relativo alla distorsione. In un'architettura come quella delle reti neurali classiche che non prevede la condivisione dei pesi questo comporterebbe un numero totale di parametri pari a $290400 \times 363 = 105705600$, solamente per il primo strato. Tale quantità, chiaramente intrattabile nella realtà, viene drasticamente ridotta dalla proprietà di condivisione dei parametri (**weight sharing**) delle CNN, la quale si fonda su una ragionevole assunzione: se la rilevazione di una caratteristica in una determinata posizione spaziale risulta utile, lo sarà anche in differenti posizioni spaziali. Si assume cioè una struttura dell'immagine **invariante rispetto a traslazioni**.

Questo è il motivo per cui facciamo scorrere lo stesso filtro su tutto il volume di input. In pratica, chiamando una singola “*sezione*” bidimensionale lungo l’asse di profondità del volume di output con depth slice (o mappa di attivazione, ad esempio un volume $55 \times 55 \times 96$ ha 96 depth slice, ciascuna ovviamente di dimensioni 55×55), si farà in modo che tutti i neuroni di ciascuna **depth slice** (attenzione a non confondere con *depth column*) utilizzino gli stessi weight e bias. Tornando all’esempio, la proprietà di weight sharing comporta una diminuzione dei parametri per il primo strato fino a $(11 \times 11 \times 3) \times 96 + 96 = 34944$, rispetto ai precedenti 105 milioni, ovvero circa tremila volte meno.

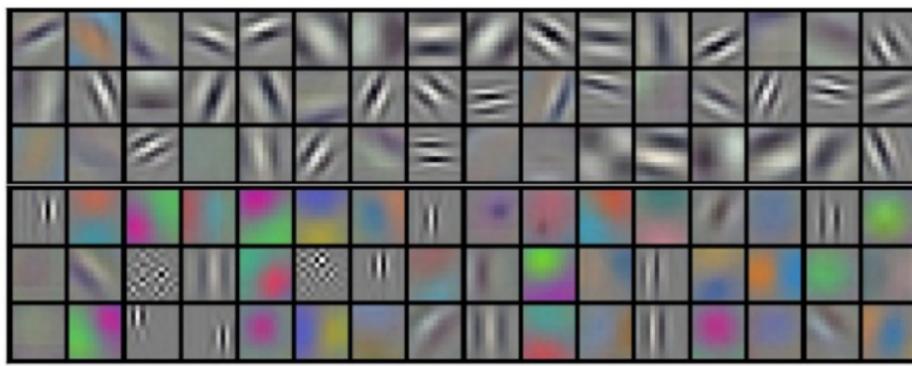


Figura 4.7: Set di 96 filtri $11 \times 11 \times 3$ appresi dall’architettura di Krizhevsky et al. (2012) in un problema di classificazione di immagini. L’assunzione di weight sharing è ragionevole dal momento che individuare una linea o uno spigolo è importante in qualsiasi posizione dell’immagine, e di conseguenza non c’è la necessità di imparare ad localizzare la stessa caratteristica in tutte le possibili zone.

E’ importante notare che in determinati contesti l’assunzione di weight sharing perde di senso. Un caso particolare è quando le immagini in input presentano una struttura specifica e centrata, e ci si aspetta di apprendere caratteristiche differenti in una determinata area dell’immagine piuttosto che in un’altra. Ad esempio, in un problema di riconoscimento facciale dove i volti sono stati ritagliati e centrati, risulterà più efficace apprendere nella parte superiore dell’immagine caratteristiche relative ad occhi o capelli, mentre nella parte inferiore quelle specifiche di bocca e naso.

Riassumendo, un layer convoluzionale:

- accetta in ingresso un volume $[W_1 \times H_1 \times D_1]$;
- richiede il settaggio dei seguenti parametri:
 - numero di kernel N_F (il parametro depth);
 - il lato dell’area del receptive field F;

- lo stride S;
- l'ammontare dello zero-padding P;
- produce un volume di output di dimensioni $[W_2 \times H_2 \times D_2]$ dove:
 - $W_2 = \frac{(W_1 - F + 2P)}{S} + 1$
 - $H_2 = \frac{(H_1 - F + 2P)}{S} + 1$
 - $D_2 = N_F$
- con la tecnica della condivisione dei parametri, si hanno $N_F \times N_F \times D_1$ pesi per ogni kernel, per un totale di $N_F \times N_F \times D_1$ pesi e N_F bias;
- nel volume di output l'i-esimo depth slice (di taglia $W_2 \times H_2$) è il risultato dell'operazione di convoluzione dell'i-esimo kernel sul volume di input con uno stride S e con la successiva aggiunta dell'i-esimo bias.

Campo recettivo Un concetto base nelle CNN è il **campo recettivo (receptive field)**. A causa della connettività sparsa, in una CNN l'output dipende solo dalla regione dell'input, a differenza delle reti FC dove il valore di ogni output dipende da tutto l'input.

Questa regione dell'input è il campo recettivo per quell'output. Più si va in profondità, più largo è il campo recettivo. Di solito il campo recettivo si riferisce all'unità di output finale della rete in relazione all'input della rete, ma la stessa definizione vale per volumi intermedi.

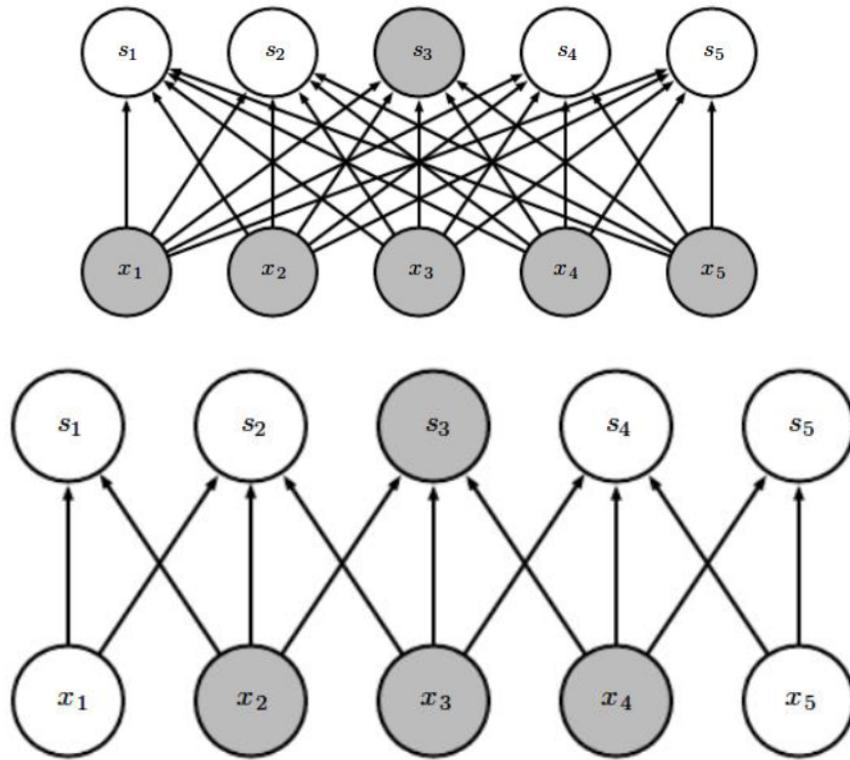


Figura 4.8: Campi recettivi di FCN (in alto) e CNN (in basso)

4.3.2 Pooling layer

Nell'architettura di una CNN è pratica comune inserire fra due o più convolutional layers uno strato di **Pooling**, la cui funzione è quella di ridurre progressivamente la dimensione spaziale degli input (larghezza e altezza), in modo da diminuire numero di parametri e carico computazionale, e di conseguenza controllare anche il sovraccarico. Per il ridimensionamento si utilizza una semplice funzione, come ad esempio un'operazione di **max (max-pooling)** oppure di **media (average-pooling)** e pertanto non comporta la presenza di pesi allenabili. L'assenza di pesi allenabili comporta anche che questo layer svolga le sue mansioni solo durante la forward propagation, nelle fasi di backward propagation retropropaga gli errori e basta senza calcolare alcuna derivata. I pooling layer hanno alcuni iperparametri settabili:

- il **lato F** dell'estensione spaziale della selezione quadrata che verrà di volta in volta considerata sull'input in ogni suo depth slice;
- il parametro di **stride S**;

Si può notare una certa somiglianza con i parametri di un layer convoluzionale: questo perchè anche qui si ha una sorta di receptive field che viene spostato di volta in volta, con un passo specificato dal parametro di stride, su ciascun depth slice del volume di input. In realtà oltre a questo, la situazione in questo caso è completamente diversa, dato che qui non vi è alcuna operazione di convoluzione. Come per il layer convoluzionale, il volume di output $[W_2 \times H_2 \times D_2]$ è funzione del volume di input $[W_1 \times H_1 \times D_1]$ e dei due iperparametri secondo formule analoghe a quelle della sezione precedente:

$$\begin{cases} W_2 = \frac{(W_1 - F)}{S} + 1 \\ H_2 = \frac{(H_1 - F)}{S} + 1 \end{cases} \quad (4.2)$$

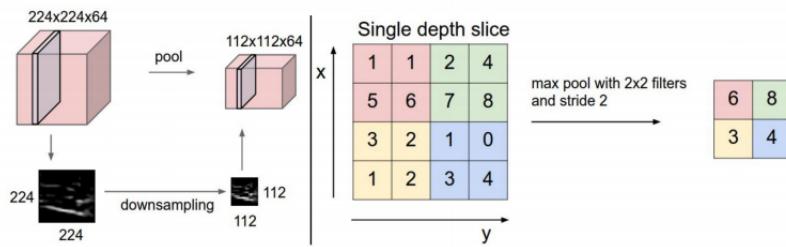


Figura 4.9:

4.3.3 Fully Connected layer

Questo tipo di layer è esattamente uguale ad uno qualsiasi dei layer di una classica rete neurale artificiale con architettura fully connected: semplicemente in un **layer Fully Connected (FC)** ciascun neurone è connesso a tutti i neuroni del layer precedente, nello specifico alle loro attivazioni. L'architettura fully-connected implica il rilassamento dell'assunzione di weight sharing. Valgono quindi la maggior parte delle considerazioni fatte nel capitolo 2, ad eccezione ovviamente di quelle che non hanno senso applicate ad un singolo strato o un sottoinsieme degli strati della rete (es. l'algoritmo di discesa del gradiente è unico per tutta la rete, non ha senso applicarne uno diverso per uno strato). Solitamente si utilizza più di un layer FC in serie e l'ultimo di questi avrà un numero di neuroni K pari al numero di classi presenti nel dataset (nel caso della classificazione) o in generale il numero di neuroni è determinato dall'output della rete desiderato. La funzione principale dei FC layer nell'ambito delle reti neurali convoluzionali è quello di effettuare una sorta di raggruppamento delle informazioni ottenute fino a quel momento, esprimendole con un singolo numero (l'attivazione di uno dei suoi neuroni), il quale servirà nei successivi calcoli per la classificazione finale.

4.4 Architettura generale di una CNN

Finora abbiamo visto i singoli strati che possono essere impiegati nell'architettura di una Convolutional Neural Network. Come per le reti neurali tradizionali, anche qui non esiste una linea guida per la scelta degli iperparametri, ne tanto meno per la sequenza da adottare nella scelta degli strati. Ci sono però alcune considerazioni ragionevoli che si possono fare per cercare quantomeno di muoversi lungo la giusta direzione. La dimensione dei filtri dovrebbe, almeno in linea teorica, essere motivata dalla correlazione spaziale dell'input che un particolare strato riceve, mentre il numero di filtri utilizzati (quindi di feature maps generate) si riflette sulla capacità della rete di cogliere rappresentazioni gerarchiche: questo significa che gli strati finali necessitano di un numero di filtri maggiore di quello destinato agli strati iniziali, altrimenti si limitano le possibilità di combinare features di basso livello per rappresentare features di alto livello, che sono di fatto quelle più vicine alla classificazione finale. La rappresentazione gerarchica sottintende che generalmente non è un sufficiente un singolo strato convoluzionale, in quanto features estratte nello stesso strato sono gerarchicamente di pari importanza. Inoltre anche per le CNN valgono le tecniche per affrontare l'overfitting quali dropout, batch normalization e altre trattate in precedenza con gli opportuni accorgimenti del caso. La maggior parte delle architetture CNN seguono il seguente schema di composizione dei layer:

INPUT -> [[CONV -> RELU]*N -> POOL?] *M -> [FC -> RELU]*K -> FC

dove:

- * indica la ripetizione
- POOL? indica l'opzionalità del pooling layer
- $N > 0$ e solitamente $N \leq 3$
- $M \geq 0$
- $k \geq 0$ e solitamente $k < 3$

Nelle prossime pagine mostreremo alcune architetture note di reti CNN e varianti per compiti specifici (es. segmentation)

4.5 Esempi architetture CNN

Architetture note di CNN sono:

1. **LeNet-5:** sviluppato nel 1998 per identificare cifre scritte a mano per il riconoscimento dello zip code nel servizio postale (60000 parametri). L'idea era quella di usare una sequenza di 3 layers: convoluzione, pooling, non-linearietà. I pixel in un'immagine sono altamente correlati, la convoluzione può essere usata come modo di estrarre le features spaziali e per avere connessioni sparse, mentre il subsample viene fatto usando la media spaziale delle mappe (average pooling). Le non-linearietà usate in

questa rete sono Tanh o sigmoidi. Infine, l'ultimo layer ha un MLP come classificatore.

2. **AlexNet:** sviluppata nel 2012, è simile a LeNet-5 (60 milioni di parametri, conv.: 3.7 milioni, FC: 58.6 milioni). La dimensione dell'input è di 224x224x3, mentre i layer sono:

- 5 layer convoluzionali
- 3 MLP

Per evitare l'overfitting la rete usa ReLU, Dropout (0.5), weight decay e maxpooling. Il primo layer convoluzionale ha 96 filtri 11x11 con stride=4. L'output è costituito da 2 volumi da 55x55x48.

3. **VGG16:** introdotta nel 2014 come variante della AlexNet (138 milioni di parametri, conv.:11%, FC:89%). La dimensione dell'input è di 224x224x3. L'idea è quella di usare convoluzioni multiple 3x3 in sequenza per ottenere campi recettivi più larghi con meno parametri e più non-linearietà anzichè con filtri più larghi in un layer singolo.
4. **GoogleNet:** rete con alta efficienza computazionale (5 milioni di parametri, 22 layers di moduli Inception). Questa rete è basata su moduli Inception, ovvero una sorta di «rete nella rete» o «moduli locali». Scegliere la giusta dimensione del kernel per le operazioni di convoluzione è difficile, dato che l'immagine potrebbe mostrare features rilevanti a scalature diverse. Le reti troppo dense tendono inoltre a overfittare e a diventare computazionalmente molto costose. La soluzione è quella di sfruttare dimensioni multiple del filtro allo stesso livello e poi fare un merge tramite concatenazione delle mappe di attivazione dell'output (con zero padding).
5. **Inception Module:** per ridurre il carico computazionale della rete, il numero dei canali di input è ridotto aggiungendo un layer convoluzionale 1x1 prima delle convoluzioni 3x3 e 5x5. Il volume dell'output ha dimensione simile, ma il numero di operazioni richieste è ridotto significativamente grazie alla convoluzione 1x1 (che incrementa quindi il numero di non-linearietà). Tale layer convoluzionale è anche chiamato "bottleneck layer".

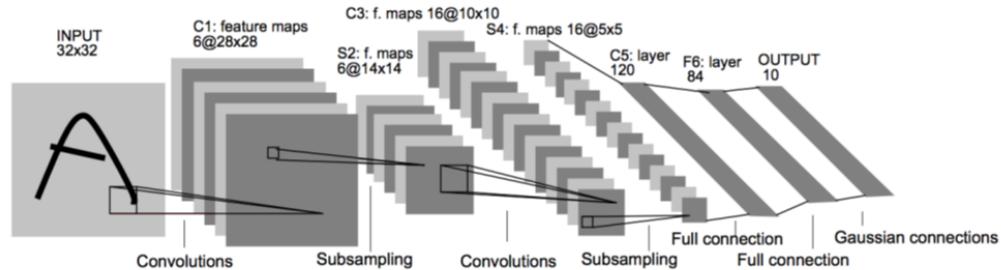


Figura 4.10: LeNet-5

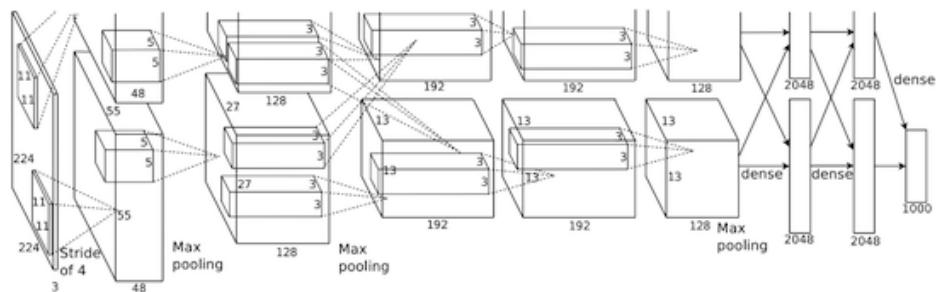


Figura 4.11: AlexNet

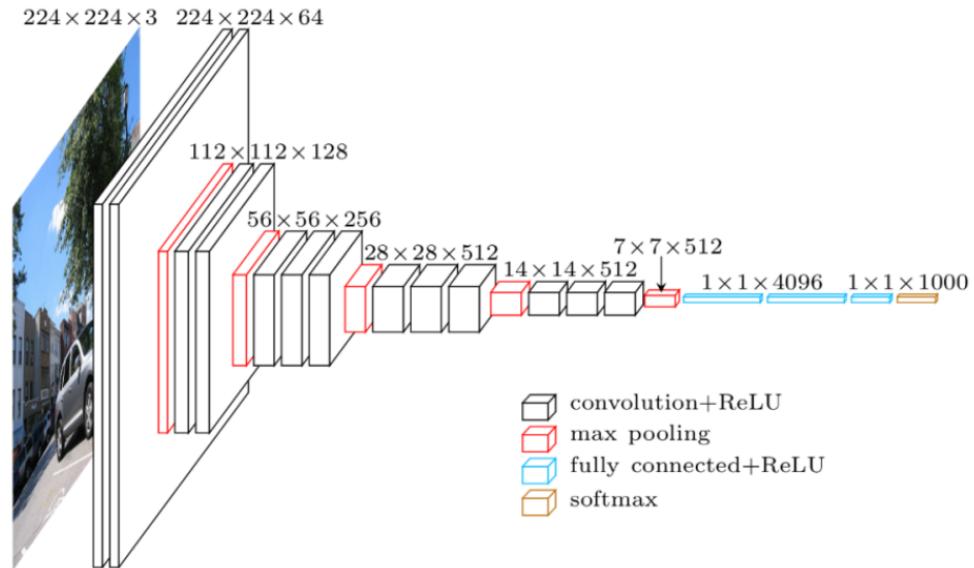


Figura 4.12: VGG16

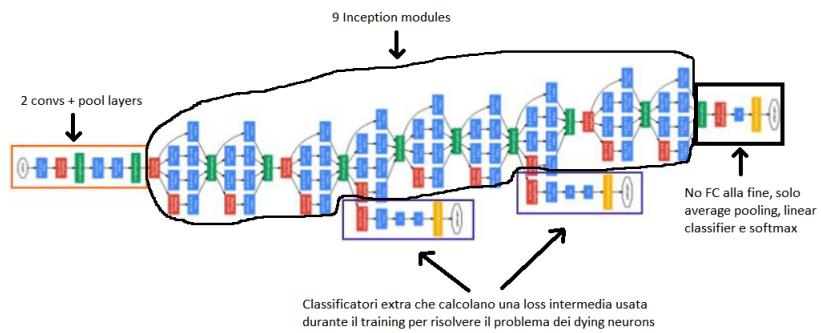


Figura 4.13: GoogleNet

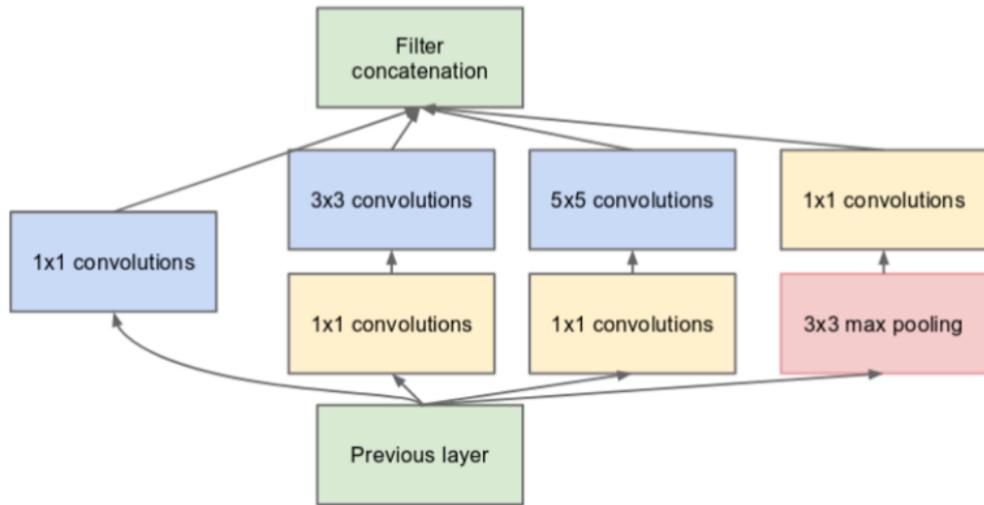


Figura 4.14: Inception module

4.6 Data Augmentation

La **Data Augmentation** è tipicamente effettuata tramite:

- **trasformazioni geometriche**
 - shift/rotazioni/distorsioni
 - shear
 - scaling
 - flip
- **trasformazioni fotometriche**
 - aggiunta di rumore
 - modifica dell'intensità media
 - superimposizione di altre immagini
 - modificazione del contrasto

Test Time Augmentation

Anche se la CNN è allenata usando Data Augmentation, non raggiungerà un'invarianza perfetta rispetto alle trasformazioni considerate.

La **Test Time Augmentation (TTA)** può essere performata a test time per migliorare la precisione delle predizioni. TTA è particolarmente utile per le immagini di test in cui il modello è parecchio insicuro.

Gli step della TTA sono:

1. performare un'augmentation randomica su ogni immagine di test I
2. fare una media delle predizioni per ogni I
3. prendere il vettore delle medie per la definizione della predizione finale

Confusion matrix

L'idea della **confusion matrix** sta nel fatto che l'elemento $C(i, j)$ della matrice corrisponda alla percentuale degli elementi appartenenti alla classe i classificati come elementi della classe j . Quindi la confusion matrix ideale ha tutti 1 sulla diagonale principale.

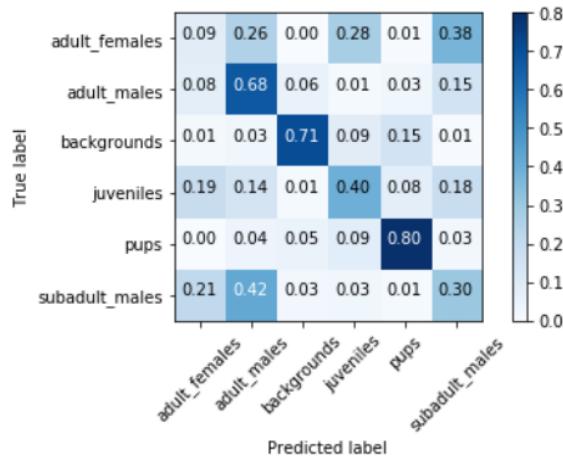


Figura 4.15: Esempio di confusion matrix

Classificazione a due classi

La performance della classificazione nel caso di classificatori binari può anche essere misurata in termini della **ROC (Receiver Operating Characteristic) curve**.

Il classificatore ideale raggiungerebbe:

- FPR = 0%
- TPR = 100%

Quindi, più larga è l'**Area Under Curve (AUC)**, meglio è.

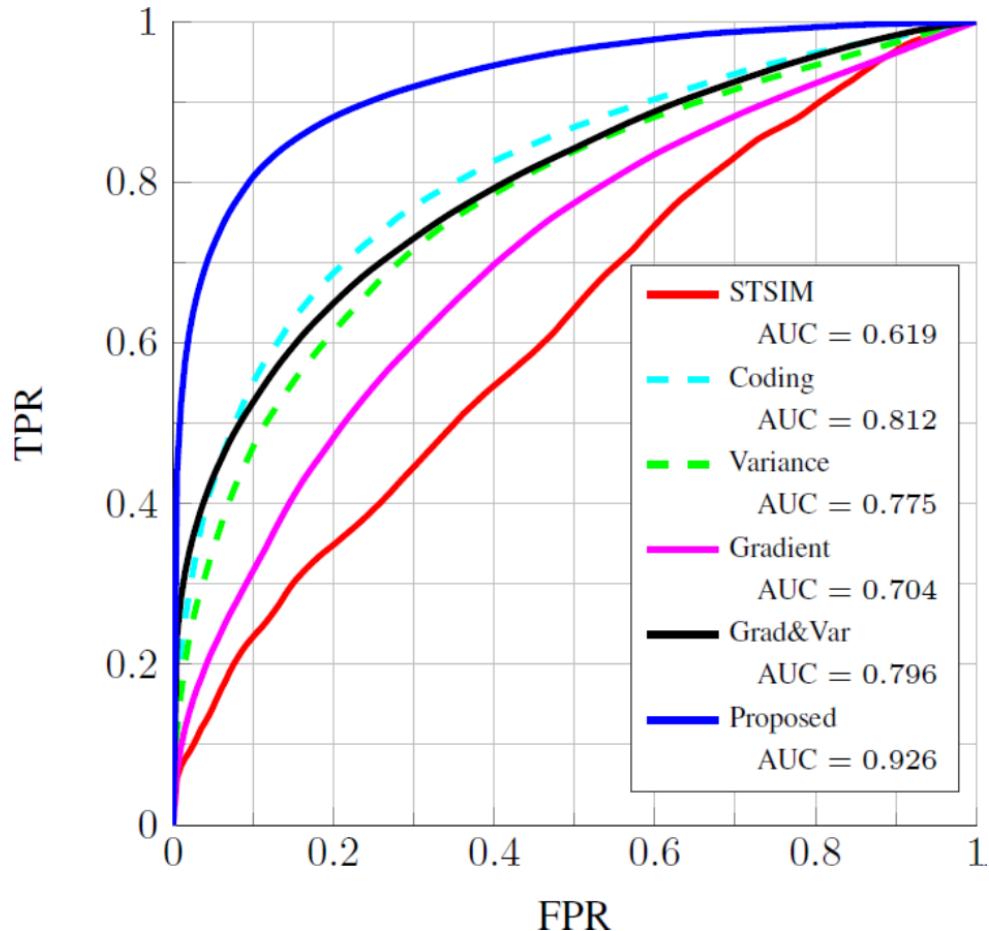


Figura 4.16: ROC (ogni punto rosso è specifico per un parametro)

4.7 Transfer Learning

L'output del fully connected layer ha la stessa dimensione del numero di classi L, e ogni componente fornisce un punteggio di appartenenza a una certa classe per l'immagine in input.

L'input del fully connected layer può essere visto come un descrittore dell'immagine in input, cioè un vettore di features, le quali sono definite per massimizzare la performance di classificazione e sono allenate con backpropagation.

Come sappiamo, più ci muoviamo in profondità, più la risoluzione spaziale è ridotta e il numero di mappe (relativo al campo percettivo) aumenta, quindi cerchiamo dei pattern di alto livello senza preoccuparci troppo della loro posizione esatta nell'immagine.

La parte convoluzionale può quindi essere vista come un estrattore di feature molto generale, mentre i FC layers sono molto personalizzati, dato che sono pensati per risolvere obiettivi di classificazione specifici.

L'idea alla base del **transfer learning** è quindi quella di:

- prendere un modello pre-allenato (es.: VGG)
- rimuovere e modificare i FC layers
- congelare i pesi nella CNN
- allenare l'intera rete sui nuovi dati di training

Ci sono due opzioni di congelamento della CNN:

- **Transfer Learning**: solo i layers della FCN vengono allenati (una buona opzione quando si hanno pochi dati di training e ci si aspetta che la CNN pre-allenata sia compatibile col problema)
- **Fine tuning**: l'intera CNN è riallenata, ma i layer convoluzionali sono inizializzati al modello pre-allenato (una buona opzione quando si hanno abbastanza dati di training o quando non ci si aspetta che la CNN pre-allenata sia compatibile col problema)

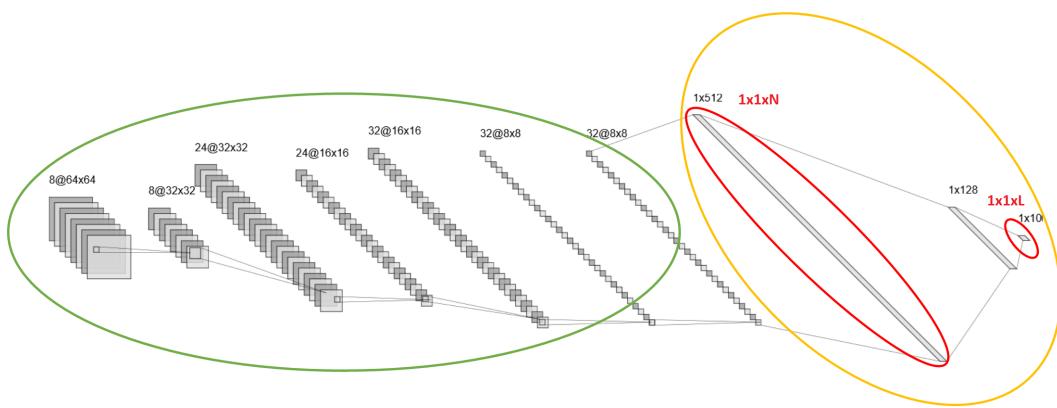


Figura 4.17: Architettura tipica di una CNN (verde: estrattore feature, giallo: classificatore feature)

4.8 Segmentazione di immagini

La **segmentazione semantica** di un'immagine ha come obiettivo quella di assegnare ad ogni pixel un'etichetta proveniente da un insieme fissato.



Figura 4.18: Segmentazione di una camera da letto

In figura 4.9 viene mostrato un esempio di segmentazione, ogni etichetta assegnata ad un pixel viene rappresentata come un colore differente. Più formalmente:

Data un'immagine I , associamo ad ogni pixel identificato dalle sue coordinate (r,c) dove r e c identificano riga e colonna rispettivamente, un'etichetta proveniente da un insieme Λ .

4.8.1 Fully-Convolutional Neural Networks per la segmentazione semantica

Le CNN sono pensate per processare input di dimensione fissata. I layer convoluzionali e di subsampling operano in maniera scorrevole sull'immagine, mentre i FC layers vincolano l'input a una dimensione fissata. Dando in pasto un'immagine di dimensione più larga alla rete, quindi, non si possono calcolare i punteggi delle varie classi, ma si possono comunque estrarre le features.

Ad ogni modo, dato che la FCN è lineare, può essere rappresentata come una convoluzione, i cui pesi sono associati al neurone di output o_i . Si ha che:

$$o_i = (\mathbf{w}_i \odot s)(0,0) + b_i \quad (4.3)$$

dove $\mathbf{w}_i = \{w_{i,j}\}_{i=j:N}$ sono i pesi associati a o_i .

Quindi, dato che la FCN è lineare, può essere rappresentata come una convoluzione di L filtri di dimensione $1 \times 1 \times N$. Ognuno di questi filtri convoluzionali contiene i pesi della FCN per il corrispondente neurone in output. Per ogni classe in output, otteniamo quindi un'immagine (**heatmap**) avente una risoluzione più bassa rispetto a quella dell'immagine in input e probabilità delle classi per il campo recettivo di ogni pixel.

In un'immagine più grande, ogni pixel nella heatmap corrisponde a un campo recettivo nell'immagine in input.

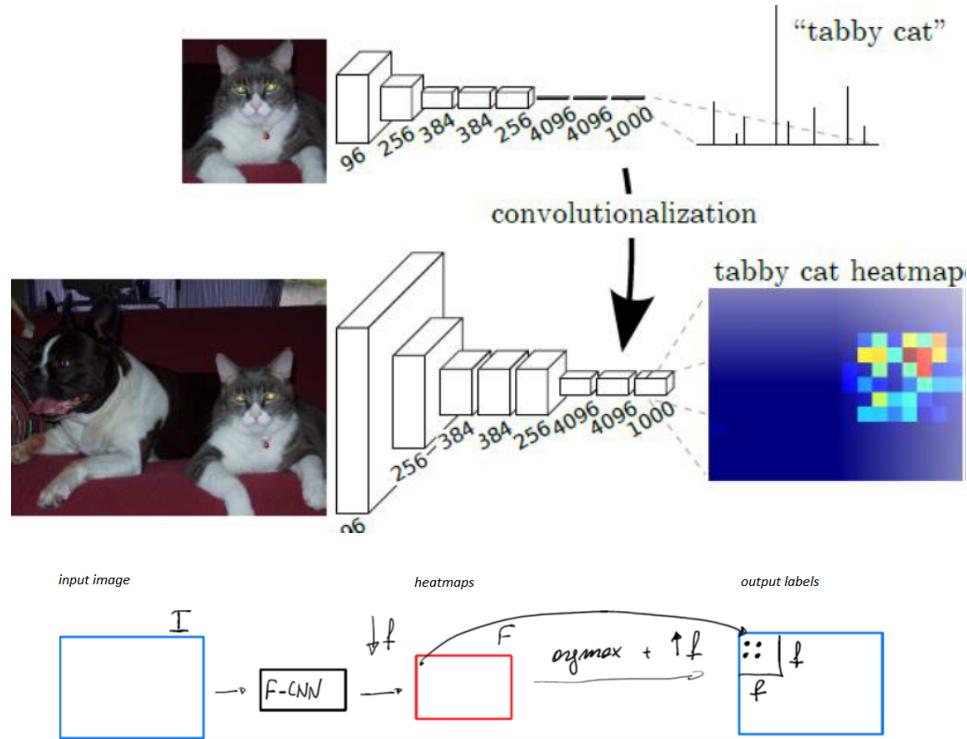


Figura 4.19: Esempio di heatmap (sopra), Direct Heatmap Predictions (sotto)

Le **Fully-Convolutional Neural Networks** sono dunque un mezzo per effettuare predizioni su immagini di dimensione arbitraria. Ci sono varie soluzioni per segmentare le immagini tramite FCNN:

1. **Direct Heatmap Predictions:** possiamo assegnare l'etichetta predetta nella heatmap all'intero campo recettivo, nonostante questa sarebbe una stima molto grossolana
2. **Shift and Stich:** assumiamo che ci sia un ratio f tra la dimensione dell'input e la dimensione della heatmap in output. Compiamo i seguenti step:
 - calcoliamo le heatmap per tutti i possibili shift f^2 dell'input ($0 \leq r, c < f$)
 - mappiamo le predizioni dalle f^2 heatmap all'immagine (ogni pixel della heatmap fornisce la predizione del pixel centrale del campo recettivo)
 - frapponiamo le heatmap per formare un'immagine larga quanto l'input

3. solo convoluzioni: campo recettivo piccolo e metodo molto inefficiente

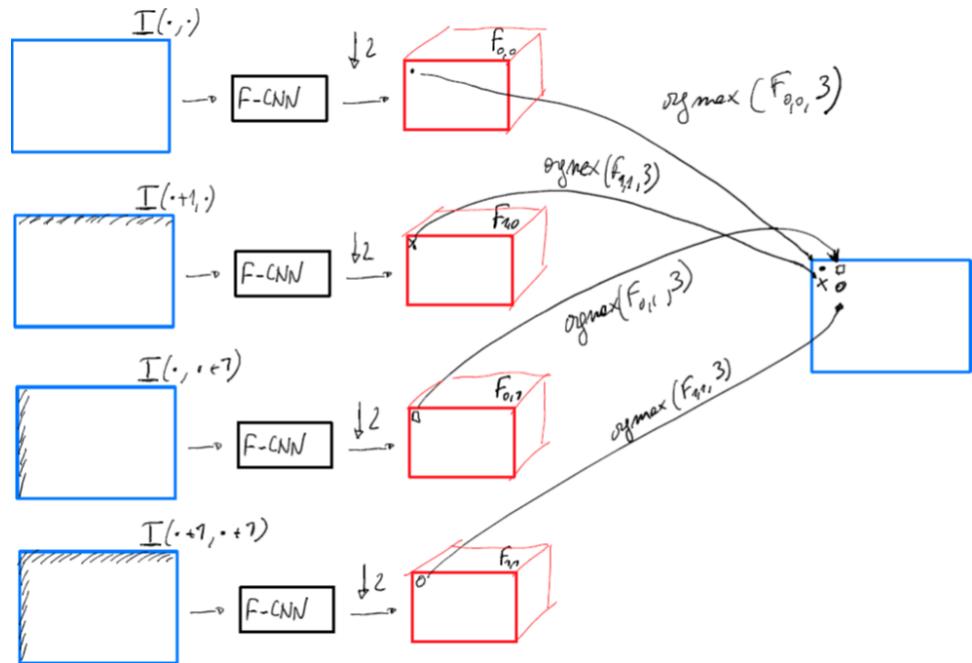


Figura 4.20: Shift and Stich

La segmentazione semantica affronta una tensione tra semantica e locazione, ovvero si pone l'obiettivo di fare predizioni locali rispetto alla struttura globale. Un modo per fare ciò è rappresentato nella figura seguente.

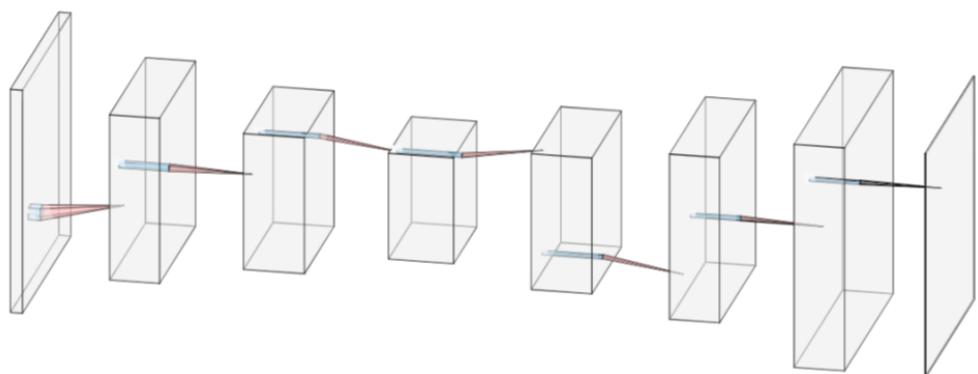


Figura 4.21: Rete con convoluzione e upsampling

Come possiamo notare, la prima parte della rete è una normale CNN (convoluzioni + pooling/downsampling), mentre la seconda metà è pensata per fare un upsampling delle predizioni per coprire ogni pixel dell'immagine (convoluzioni + upsampling).

Per fare l'**upsampling** possiamo usare 2 metodi:

- **Nearest Neighbor**: si riempiono tutti i pixel del layer di output col valore del corrispettivo pixel del layer di input
- **Bed of Nails**: si riempiono tutti i pixel del layer di output con degli 0, eccetto un pixel che avrà il valore del corrispettivo pixel del layer in input

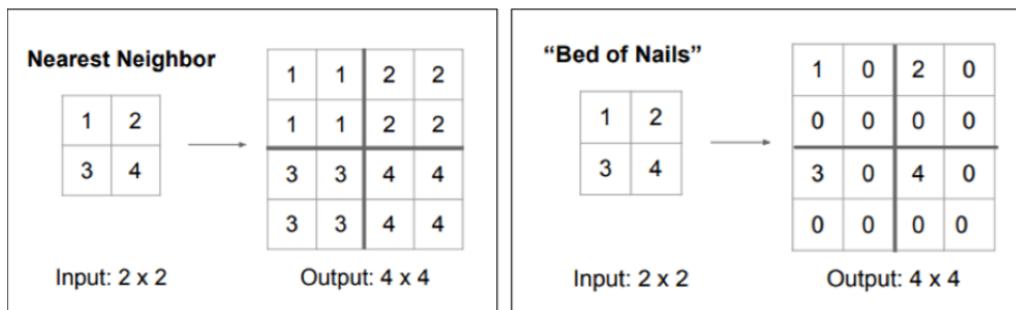


Figura 4.22: UpSampling

Per mantenere anche le informazioni delle posizioni dei pixel, usiamo un meccanismo chiamato **Max Unpooling** (l'inverso del Max Pooling, vedi figura sotto).

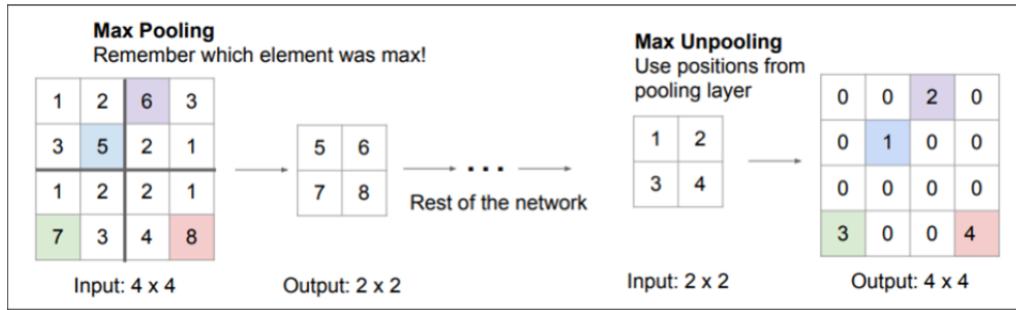


Figura 4.23: Max Unpooling

Inoltre possiamo anche fare una convoluzione trasposta, che è esattamente l'operazione reciproca della convoluzione. L'unico problema sta nei pixel che si sovrappongono, ma la soluzione consiste nel fare una somma pesata tra i pixel in input e la parte di filtro compresa nella regione di overlap.

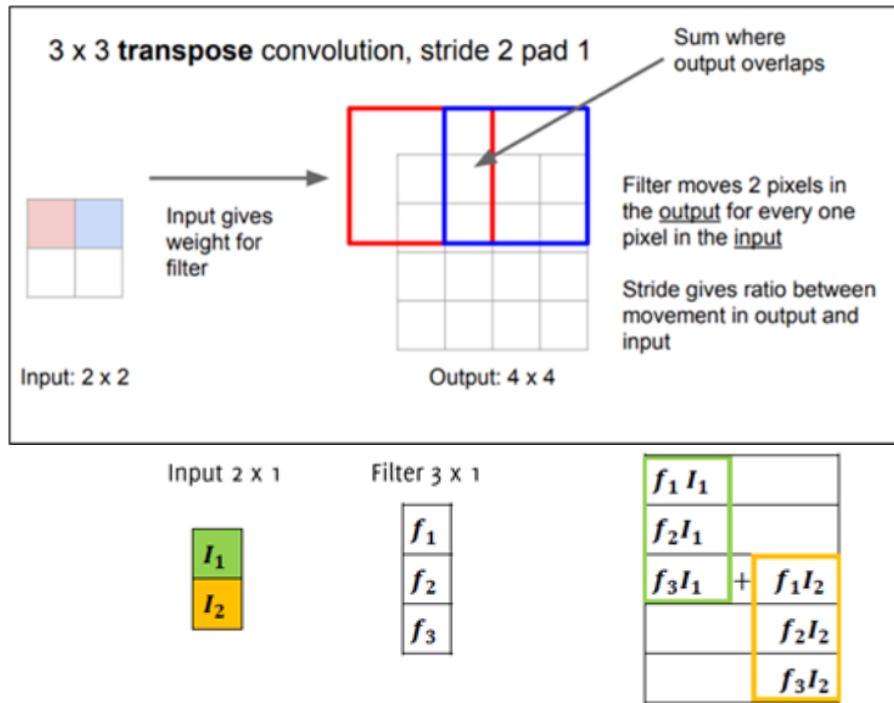


Figura 4.24: Convoluzione trasposta (stride = 2)

I filtri di upsampling possono essere appresi durante il training con un'inizializzazione uguale all'interpolazione bilineare. Le predizioni derivate dall'upsampling sono comunque troppo grossolane. La soluzione a questo problema è quella di usare delle **Skip Connections**. Queste integrano una tradizionale rete di contrazione dove la convoluzione è rimpiazzata dalla convoluzione trasposta.

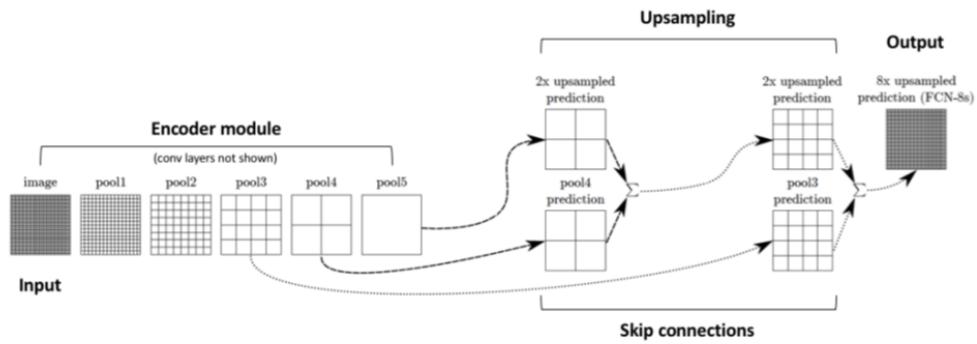


Figura 4.25: Skip connections

Metodi per allenare una FCNN

1. **Patch-Based:** eseguiamo i seguenti step:

- prepariamo un set di training per una rete di classificazione
- ritagliamo delle patch x_i dalle immagini e assegnamo a ogni patch l'etichetta corrispondente al punto centrale della patch
- alleniamo una CNN per la classificazione da zero o facciamo fine-tuning di un modello pre-allenato sulle classi di segmentazione
- spostiamo i FC layers nelle convoluzioni 1x1
- disegniamo la parte di upsampling della rete e ne alleniamo i filtri
- la rete di classificazione è allenata per minimizzare l'errore di classificazione l su un mini-batch

$$\hat{\theta} = \min_{\theta} \sum_{x_j} l(\mathbf{x}_j, \theta) \quad (4.4)$$

- i batch sono poi randomicamente assemblati durante il training

2. **Full-Image:** è possibile allenare direttamente una FCNN che include i layer di upsampling, quindi il learning diventa la seguente minimizzazione

$$\min_{x_j \in I} \sum_{x_j} l(\mathbf{x}_j, \theta) \quad (4.5)$$

dove \mathbf{x}_j sono i pixel in una regione dell'immagine di input e la funzione di errore è valutata sulle etichette corrispondenti nell'annotazione. Tale metodo è chiamato anche "*metodo end-to-end*" ed è più efficiente del Patch-Based.

4.8.2 U-Net

La **U-Net** è formata da una parte contrattiva e una parte espansiva e non ha FC layers (è **simmetrica**). Essa usa un gran numero di feature maps nella parte di downsampling e usa molta data augmentation.

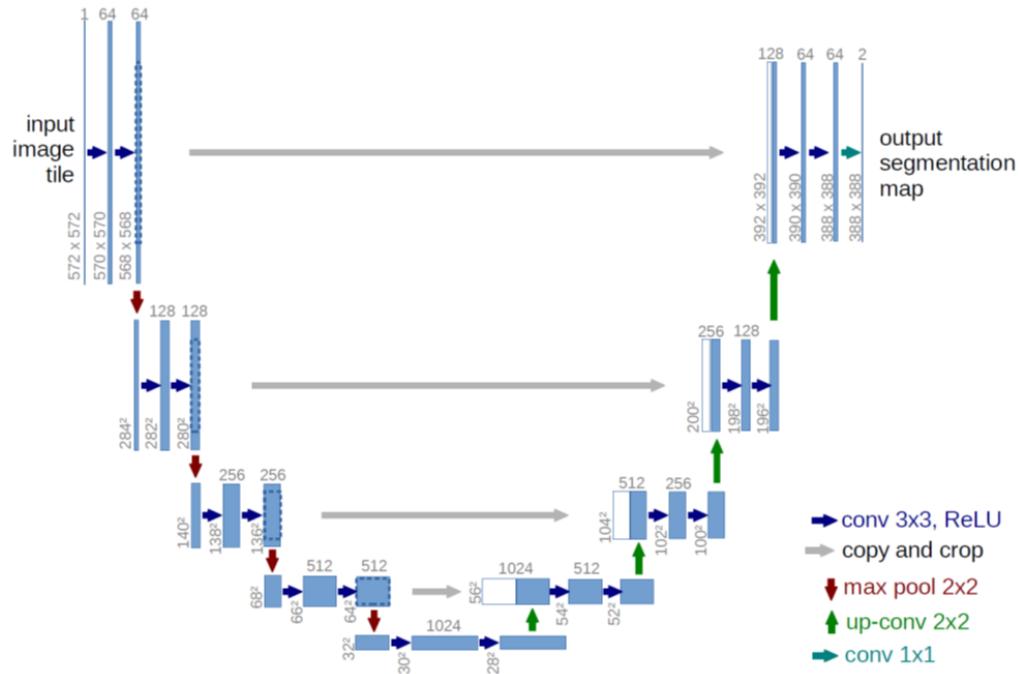


Figura 4.26: U-Net

Nella parte di **contrazione**, la rete ripete blocchi aventi:

- 2 layer convoluzionali 3x3 + ReLU (opzione 'valid', no padding)
- maxpooling 2x2

A ogni downsampling, inoltre, il numero di feature maps è raddoppiato. La particolarità di questa rete è che implementa le skip connections, tramite le quali aggrega tramite concatenazione l'ultimo layer convoluzionale, per ogni blocco nella rete di contrazione, col primo layer convoluzionale del corrispettivo blocco della rete di espansione.

Nella parte di **espansione**, la rete ripete blocchi aventi:

- convoluzione trasposta 2x2, dimezzando il numero di feature maps (ma raddoppiando la risoluzione spaziale)
- concatenazione delle cropped features corrispondenti
- 2 layer convoluzionali 3x3 + ReLU

Il risultato è un'immagine in output più piccola di quella in input.

La rete U-Net usa il metodo di training Full-Image con **una funzione di errore pesata** (**NB.** È la funzione appositamente formulata per la segmentazioni di

immagini biomediche per cui U-Net è stata pensata <https://arxiv.org/pdf/1505.04597.pdf>)

$$\hat{\theta} = \min_{\theta} \sum_{x_j} w(x_j) \log(x_j, \theta) \quad (4.6)$$

dove il peso è dato dalla relazione

$$w(\mathbf{x}) = w_c(\mathbf{x}) + w_0 e^{-\frac{(d_1(\mathbf{x})+d_2(\mathbf{x}))^2}{2\sigma^2}} \quad (4.7)$$

In quest'ultima relazione, w_c è usato per bilanciare le proporzioni delle classi, d_1 è la distanza tra il bordo e la cellula più vicina, mentre d_2 è la distanza tra il bordo e la seconda cellula più vicina (i pesi sono più grandi quando la distanza dalle due celle più vicine al bordo è piccola). Inoltre, il primo termine dell'equazione tiene conto dello sbilanciamento delle classi nel training set, mentre il secondo migliora la performance della classificazione ai bordi di oggetti diversi tra loro.

4.8.3 Global Averaging Pooling

L'idea principale consiste nell'usare, anzichè le convoluzioni tradizionali, uno stack di convoluzioni 1x1 + ReLU, che corrisponde alle reti di MLP usate in maniera scorrevole su tutta l'immagine. Viene quindi introdotto il **Global Averaging Pooling layer**, che calcola la media di ogni feature map al posto del FC layer alla fine della rete e predire usando una semplice softmax. L'importante è che il numero di feature maps sia uguale al numero di classi in output. Tra i vantaggi di questo layer vi è quello di riuscire a classificare immagini di dimensioni diverse.

In pratica è come se avessimo una **rete nella rete (NiN)** composta da:

- MLP convolutional layers + ReLU + dropout
- maxpooling
- GAP layer
- softmax

Delle semplici NiN raggiungono la miglior performance su piccoli dataset grazie al fatto che il GAP riduce efficacemente l'overfitting rispetto al FC. Possiamo quindi affermare che il GAP agisce come un regolarizzatore strutturale.

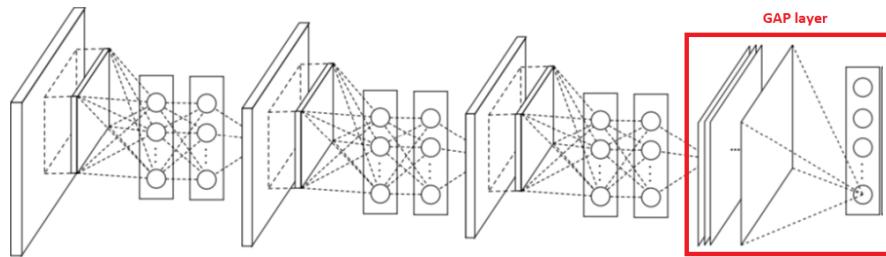


Figura 4.27: Network in Network

Weakly-Supervised Localization La **Weakly-Supervised Localization** effettua una localizzazione su un’immagine senza box di contorno annotati. Il training set è fornito come per la classificazione con coppie <immagine, etichetta> (I, l) dove non è fornita alcun’informazione sulla localizzazione.

CAM I vantaggi del GAP layer vanno oltre al semplice agire come regolarizzatore che previene l’overfitting. Infatti, la rete può conservare una notevole abilità di localizzazione fino al layer finale. Una CNN allenata sulla cATEGorizzazione di un oggetto è quindi capace di localizzare con successo le regioni discriminative per la classificazione contenenti gli oggetti con cui gli umani interagiscono piuttosto che gli umani stessi. Ciò viene fatto tramite una tecnica chiamata **Class Activation Mapping (CAM)**, che permette di identificare quali regioni di un’immagine verranno usate per la discriminazione. Essa richiede solo un FC layer dopo il GAP e una piccola regolarizzazione. Tale layer calcola lo score S_c per ogni classe c come

$$S_c = \sum_k w_k^c F_k \quad (4.8)$$

dove

$$F_k = \sum_{(x,y)} f_k(x, y) \quad (4.9)$$

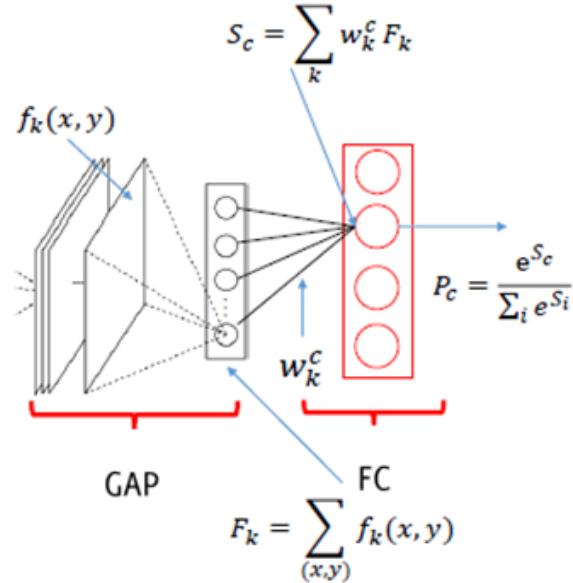


Figura 4.28: CAM

Dopodichè, calcola la probabilità P_c della classe c come

$$P_c = \frac{e^{S_c}}{\sum_i e^{S_i}} \quad (4.10)$$

Ad ogni modo, unendo la (128) e la (129), e sapendo che il CAM è definito come

$$M_c(x, y) = \sum_k w_k^c f_k(x, y) \quad (4.11)$$

otteniamo che

$$S_c = \sum_{(x,y)} M_c(x, y) \quad (4.12)$$

dove $M_c(x, y)$ indica direttamente l'importanza delle attivazioni a (x, y) per la classe c . Inoltre, grazie alla softmax, la profondità dell'ultimo layer convoluzionale può essere diversa dal numero di classi. Potrebbe anche essere necessario un upsampling per poter matchare l'immagine in input (vedi figura sotto). I pesi rappresentano quindi l'importanza di ogni feature map per portare alla predizione finale.

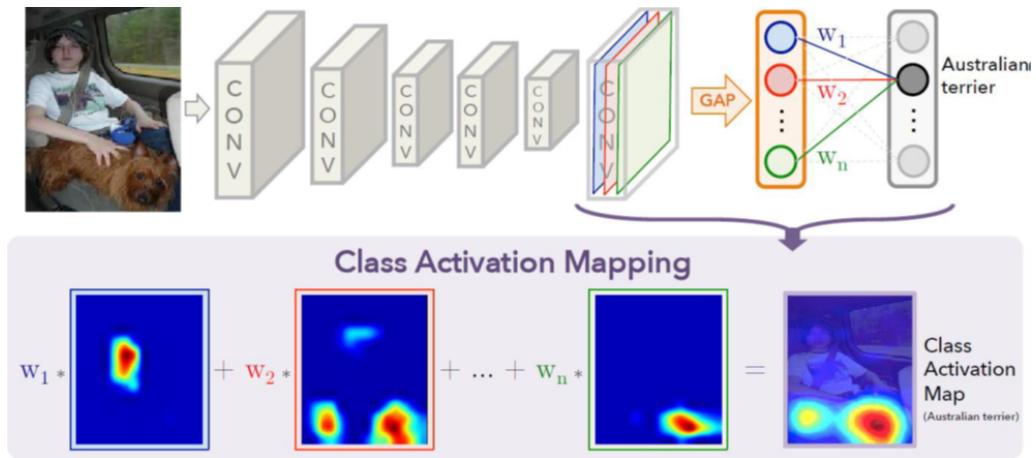


Figura 4.29: Class Activation Mapping

Grad-CAM CAM ha il grande svantaggio di dover modificare la rete per poter predire un'altra classe. Una tecnica che risolve questo problema è la **Grad-CAM**, la quale usa il **gradiente** per calcolare la heatmap a bassa risoluzione g^c , dove

$$g^c = \text{RELU} \left(\sum_k w_k^c a_k \right) \quad (4.13)$$

$$w_k^c = \frac{1}{nm} \sum_i \sum_j \frac{\partial y_c}{\partial a_k(i, j)} \quad (4.14)$$

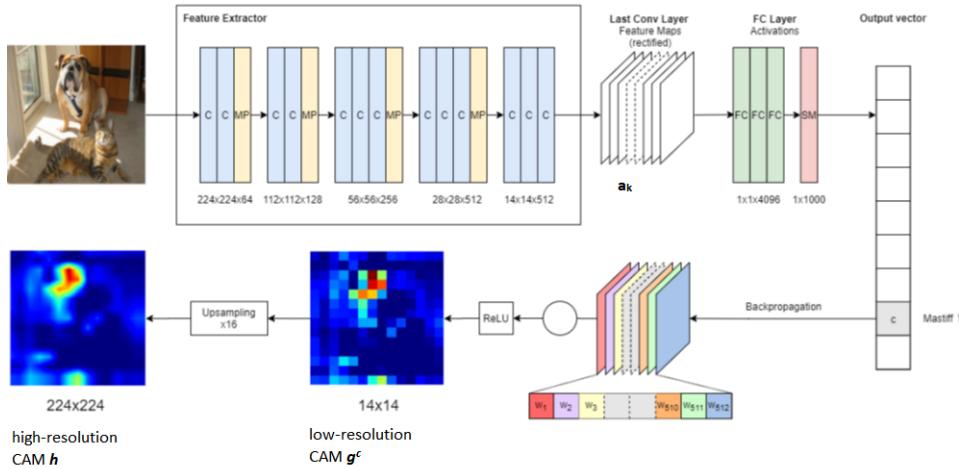


Figura 4.30: Grad-CAM

Augmented Grad-CAM Consideriamo l’operatore di augmentation $\mathcal{A}_l : \mathbb{R}^{N \times M} \rightarrow \mathbb{R}^{N \times M}$, che include rotazioni e traslazioni randomiche dell’immagine di input \mathbf{x} . L’**Augmented Grad-CAM** incrementa la risoluzione delle heatmap tramite **image augmentation**. Tutte le risposte generate dalla CNN alle versioni multiple aumentate della stessa immagine di input hanno molte informazioni per la ricostruzione della heatmap \mathbf{h} ad alta risoluzione.

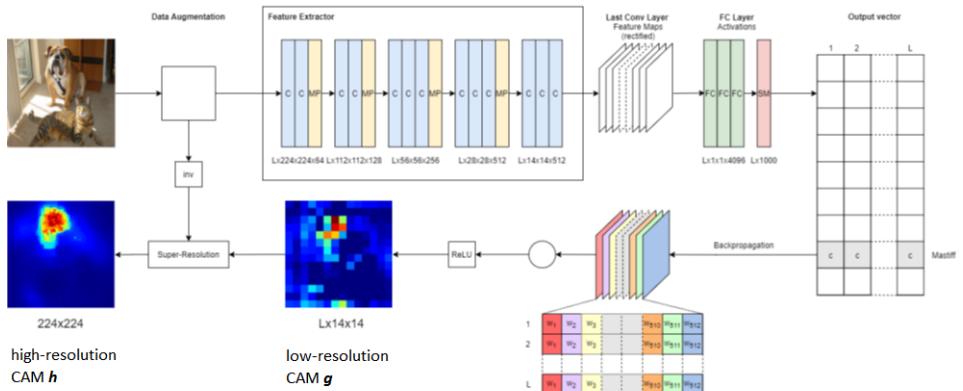


Figura 4.31: Augmented Grad-CAM

Viene fatta la cosiddetta **Super-Resolution (SR)** della heatmap sfruttando le informazioni condivise nelle heatmap a bassa risoluzione calcolate dallo stesso input sotto diverse, ma note, trasformazioni. Le CNN sono in generale invarianti alle rototraslazioni, in termini di predizioni, ma ogni heatmap a bassa risoluzione

\mathbf{g}_l contiene di fatto diverse informazioni. Modelliamo le heatmap calcolate col Grad-CAM come risultato di un operatore di downsampling sconosciuto $\mathcal{D} : \mathbb{R}^{N \times M} \rightarrow \mathbb{R}^{N \times M}$. L'alta risoluzione della heatmap \mathbf{h} è recuperata risolvendo un problema inverso

$$\operatorname{argmin}_{\mathbf{h}} \frac{1}{2} \sum_{l=1}^L \left\| \mathcal{D}\mathcal{A}_l \mathbf{h} - \mathbf{g}_l \right\|_2^2 + \lambda TV_{l_1}(\mathbf{h}) + \frac{\mu}{2} \|\mathbf{h}\|_2^2 \quad (4.15)$$

dove

$$TV_{l_1}(\mathbf{h}) = \sum_{i,j} \left\| \partial_x \mathbf{h}(i,j) \right\| + \left\| \partial_y \mathbf{h}(i,j) \right\| \quad (4.16)$$

è la **regolarizzazione Anisotropic Total Variation** usata per preservare i bordi della heatmap target ad alta risoluzione. Questo viene risolto tramite **Subgradient Descent** (https://en.wikipedia.org/wiki/Subderivative#The_subgradient) dato che la funzione è convessa e non-smooth (non differenziabile e discontinua).

4.9 Localizzazione

4.9.1 R-CNN

Per quanto riguarda il task del riconoscimento degli oggetti (**object detection**), una possibile soluzione potrebbe essere quella di usare una finestra scorrevole (**sliding window**) che ha un'etichetta assegnata al pixel centrale e consente di analizzare un'immagine di dimensioni variabili a piccoli pezzi di dimensioni fisse. Tali “pezzi”, cioè regioni di immagine, vengono poi date in pasto a un modello preallenato per effettuare predizioni sulle stesse. Questo approccio ha, però, vari contro, tra cui il fatto che non riutilizza features condivise fra regioni sovrapposte. Una possibile idea per risolvere questo problema è usare un algoritmo a proposta di regione (**region proposal algorithm**), che classifica tramite una CNN l'immagine dentro ogni regione proposta. Tale rete prende quindi il nome di R-CNN (dove R sta per **regioni**).

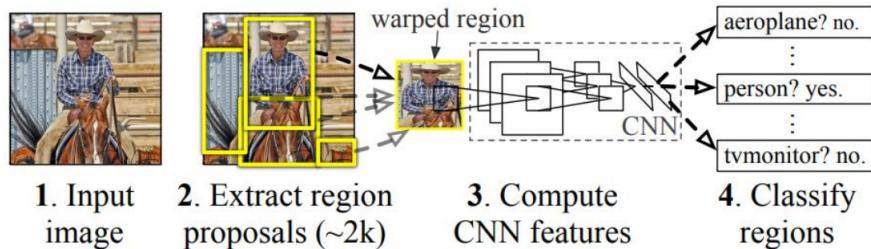


Figura 4.32: Funzionamento R-CNN

Lo step 4 viene formato da un regressore **Support Vector Machine (SVM)**, allenato per minimizzare l'errore di classificazione sulla regione di interesse (Region of Interest, **ROI**) estratta e da un regressore **Bounding Box (BB)**, che rifinisce le regioni correggendo la stima del bounding box dall'algoritmo di estrazione delle ROI. La CNN pre allenata, invece, è fine-tunata sulle classi da rilevare inserendo un FC layer dopo l'estrazione delle feature. Per scartare le regioni che non corrispondono a nessun oggetto, inoltre, va inclusa la classe di background.

4.9.2 Fast R-CNN

Le R-CNN hanno delle limitazioni, fra cui la lentezza computazionale, perciò sono state introdotte le Fast R-CNN, nelle quali:

1. Le immagini intere sono date in pasto alla CNN che estrae le feature maps.
2. Le proposte di regione sono identificate a partire dall'immagine e proiettate nelle feature maps. Inoltre, le regioni sono direttamente ritagliate dalle feature maps anziché dall'immagine (si riusa la computazione convoluzionale).
3. Una dimensione fissata è comunque richiesta per poter dare i dati in pasto a un FC layer. Dopodiché i layer di ROI pooling estraggono un vettore di feature di dimensione fissa $H \times W$ da ogni regione proposta. In pratica ogni ROI nelle feature maps è diviso in una griglia $H \times W$ su cui viene fatto un maxpooling per ottenere il feature vector (vettore di feature).
4. I layer FC stimano sia le classi che la posizione dei bounding box (ovvero, come nel regressore BB). Viene poi usata una combinazione convessa (cioè una combinazione lineare con termini aventi coefficienti non-negativi la cui somma è 1) dei due come loss multitask da ottimizzare (come nelle R-CNN, ma senza l'uso di regressori SVM).

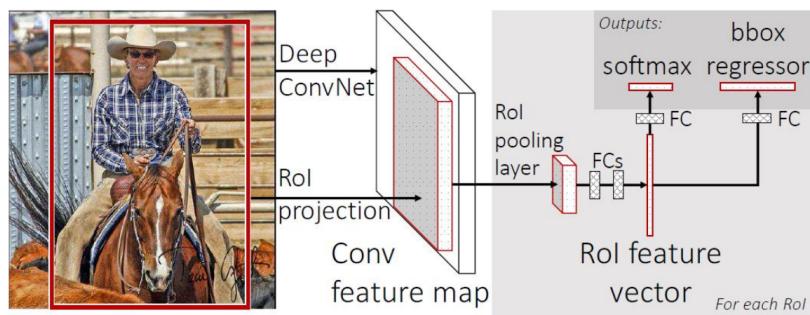


Figura 4.33: Funzionamento Fast R-CNN

In questa architettura è quindi possibile retro-propagare l'errore attraverso l'intera rete, in modo che essa venga allenata in maniera end-to-end. Dal momento che le convoluzioni non sono ripetute su aree sovrapposte, la stragrande maggioranza del tempo di test è speso sull'estrazione delle ROI.

4.9.3 Faster R-CNN

Invece dell'algoritmo di estrazione delle ROI, le **Faster R-CNN** allenano una **Region Proposal Network (RPN)**, la quale è una Fully-CNN. La RPN opera sulle stesse feature maps usate per la classificazione, cioè negli ultimi layer convoluzionali. La RPN può quindi essere vista come un modulo aggiuntivo che migliora l'efficienza e concentra la Fast R-CNN sulle regioni più promettenti per il riconoscimento degli oggetti.

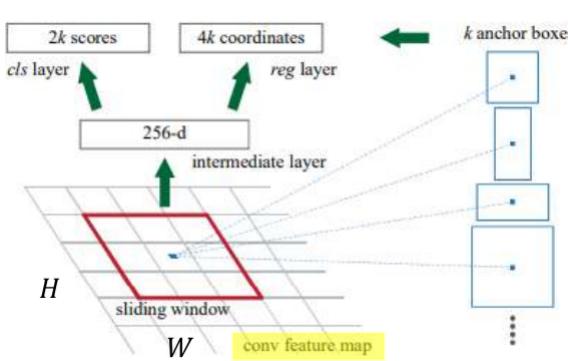


Figura 4.34: RPN

Il suo scopo è quello di associare a ogni locazione spaziale k box “ancora”, ovvero ROI aventi scale e ratio differenti. Tale rete produrrà in output delle ancore candidate di dimensione $H \times W \times k$ insieme a dei punteggi di stima per ogni ancora. Il layer intermedio di una RPN è un layer convoluzionale standard che prende come input l'ultimo layer della rete di estrazione delle feature e usa un filtro di dimensione $3 \times 3 \times 256$. Tale layer è usato per ridurre la dimensionalità della feature map, cioè mappa una certa regione a un vettore di dimensione inferiore $H \times W \times 256$. Vi sono poi 2 layer, cioè:

- **La rete di classificazione:** allenata per predire la **probabilità dell'oggetto**, cioè la probabilità che una certa ancora contenga un oggetto (dovrà quindi produrre $2k$ punteggi in output, ovvero k coppie con probabilità <contiene, non contiene>). Essa è fatta da uno stack di layer convoluzionali di dimensione 1×1 . Ognuna delle k coppie di probabilità corrisponde a un'ancora specifica ed esprime la probabilità che quell'ancora, in quella locazione spaziale, contenga un oggetto.

- La **rete di regressione**: allenata per aggiustare ognuna delle k ancore predette (deve quindi fare $4k$ stime, ovvero una stima per ognuna delle 4 coordinate del bounding box e per ogni bounding box).

Abbiamo detto che la RPN restituisce in output $H \times W \times k$ proposte di regioni, quindi rimpiazza l'algoritmo di proposta di regioni (**selective search**). Nelle Faster R-CNN, dopo la RPN vi è una soppressione dei non-massimi basata su punteggi di oggettività. Le proposte rimanenti sono quindi date in input al layer di ROI pooling e infine classificate dall'architettura Fast R-CNN standard.

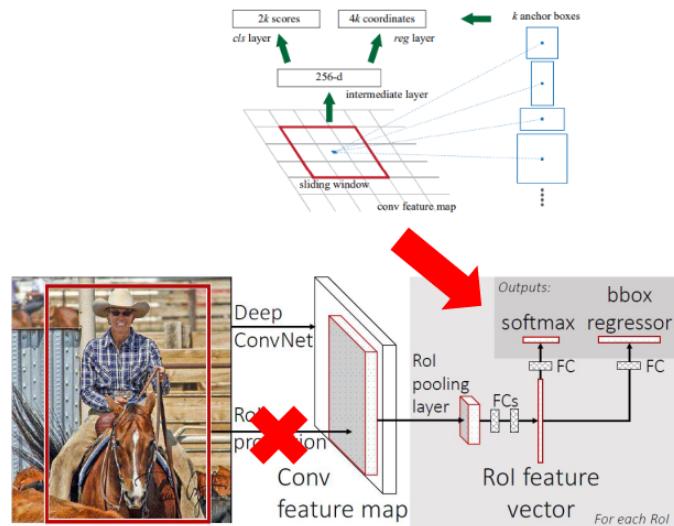


Figura 4.35: Faster R-CNN

La procedura di training è la seguente:

1. Allenare la RPN mantenendo congelata la rete strutturale (Deep ConvNet) e allenando solo i layer RPN.
2. Si allena la Fast R-CNN usando le proposte di regioni restituite dalla RPN allenata nello step precedente, cioè si fa fine-tuning dell'intera Fast R-CNN includendo la Deep ConvNet.
3. Si fa fine-tuning in cascata della RPN.
4. Si congela la Deep ConvNet e si fa fine-tuning solo degli ultimi layer della Faster R-CNN.

4.10 Altri esempi di architetture

4.10.1 YOLO (You Only Look Once)

YOLO è un metodo *region-free*, in cui riformuliamo il riconoscimento degli oggetti come un problema di regressione singolo che parte dai pixel dell'immagine alle coordinate dei bounding box e alle probabilità delle classi. Esso risolve questi problemi di regressione in una volta sola, tramite una CNN larga.

Gli step su cui si basa YOLO sono:

1. Dividere l'immagine in una griglia grossolana (di dimensione $N \times M$).
2. Ogni griglia contiene B ancore (**bounding box base**) associate.
3. Per ogni cella e per ogni ancora prediciamo:
 - (a) l'offset del bounding box base: ($dx, dy, dh, dw,$ punteggio dell'oggettività).
 - (b) il punteggio di classificazione del bounding box base sulle C categorie considerate (includendo lo sfondo).

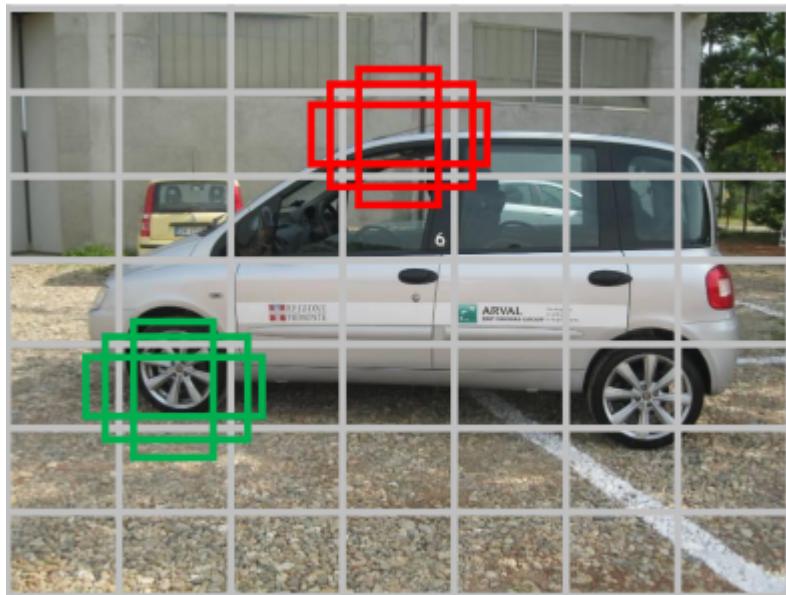


Figura 4.36:

L'output della rete avrà quindi dimensione: $N \times M \times B \times (5 + C)$. L'intera predizione è fatta in un singolo passo di forward sull'immagine e da una singola rete convoluzionale. Ciò lo rende molto veloce, ma meno accurato rispetto alle R-CNN.

4.10.2 Mask R-CNN

La segmentazione d'istanza combina le sfide di rilevamento degli oggetti (istanze multiple presenti nell'immagine) e di segmentazione semantica (etichette associate a ogni pixel). Un'architettura che risolve tale problema è la Mask R-CNN.

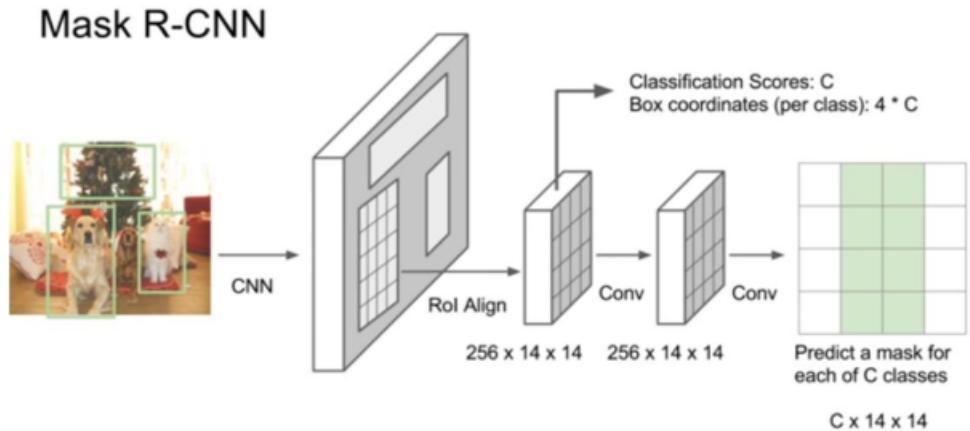


Figura 4.37: Mask R-CNN

Come nelle Fast R-CNN, viene classificata l'intera ROI e viene fatta una regressione sul bounding box (possibilmente stimando la posa degli oggetti). Dentro ogni ROI viene poi fatta, per ogni classe, una segmentazione semantica tramite la stima di una maschera (vedi figura sotto).



Figura 4.38: Output mask R-CNN

Capitolo 5

Reti Ricorrenti

5.1 Modellazione sequenziale

Finora abbiamo considerato solo dataset «statici». Ora invece ci concentreremo su dataset «*dinamici*» in due modi:

- **modelli senza memoria**
 - **modelli autoregressivi**: predicono il prossimo input a partire dal precedente sfruttando dei «ritardi»
 - **Feed Forward Neural Networks**: generalizzano i modelli autoregressivi usando layer nascosti non-lineari

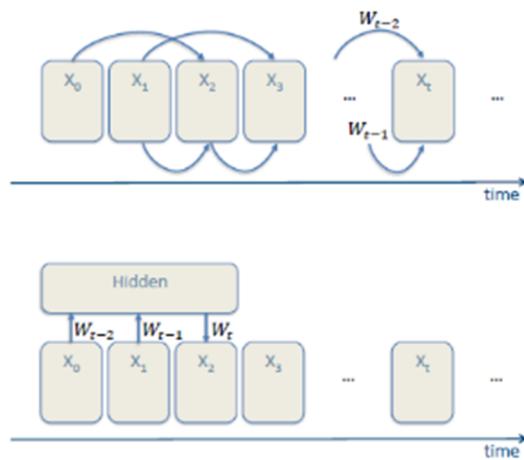


Figura 5.1: Modello autoregressivo (sopra), FFNN (sotto)

- **modelli con memoria** (sistemi dinamici lineari, Hidden Markov models, Recurrent Neural Networks)

I **sistemi dinamici** sono modelli generativi con uno stato nascosto che non può essere osservato direttamente. Tale stato ha delle dinamiche che possono essere affette da rumore e produce l'output.

5.1.1 Sistemi dinamici lineari

Nei **sistemi dinamici lineari** ciò si traduce nell'avere uno stato continuo con incertezza Gaussiana che può essere stimato usando il **Kalman filtering**. Le trasformazioni sono assunte essere lineari.

5.1.2 Hidden Markov models

Negli **HMM** lo stato è assunto come discreto e può essere stimato tramite l'**algoritmo di Viterbi**, mentre le transizioni sono *stocastiche* (si usa la cosiddetta "matrice di transizione"). L'output è una funzione stocastica degli stati nascosti.

5.1.3 Recurrent Neural Networks

Le **reti neurali ricorrenti** implementano la memoria tramite delle connessioni chiamate **connessioni ricorrenti**. Inoltre, gli stati distribuiti consentono di salvare efficientemente le informazioni, mentre le dinamiche non-lineari consentono di aggiornare stati nascosti complessi.

"Con abbastanza neuroni e tempo, le RNN possono calcolare qualsiasi cosa che possa essere calcolata da un computer." (Computation Beyond the Turing Limit, Hava T. Siegelmann, 1995)

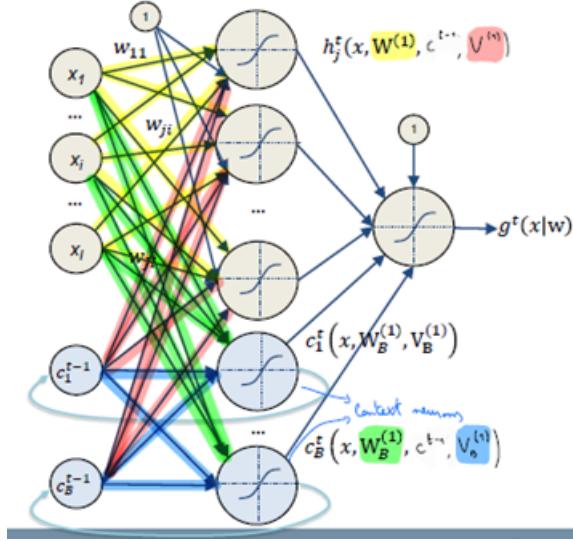


Figura 5.2: Rete neurale ricorrente

Nella figura sopra sono raffigurati due tipi di neuroni: quelli grigi, che dipendono dall'**input corrente**, e quelli blu, che dipendono dalla **storia della rete**. Inoltre abbiamo dei parametri nuovi chiamati:

- V rappresenta i pesi tra neuroni blu e neuroni grigi
- V_B rappresenta i pesi tra i neuroni blu
- W rappresenta i pesi tra neuroni grigi
- W_B rappresenta i pesi tra neuroni grigi e neuroni blu

Le funzioni di output dei neuroni sono

$$h_j^t(\cdot) = h_j^t \left(\sum_{j=0}^J w_{ji}^{(1)} \cdot x_{i,n} + \sum_{b=0}^B v_{jb}^{(1)} \cdot c_b^{t-1} \right) \quad (5.1)$$

$$c_b^t(\cdot) = c_b^t \left(\sum_{j=0}^J w_{bi}^{(1)} \cdot x_{i,n} + \sum_{b'=0}^B v_{bb'}^{(1)} \cdot c_{b'}^{t-1} \right) \quad (5.2)$$

mentre la funzione di output della rete è

$$g^t(x_n|w) = g \left(\sum_{j=0}^J w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^B v_{1b}^{(2)} \cdot c_b^t(\cdot) \right) \quad (5.3)$$

La rete nascosta è chiamata **context network** e contiene B neuroni, mentre la rete "visibile" ne ha J . La context network è la parte più difficile da allenare in quanto non è una FFNN.

La tecnica usata per allenarla è chiamata **Backpropagation Through Time**. Gli step che la compongono sono:

1. disiegare (unfold) la RNN per U step temporali, ottenendo una FFNN: sia N una RNN che deve apprendere un task temporale a partire dal tempo $t-u$ fino al tempo t , e sia N^* la FFNN che risulta dall'unfold della rete N :

- per ogni istante di tempo nell'intervallo $(t-u, t]$ la rete N^* ha un layer contenente k neuroni, dove k è il numero di neuroni di N
- per ogni layer di N^* c'è una copia di ogni neurone in N
- per ogni istante di tempo $\tau \in (t-u, t]$ il peso sinaptico dal neurone i nel layer τ al neurone j nel layer $\tau+1$ di N^* è una copia della connessione sinaptica dal neurone i al neurone j in N

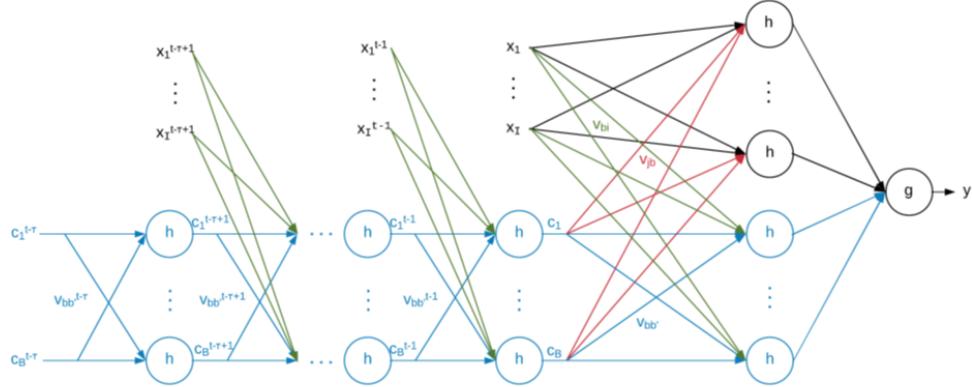


Figura 5.3: Unfolding della rete

2. inizializzare le repliche di W_B e V_B in modo da renderle uguali
3. fare una backpropagation su tutta la nuova rete: tutti i pesi sono allenati col gradient descent, cioè considerando un istante di tempo generico τ . Applicando il solito metodo di aggiornamento, avremo valori diversi per gli stessi pesi in diversi istanti di tempo. Ciò non è conveniente perché di fatto sono gli stessi pesi, quindi poi se ne fa la media dopo averli aggiornati sulla base del gradient descent. Una tecnica alternativa possibile è quella di aggiornare i pesi con la media del gradiente, ottenendo quindi gli stessi risultati ma il processo sarà più efficiente. La formula di aggiornamento per i pesi da fornire alla context network è:

$$W_B = W_B - \eta \cdot \frac{1}{U} \sum_0^{U-1} \frac{\partial E}{\partial W_B^{t-u}} \quad (5.4)$$

Mentre per mantenere la memoria del contesto si usa:

$$V_B = V_B - \eta \cdot \frac{1}{U} \sum_0^{U-1} \frac{\partial E^t}{\partial V_B^{t-u}} \quad (5.5)$$

5.2 Vanishing Gradient

Le RNN non riescono ad andare indietro nel passato per più di 10 step a causa del cosiddetto effetto del **Vanishing Gradient**. Per spiegarlo proviamo a semplificare la RNN come nella figura sotto.

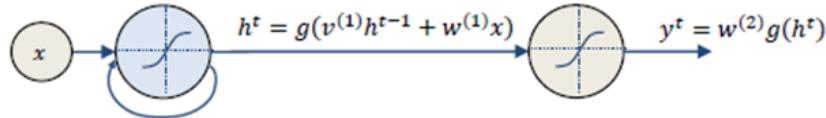


Figura 5.4:

La backpropagation su un'intera sequenza S è calcolata come

$$\frac{\partial E}{\partial w} = \sum_{t=1}^S \frac{\partial E^t}{\partial w} = \sum_{t=1}^S \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial h^k}{\partial w} \quad (5.6)$$

dove

$$\frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^t \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^t v^{(1)} g'(h^{i-1}) \quad (5.7)$$

Considerando la norma di questi termini

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| = \| v^{(1)} \| \| g'(h^{i-1}) \| \quad (5.8)$$

otteniamo che

$$\left\| \frac{\partial h^t}{\partial h^k} \right\| \leq \left(\gamma_v \gamma_g' \right)^{t-k} \quad (5.9)$$

dove se $(\gamma_v \gamma_g') < 1$ allora la norma converge a 0. In particolare γ_v è la norma dei pesi e γ_g' è la norma della derivata g' e vale 1 se usiamo una ReLU. Usando sigmoide o Tanh otteniamo il cosiddetto **Vanishing Gradient**, che dimostra che più si va indietro nel passato, più il gradiente tenderà a 0 indipendentemente dall'errore. Basandosi sullo stesso principio, se $v^{(1)} > 1$ il gradiente incrementerà drammaticamente fino a risultare in un'«esplosione» che colpirà la capacità di apprendimento della rete.

Per risolvere questo problema si può usare la ReLU, che forza tutti i gradienti ad essere 0 o 1. Infatti ricordiamo che:

$$ReLU(x) = f(x) = \max(0, x) \quad (5.10)$$

$$f'(x) = 1_{x>0} \quad (5.11)$$

Il trick sta quindi nel costruire la RNN usando piccoli moduli fatti apposta per ricordare valori per tanto tempo.

5.3 Long Short-Term Memories

Nel 1997, Hochreiter&Schmidhuber hanno risolto il problema del vanishing gradient disegnando una **cella di memoria** che usa unità logistiche e lineari con interazioni moltiplicative. L'informazione entra nella cella quando il suo gate di "write" è on, rimane nella cella fintantochè il suo gate di "keep" è on, e viene infine letta dalla cella mettendo a on il suo gate di "read".

Il problema viene quindi risolto poiché il loop ha un peso fissato, cioè non vanno imparati i pesi della parte di rete "unfoldata" dato che valgono 1.

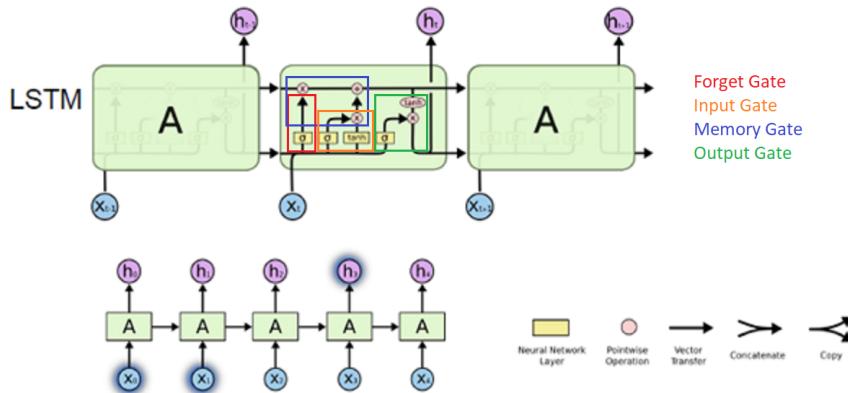


Figura 5.5: LSTM

La **LSTM** è composta da:

- un **input gate** (ci dice quanto teniamo in memoria)

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \quad (5.12)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \quad (5.13)$$

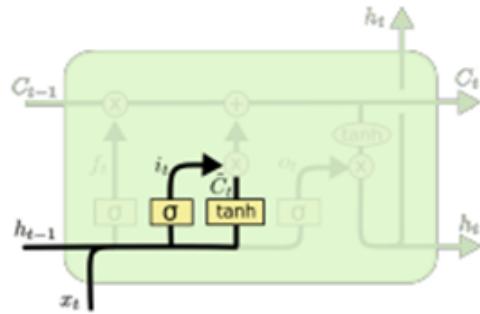


Figura 5.6: Input gate

- un **forget gate**

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \quad (5.14)$$

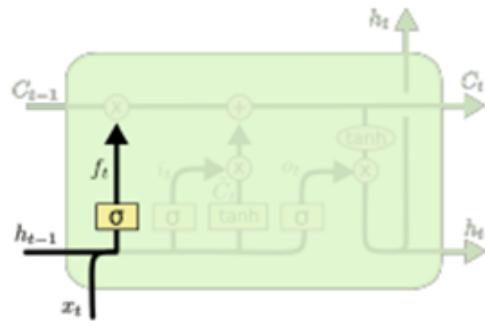


Figura 5.7: Forget gate

- un **memory gate**

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \quad (5.15)$$

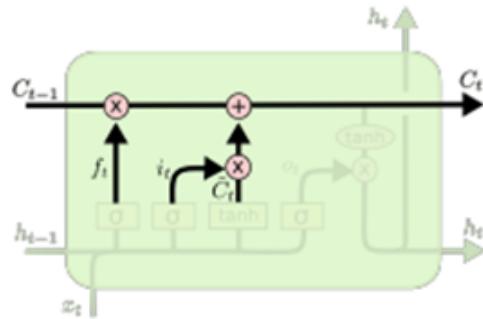


Figura 5.8: Memory gate

- un **output gate**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \quad (5.16)$$

$$h_t = o_t * \tanh(C_t) \quad (5.17)$$

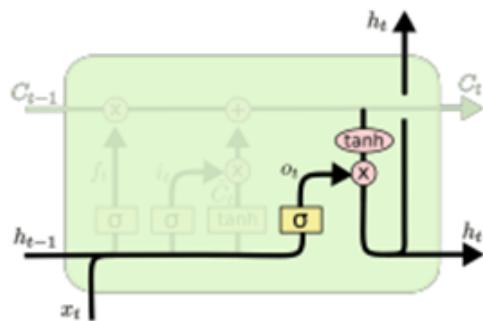


Figura 5.9: Output gate

Possiamo quindi costruire un grafo computazionale con trasformazioni continue come nella figura sotto. In pratica le LSTM funzionano come i layer convoluzionali nelle CNN perché sono una sorta di *estrattori di features*.

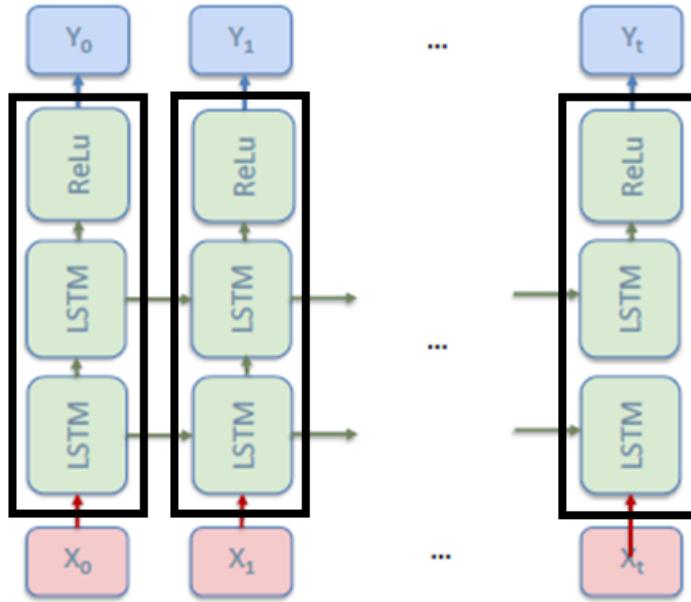


Figura 5.10: LSTM in sequenza (le parti nei riquadri neri sono i layer nascosti)

Un altro approccio possibile consiste nell'usare reti **LSTM bidirezionali** che sfruttano l'intera sequenza in input. Sono composte da una RNN che attraversa l'intera sequenza da sinistra a destra e da una RNN che attraversa l'intera sequenza da destra a sinistra. Queste due RNN vengono poi concatenate per essere usate come rappresentazione delle features. Per inizializzarle bisogna specificare lo stato iniziale trattandolo come parametro da apprendere. L'inizializzazione può essere fatta a 0 oppure a valori randomici. Dopodichè si inizia a predire randomicamente i valori dello stato iniziale, si backpropaga l'errore di predizione nel tempo fino allo stato iniziale e si calcola il gradiente dell'errore rispetto ai nuovi valori dello stato iniziale.

Gated Recurrent Unit La **GRU** combina i gate di input e forget in un unico **"update gate"** e fa un merge tra lo stato della cella e lo stato nascosto (più altri cambiamenti). Le parti che caratterizzano tale unità sono:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t]) \quad (5.18)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t]) \quad (5.19)$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t]) \quad (5.20)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t \quad (5.21)$$

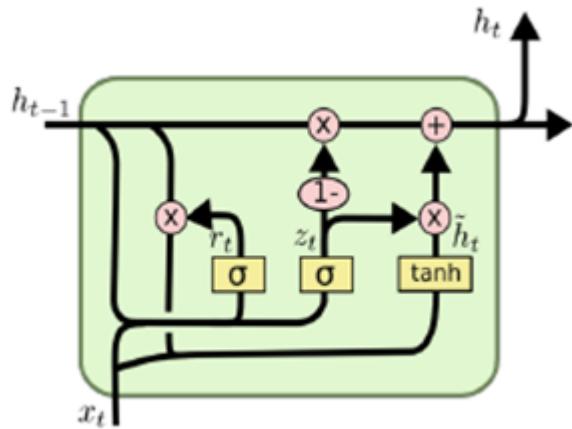


Figura 5.11: GRU

La GRU può essere usata in alternativa alla LSTM.

5.4 Modellazione Sequence to Sequence

Il **sequence modeling** risolve vari problemi in base all'architettura usata. Le architetture possibili sono:

- **uno a uno:** un input di dimensione fissa viene trasformato in un output di dimensione fissata (es.: image classification)
- **uno a molti:** output sequenziale (es.: captioning delle immagini che prende un'immagine in input e produce in output una sequenza di parole)
- **molti a uno:** input sequenziale (es.: analisi dei sentimenti dove una data frase viene classificata sulla base della positività o negatività dei sentimenti che esprime)
- **molti a molti:**
 - input e output sequenziali (es.: traduttori, cioè una RNN che legge una frase in inglese e ne produce in output la traduzione francese)
 - input e output sequenziali sincronizzati (es.: classificazione dei video, dei quali vogliamo etichettare ogni frame)

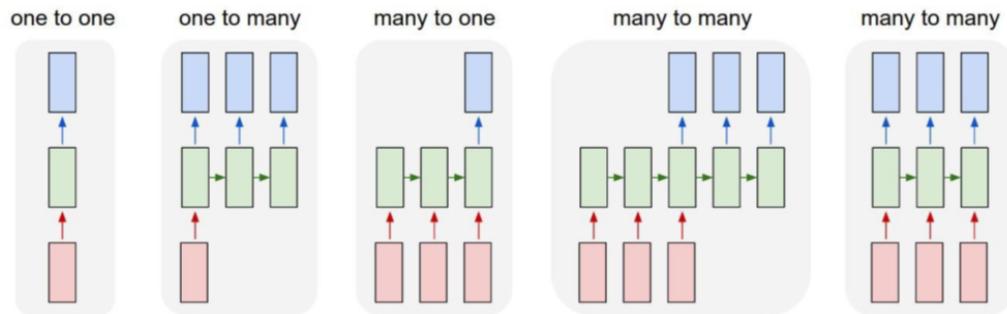


Figura 5.12: Architetture seq2seq

Il modello seq2seq segue la classica **architettura encoder-decoder**, in cui durante la fase di inferenza (e non durante il training) il decoder dà in pasto l'output di ogni istante di tempo all'input successivo.

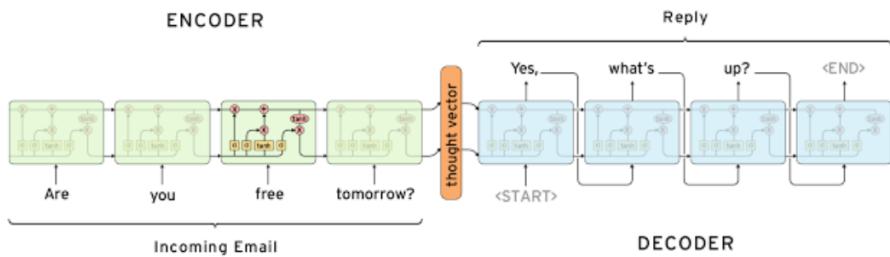


Figura 5.13: Architettura Encoder-Decoder

Il processo di *training* è caratterizzato da una **trasformazione** delle parole a identificativi e da una fase di **embedding**. Il processo di inferenza (testing) è caratterizzato anch'esso da una prima fase di **embedding** e da una **trasformazione** di identificativi in parole. La fase di embedding consiste nella rappresentazione delle parole usando un vettore di parole (vocabolario) denso. L'encoder può essere una RNN così come una CNN, ad esempio nel caso di image captioning.

Vi sono inoltre dei **caratteri speciali** che vengono dati in input al decoder:

- **<PAD>**: durante il training gli esempi sono dati in pasto alla rete in batch, i cui input devono essere della stessa larghezza. Questo carattere viene usato per "gonfiare" input più corti rendendoli della stessa dimensione del batch
- **<EOS>**: necessario per il batching nel decoder, ovvero indica al decoder dove finisce la frase e gli consente di indicare la stessa cosa nel suo output

- <UNK>: su dati reali può ampiamente migliorare l'efficienza delle risorse per ignorare le parole che non si presentano abbastanza spesso nel vocabolario (rimpiazzandole quindi con questo carattere)
- <SOS>/<GO>: questo è l'input del decoder nel primo istante di tempo per consentirgli di capire quando iniziare a generare l'output

La **preparazione dei batch** si compone dei seguenti step:

1. prendere un campione di coppie <source_sequence, target_sequence> di dimensione batch_size
2. appendere <EOS> a source_sequence
3. prependere <SOS> a target_sequence per ottenere la target_input_sequence e appendere <EOS> per ottenere la target_output_sequence
4. "gonfiare" le frasi fino alla max_input_length (o max_target_length) all'interno dello stesso batch usando il token <PAD>
5. codificare i token basandosi sul vocabolario (embedding)
6. sostituire i token OOV (out of vocabulary) con <UNK> e calcolare la lunghezza di ogni sequenza di input e target nel batch

Il **modello seq2seq** ci dice quindi che, data una coppia <S,T>, legge S e restituisce in output T' che corrisponda a T. In pratica il problema riguarda la probabilità di ottenere una sequenza in output data una sequenza in input:

$$p(y_1, \dots, y_{T'} | x_1, \dots, x_T) = \prod_{t=1}^{T'} p(y_t | v, y_1, \dots, y_{t-1}) \quad (5.22)$$

dove v è lo stato nascosto. Quindi, tale problema si riconduce alla **massimizzazione della crossentropy media**:

$$\frac{1}{|S|} \sum_{(T,S) \in \mathcal{S}} \log p(T|S) \quad (5.23)$$

5.5 Macchine neurali di Turing

Le **Neural Turing Machines** combinano una RNN con una banca di memoria esterna, cioè un array di vettori. La rete principale scrive su e legge da questa memoria a ogni step.

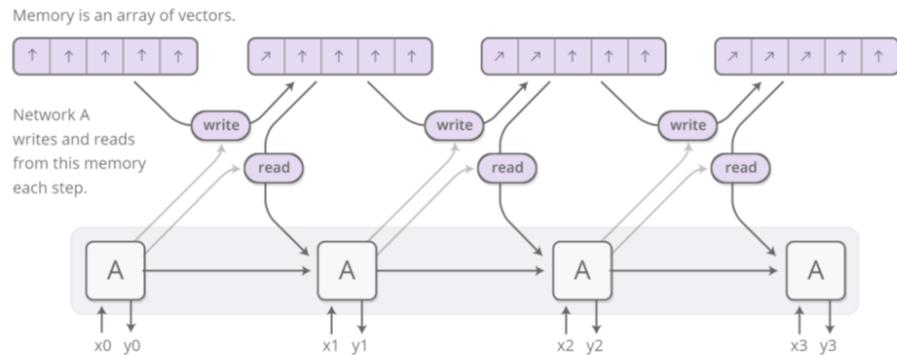


Figura 5.14: Neural Turing Machine

La sfida è quella di imparare cosa scrivere/leggere e dove scrivere/leggere. La soluzione è quella di, a ogni step, leggere e scrivere ovunque ma con diversa misura. Tale meccanismo è chiamato **attention mechanism** e si basa sul porre l'**attenzione** su una specifica parte della memoria per produrre l'output. In pratica la RNN dà una **distribuzione di attenzione** (dettata dalla softmax) che descrive come distribuiamo in memoria ciò che ci interessa (vedi figura sotto).

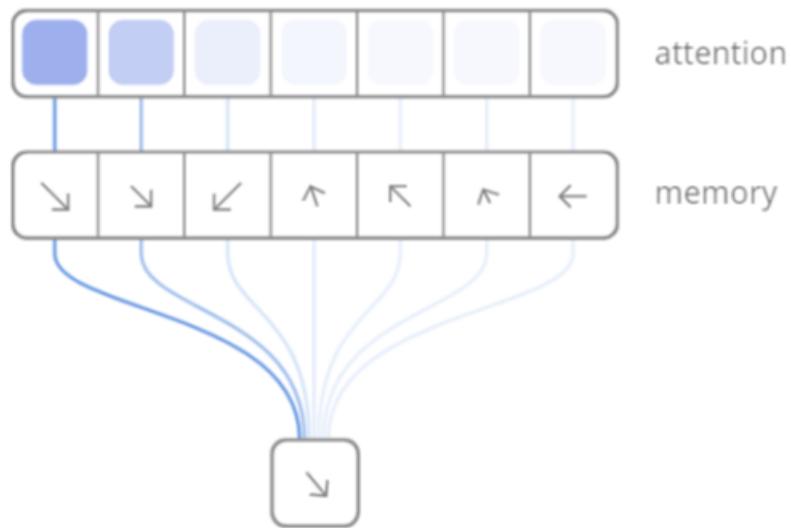


Figura 5.15: Attention mechanism in lettura

Il risultato della lettura è una somma pesata differenziabile:

$$r \leftarrow \sum_i a_i M_i \quad (5.24)$$

In fase di scrittura, invece, anziché scrivere in una posizione specifica, scriviamo ovunque ma con diversa misura. La RNN dà una distribuzione di attenzione, descrivendo quanto dovremmo cambiare ogni posizione della memoria nella direzione del valore di scrittura. Sotto vi è una figura che spiega questo passaggio.

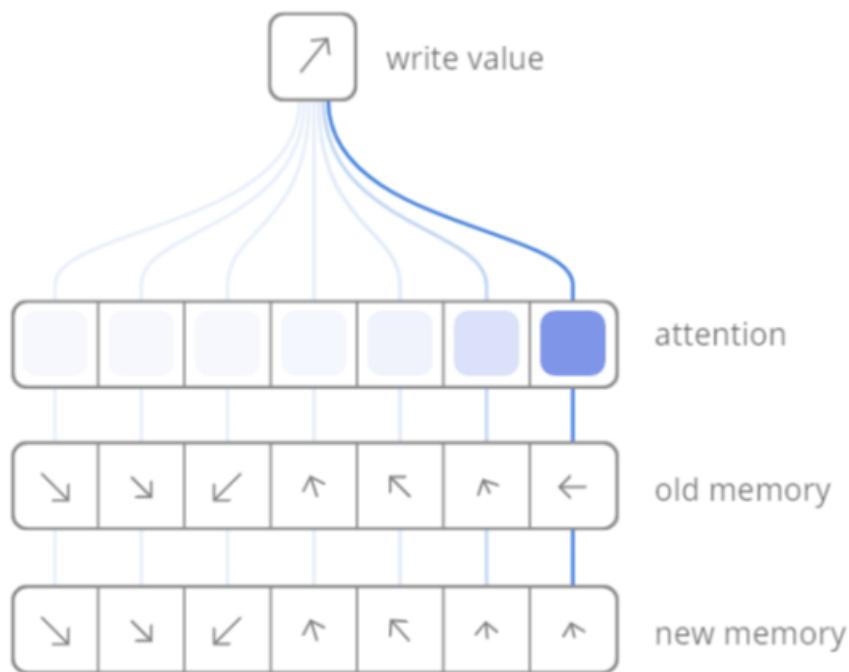


Figura 5.16: Attention mechanism in scrittura

L'aggiornamento della memoria si basa sulla formula seguente:

$$M_i \leftarrow a_i w + (1 - a_i) M_i \quad (5.25)$$

Meccanismo di attenzione Vi sono due componenti principali in una Neural Turing Machine: l'**attention mechanism** e il **RNN controller**. L'architettura è inoltre divisa in due parti:

- **content-based attention:** cerca in memoria e si concentra su posti che corrispondono a ciò che si sta cercando

- **location-based attention:** consente il movimento relativo nella memoria, abilitando la NTM a ciclare

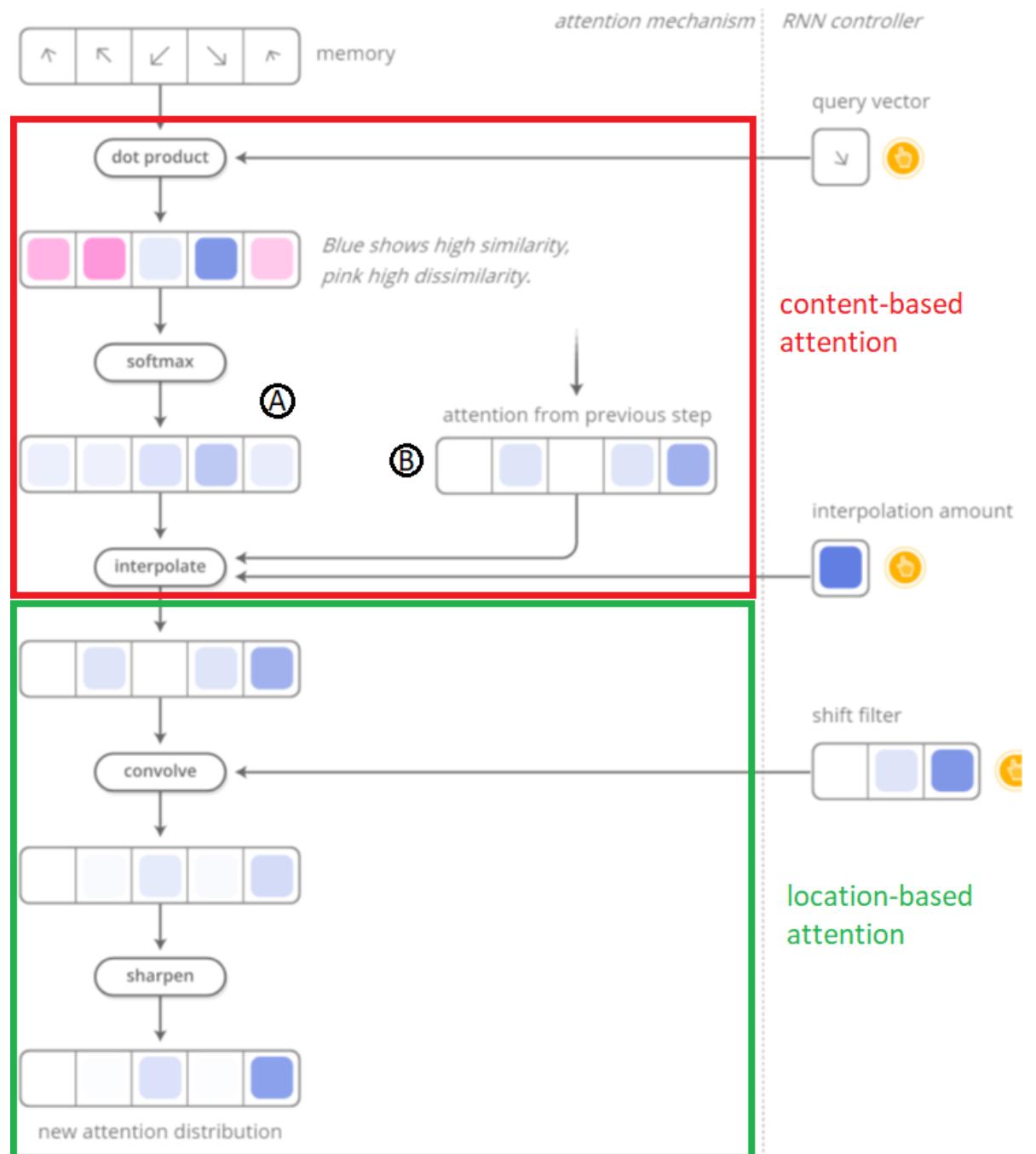


Figura 5.17: Attention in una NMT

Innanzitutto, il controller dà un **query vector** (output della RNN, usato come meccanismo di indirizzamento) all'attention mechanism e a ogni entry in memoria viene assegnato un punteggio basandosi sulla similarità con il vettore. I punteggi vengono quindi convertiti in una **distribuzione** di probabilità usando la softmax. Poi si **interpola** il risultato della softmax con l'attention generata dallo step precedente, tenendo conto di un **interpolation amount** che rappresenta la velocità di *interpolazione*, che è un parametro predetto dalla RNN (vedi figura 5.17: se è 0 si prende solo A, se è 1 si prende solo B). Fatto ciò, viene effettuata una convoluzione tra l'attenzione e un filtro di shift, che consente al controller di spostare il focus dell'attention mechanism su una particolare zona. Infine, si affina la **distribuzione dell'attention**, che viene poi data in pasto all'operazione di read o write.

Consideriamo il seguente dataset sequenziale: $\{((x_1, \dots, x_n), (y_1, \dots, y_n))\}_{i=1}^N$. Il ruolo del decoder è quello di modellare la probabilità generativa $P(y_1, \dots, y_m | x)$. Nei modelli seq2seq "vanilla" (convenzionali), il decoder viene condizionato inizializzando lo stato iniziale con l'ultimo stato dell'encoder. Ciò funziona bene per frasi medio-corte, mentre per frasi lunghe ciò diventa un *bottleneck*.

La **funzione di attention** mappa il query vector e l'insieme di coppie chiave-valore a un output, che è calcolato come la somma pesata dei valori, dove il peso assegnato a ogni valore è calcolato da una **funzione di compatibilità**. Tale funzione:

1. compara lo stato nascosto corrente h_t , con stati di origine h_s per derivare l'attention usando una funzione di scoring tra le due seguenti:

$$score(\mathbf{h}_t, \bar{\mathbf{h}}_s) = \begin{cases} \mathbf{h}_t^T \mathbf{W} \bar{\mathbf{h}}_s \\ \mathbf{v}_a^T \tanh(\mathbf{W}_1 \mathbf{h}_t + \mathbf{W}_2 \bar{\mathbf{h}}_s) \end{cases} \quad (5.26)$$

2. applica la softmax sugli score e calcola i pesi dell'attention, uno per ogni token dell'encoder, nel modo seguente:

$$\alpha_{ts} = \frac{e^{score(\mathbf{h}_t, \bar{\mathbf{h}}_s)}}{\sum_{s'=1}^S e^{score(\mathbf{h}_t, \bar{\mathbf{h}}_{s'})}} \quad (5.27)$$

3. calcola il **context vector** come media pesata degli stati di origine:

$$\mathbf{c}_t = \sum_s \alpha_{ts} \bar{\mathbf{h}}_s \quad (5.28)$$

4. combina il context vector con lo stato nascosto target corrente per produrre il vettore di attention finale:

$$\mathbf{a}_t = f(c_t, \mathbf{h}_t) = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{h}_t]) \quad (5.29)$$

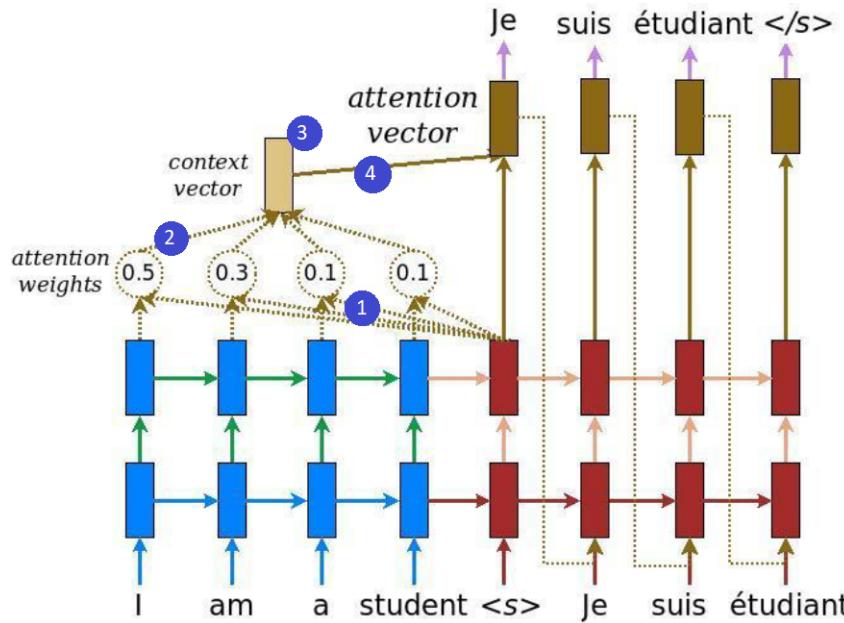


Figura 5.18: Esempio di funzione di compatibilità

Per visualizzare i pesi dell'attention tra le frasi di **source** e **target** si può usare la cosiddetta **alignment matrix**. Per ogni passo di decodifica (cioè per ogni token target generato) descrive quali sono i token di origine che sono più presenti nella somma pesata e che hanno quindi condizionato la decodifica. L'attention è quindi uno strumento che, durante la decodifica, consente alla rete di prestare più attenzione a varie parti della frase originale.

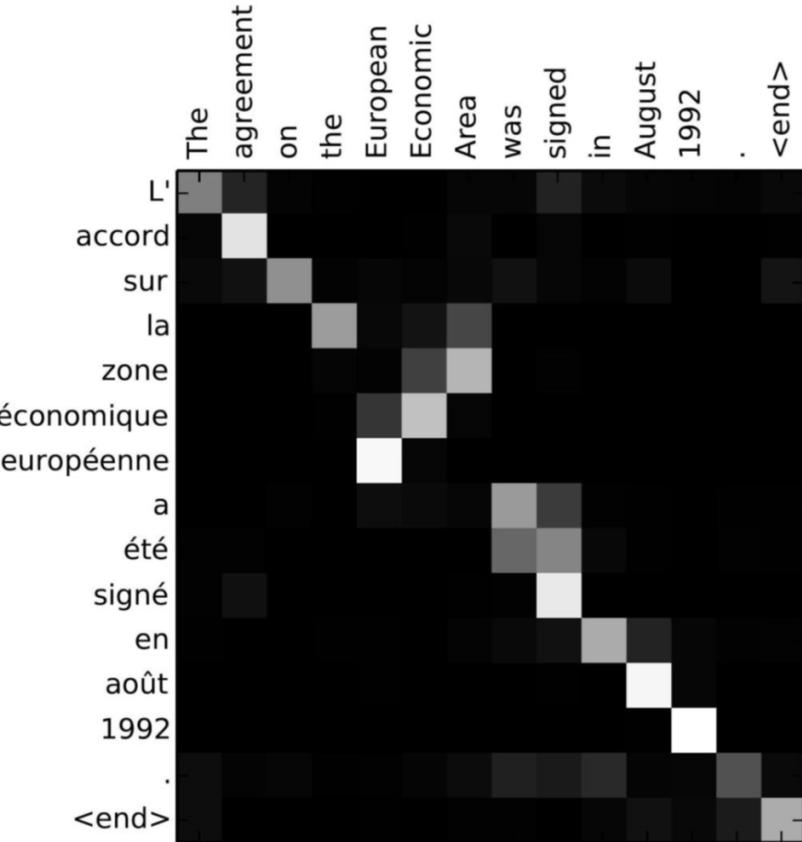


Figura 5.19: Esempio di alignment matrix

Tra le possibili applicazioni del meccanismo di attention, oltre alle traduzioni, troviamo il riconoscimento vocale, l'image captioning e la generazione automatica di risposte (es.: chatbot).

Chatbot I **chatbot** possono essere definiti lungo due dimensioni:

- l'**algoritmo core**:
 - **generativo**: codifica la domanda in un context vector e genera la risposta parola per parola usando la distribuzione della probabilità condizionata sul vocabolario della risposta (modello encoder-decoder)
 - **retrieval**: si basa sulla conoscenza di base delle coppie domanda-risposta, cioè quando viene fatta una nuova domanda, la fase di inferenza la codifica in un context vector e recupera, usando misure di

similarità, i top-k nearest neighbours basandosi sulla conoscenza di base posseduta

- la **gestione del contesto**:

- **single-turn**: costruisce il vettore di input considerando la domanda in arrivo (può perdere informazioni importanti riguardanti la storia della conversazione e generare, quindi, risposte irrilevanti, ma è più facile da implementare e allenare)

$$\{(\mathbf{q}_i, \mathbf{a}_i)\}$$

- **multi-turn**: costruisce il vettore di input considerando un contesto di conversazione "multi-turn", cioè basandosi sulla domanda in arrivo ma anche sulle domande precedenti

$$\{([\mathbf{q}_{i-2}; \mathbf{a}_{i-2}; \mathbf{q}_{i-1}; \mathbf{a}_{i-1}; \mathbf{q}_i], \mathbf{a}_i)\}$$

I chatbot generativi, in particolare, usano una RNN e la trainano per mappare ciò che viene detto dalla prima persona a ciò che viene risposto dal bot basandosi sempre sull'attention mechanism. Tale meccanismo di risposta single-turn è stato esteso nel 2017 in un **attention mechanism gerarchico**.

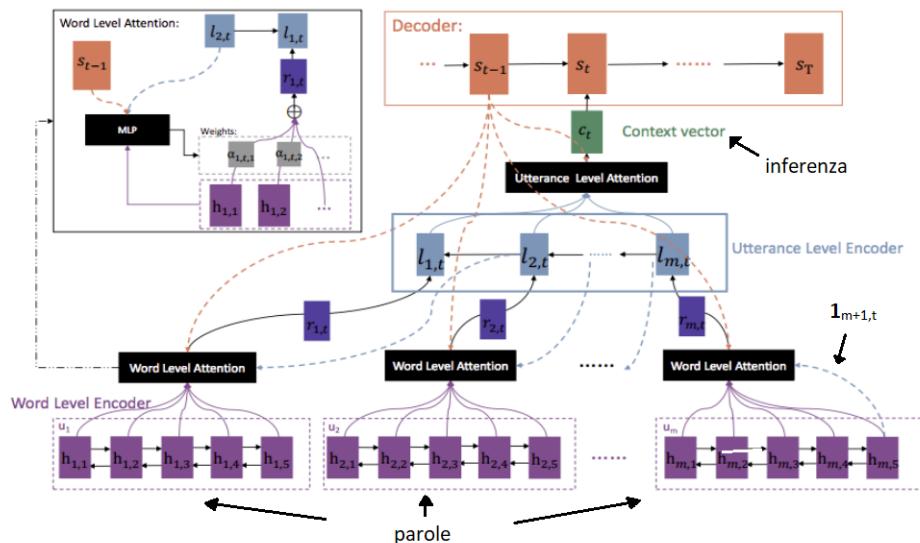


Figura 5.20: Chatbot gerarchico multi-turn generativo

In tale modello si usano le seguenti formule:

$$\mathbf{U} = (u_1, \dots, u_m) \quad (5.30)$$

$$\mathbf{u}_i = (w_{i,1}, \dots, w_{i,T_i}) \quad (5.31)$$

$$\mathbf{h}_{i,k} = \text{concat} \left(\overrightarrow{\mathbf{h}_{i,k}}, \dots, \overleftarrow{\mathbf{h}_{i,k}} \right) \quad (5.32)$$

$$\mathbf{r}_{i,t} = \sum_{j=1}^{T_i} \boldsymbol{\alpha}_{i,t,j} \mathbf{h}_{i,j} \quad (5.33)$$

$$e_{i,t,j} = \eta(\mathbf{s}_{t-1}, \mathbf{1}_{i+1,t}, \mathbf{h}_{i,j}) \quad (5.34)$$

$$\boldsymbol{\alpha}_{i,y,j} = \frac{\exp(e_{i,t,j})}{\sum_{k=1}^{T_i} \exp(e_{i,t,j})} \quad (5.35)$$

$$(\mathbf{1}_{i,t}, \dots, \mathbf{1}_{m,t}) \quad (5.36)$$

$$\mathbf{c}_t = \sum_{i=1}^m \beta_{i,t} \mathbf{1}_{i,t} \quad (5.37)$$

dove $\boldsymbol{\alpha}_{i,y,j}$ è la **softmax**. Le reti di attenzione gerarchiche sono state usate anche per la classificazione degli argomenti (es.: dataset di Yahoo Answers), così come per l'analisi dei sentimenti.

5.6 Trasformatore

Un modello **trasformatore** è caratterizzato da uno stack di encoder che codificano la sequenza in input e da uno stack di decoder che producono l'output. Ogni encoder e decoder hanno una struttura interna che consente di non usare un modello ricorrente. Ciò consente di allenare il modello con una quantità molto vasta di parole. L'elemento più importante che li caratterizza è la self-attention, che presta attenzione a una parola basandosi sulle parole che sono connesse ad essa.

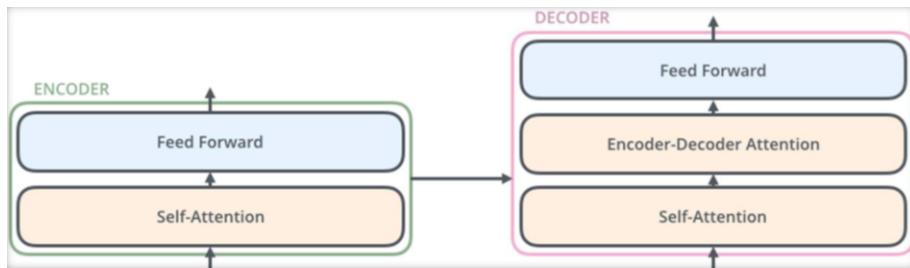


Figura 5.21: Encoder e decoder di un transformer

Andando più nel dettaglio, gli elementi che compongono un trasformatore sono:

- **dot-product attention scalata**
- **multi-head attention**

- FFNN posizionale
- embedding e softmax
- codifica posizionale

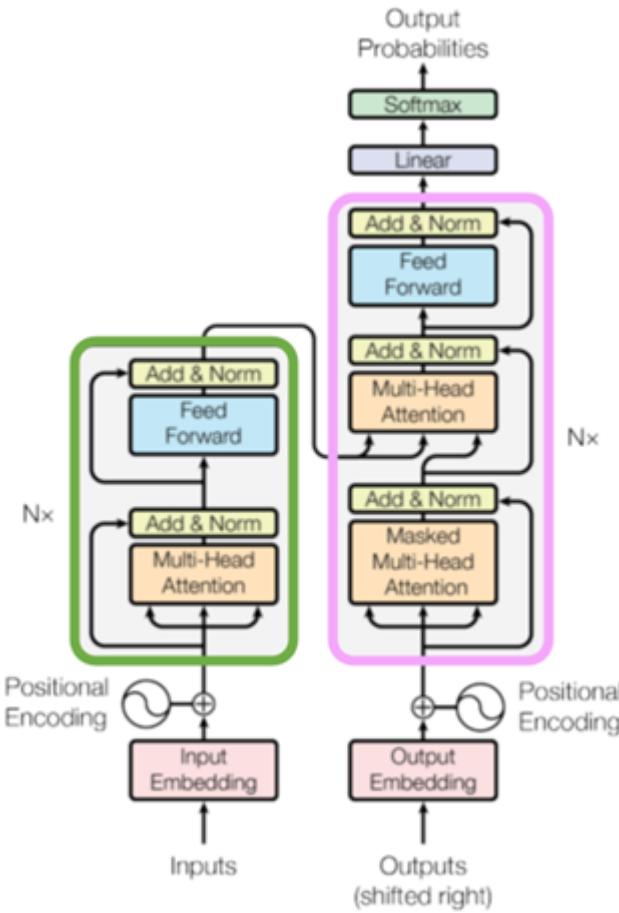


Figura 5.22: Architettura del Transformer

L'attention qui è definita come

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (5.38)$$

In particolare, Q, K e V sono delle matrici che indicano rispettivamente l'encoding delle parole in input (**query**), la trasformazione delle query in chiavi (**key**)

e il valore di queste parole trasformate in chiavi (**value**). Nella dot-product **attention scalata**, a ogni step viene paragonato ogni valore di query q_i con ogni altro valore chiave k_j .

Per quanto riguarda la multi-head attention, si cerca di calcolare la dot-product attention più volte in parallelo e poi concatenare i risultati.

Dopodichè si ha una **FFNN** che, per ogni parola in input, calcola la **multi-head attention** per quella parola e poi calcola una funzione non-lineare (sempre per la stessa parola):

$$FFNN(x) = \max(0, xW_1 + b_1) W_2 + b_2 \quad (5.39)$$

Vengono quindi ripetute le stesse operazioni sul decoder in modo tale da ottenere in output, per ogni parola in input, la probabilità di generare la 'giusta' parola (ad esempio, se in input ho "A B C" e in output voglio avere "D E F", il decoder mi calcolerà qual è la probabilità di generare D al primo step, E al secondo e F al terzo).

Infine, per tenere traccia della posizione delle parole in input si aggiunge un **encoding posizionale**, che assegna una sorta di timestamp alle parole. Ci sono due tipi di encoding posizionale, cioè il **learned positional embedding** e la **sinusoide**. Quest'ultima può essere:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{1000^{2i/d_{model}}}\right) \quad (5.40)$$

oppure:

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{1000^{2i/d_{model}}}\right) \quad (5.41)$$

5.7 Word embedding

La performance di un'applicazione nel mondo reale (es.: chatbot, classificatori di documenti, sistemi di recupero delle informazioni) dipende dalla codifica dell'input, che può essere basata su:

- rappresentazioni locali
 - **N-gram**
 - **bag-of-words**
 - **codifica 1-di-N**
- rappresentazioni continue
 - **Latent Semantic Analysis**
 - **Latent Dirichlet Allocation**
 - **rappresentazioni distribuite**

In particolare nella **rappresentazione N-gram** si cerca di determinare la probabilità $P(s = w_1, \dots, w_k)$ di una frase per capire qual è il modello sottostante di un certo documento. Questo viene ottenuto tramite il calcolo della probabilità congiunta

$$P(s_k) = \prod_i^k P(w_i | w_1, \dots, w_{i-1}) \quad (5.42)$$

Dal momento che nei modelli di lingua tradizionali n-gram "la probabilità di una parola dipende solo dal contesto delle $n-1$ parole precedenti", la probabilità diventa

$$\hat{P}(s_k) = \prod_i^k P(w_i | w_{i-n+1}, \dots, w_{i-1}) \quad (5.43)$$

Un processo tipico di apprendimento *ML-smoothing* effettua una levigazione della probabilità per evitare di avere probabilità pari a 0. Per fare ciò calcola la probabilità

$$\hat{P}(w_i | w_{i-n+1}, \dots, w_{i-1}) = \frac{\#w_{i-n+1}, \dots, w_{i-1}, w_i}{\#w_{i-n+1}, \dots, w_{i-1}} \quad (5.44)$$

Neural Net Language Model Si definisce **embedding** una tecnica che mappa una parola (o frase) dal suo spazio di input alto-dimensionale (il corpo di tutte le parole) a uno spazio vettoriale numerico di dimensione inferiore.

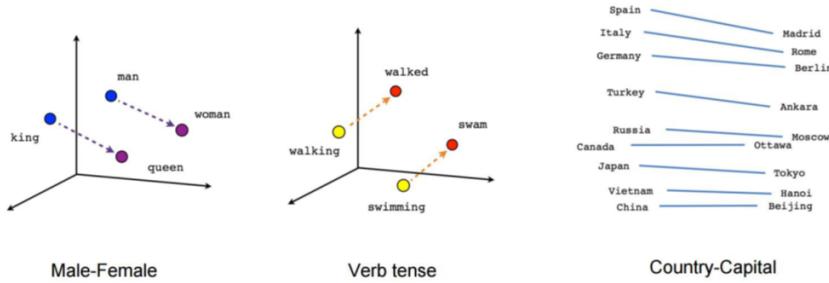


Figura 5.23: Esempi di embedding

Ogni parola unica w in un vocabolario V (con dimensione tipicamente nell'ordine di 10^6) viene quindi mappata in uno spazio m -dimensionale continuo (con dimensione tipicamente compresa nel range $100 < m < 500$). Un modello tipico per il *word embedding* è il **Neural Net Language Model**. Per ogni sequenza di training, si ha che l'input è costituito da una coppia \langle contesto, target \rangle fatta così: $\langle w_{t-n+1} \dots w_{t-1}, w_t \rangle$. L'obiettivo, inoltre, è quello di minimizzare la funzione di errore

$$E = -\log \hat{P}(w_t | w_{t-n+1} \dots w_{t-1}) \quad (5.45)$$

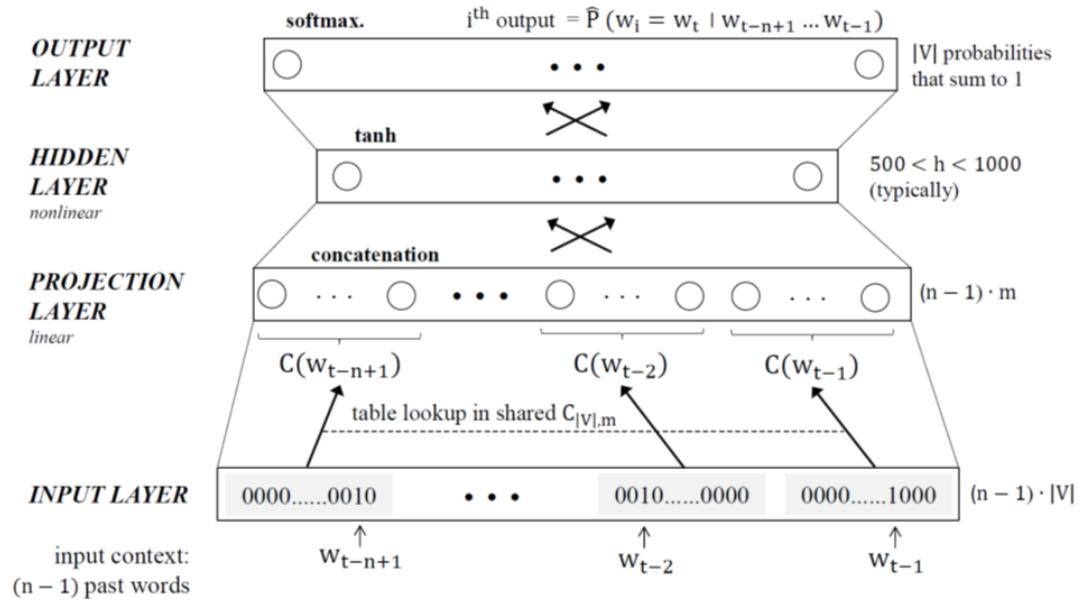


Figura 5.24: Neural Net Language Model

Il **layer di proiezione** contiene i vettori in $C_{|V|,m}$. La softmax è usata per produrre in output una **distribuzione multinomiale** con probabilità

$$\hat{P}(w_i = w_t | w_{t-n+1} \dots w_{t-1}) = \frac{e^{y_{w_i}}}{\sum_{i'} e^{y_{w_{i'}}}} \quad (5.46)$$

dove:

- $y = b + U \cdot \tanh(d + Hx)$
- x è la concatenazione $C(w)$ del contesto dei vettori dei pesi
- d e b sono dei bias, rispettivamente degli elementi h e $|V|$
- U è la matrice di dimensione $|V| \times h$ dei pesi tra il layer nascosto e il layer di output
- H è la matrice di dimensione $h \times (n-1) \cdot m$ dei pesi tra il layer di proiezione e il layer nascosto

Word2vec di Google L'idea è quella di ottenere una performance che consente il training di un modello meno profondo su una quantità di dati più grande. In questo modello non ci sono layer nascosti, il layer di proiezione è condiviso (non solo la matrice dei pesi) e il contesto contiene le parole provenienti sia dalla storia che dal futuro.

Le due architetture possibili sono l'**architettura skip-gram** e l'**architettura bag-of-words continua** (vedi figura sotto).

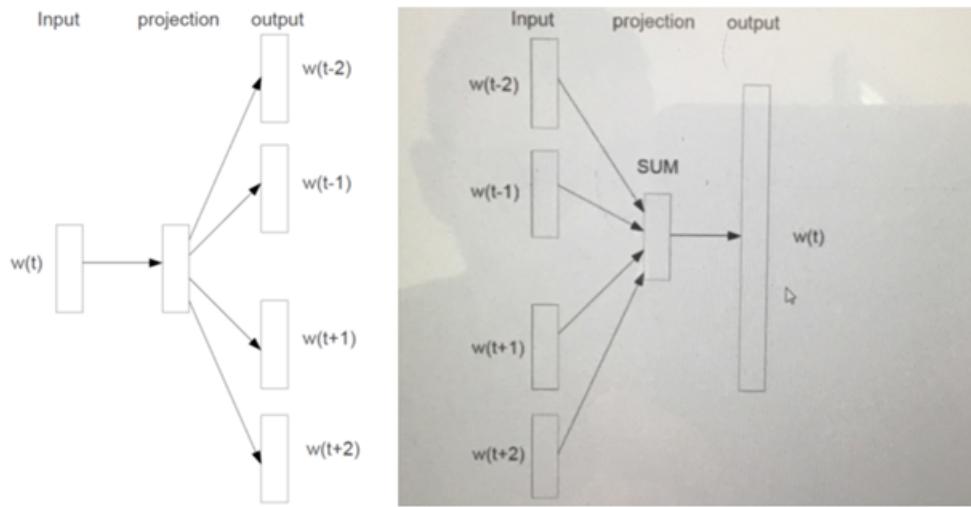


Figura 5.25: Architettura skip-gram (a sinistra) e architettura bag-of-words (a destra)

In particolare, nell'architettura bag-of-words continua (**CBOW**), per ogni sequenza di training, si ha che l'input è costituito da una coppia <contesto, target> fatta così: $\langle w_{t-n+1} \dots w_{t-1} w_{t+1} \dots w_{t+\frac{n}{2}}, w_t \rangle$. L'obiettivo, inoltre, è quello di minimizzare la funzione di errore

$$E = -\log \hat{P}(w_t | w_{t-\frac{n}{2}} \dots w_{t-1} w_{t+1} \dots w_{t+\frac{n}{2}}) \quad (5.47)$$

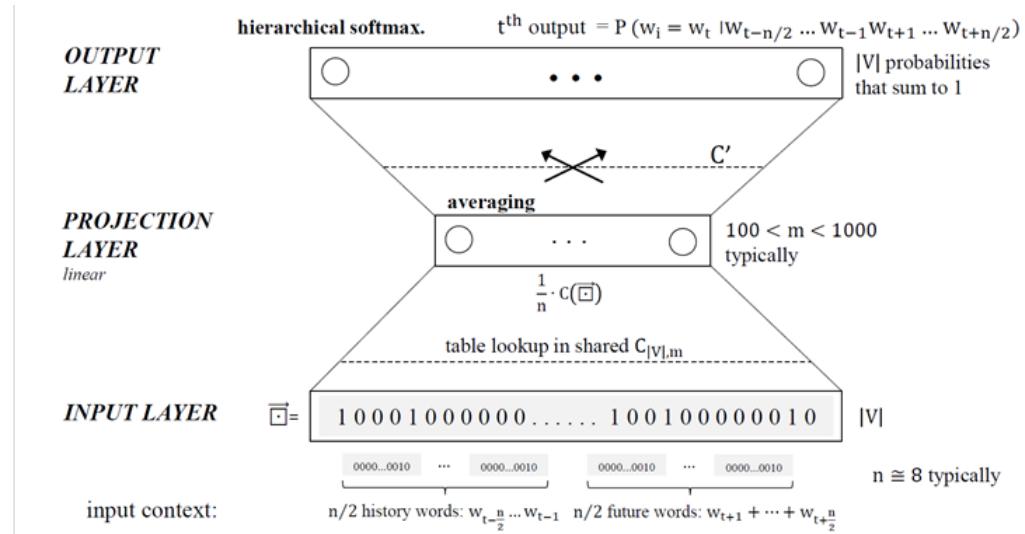


Figura 5.26: CBOW

In pratica, per ogni coppia $\langle \text{contesto}, \text{target} \rangle$ solo le parole di contesto vengono aggiornate. Se la probabilità $\hat{P}(w_i = w_t | \text{context})$ è sovrastimata, una certa porzione di $C'(w_i)$ viene sottratta dai vettori delle parole contestuali in $C_{|V|,m}$; se invece è sottostimata, la porzione viene aggiunta ai vettori delle parole contestuali in $C_{|V|,m}$. Applicazioni di **word2vec** riguardano la classificazione di documenti e l'analisi sentimentale.

GloVe **GloVe** fa esplicitamente ciò che word2vec fa implicitamente, ovvero codifica il significato come **vettore di offset** in uno spazio di embedding. Tale significato viene codificato da dei rapporti di probabilità di co-occorrenza. L'allenamento avviene con una *least squares pesata*

$$J = \sum_{i,j=1}^V f(X_{ij}) \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij} \right)^2 \quad (5.48)$$

In pratica questo metodo si basa sull'individuazione dei **nearest neighbours** di una certa parola.

Capitolo 6

Autoencoder

6.1 Struttura

Un' **autoencoder** è un tipo di rete neurale che nella sua formulazione più semplice cerca di apprendere la **funzione identità**. In altre parole impara a produrre una copia dell'input.

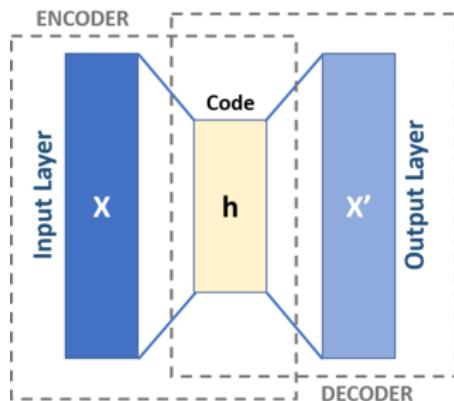


Figura 6.1: Schema generale di un autoencoder

È composto principalmente da due parti.

1. **Encoder**, la cui funzione è mappare l'input in una rappresentazione latente di dimensione minore ed eventualmente con altri vincoli.
2. **Decoder** che mappa la rappresentazione latente nell'output.

Formalmente possiamo descrivere un autoencoder attraverso tre funzioni matematiche:

$$\phi : \mathcal{X} \rightarrow \mathcal{F}$$

$$\psi : \mathcal{F} \rightarrow \mathcal{X}$$

$$\phi, \psi = \arg \min_{\phi, \psi} \|X - (\psi \circ \phi)X\|^2 \quad (6.1)$$

$X \in \mathbb{R}^d$ e $F \in \mathbb{R}^n$ con $n \ll d$. Le features F vengono solitamente chiamate **rappresentazione latente**. Lo schema di un modello di autoencoder è abbastanza generale ed è quindi implementabile tramite varie architetture come *feedforward*, *convoluzionali*, *ricorrenti* e con varie profondità della rete, andando incontro al ben noto *trade-off* tra capacità di approssimazione e sovraccarico.

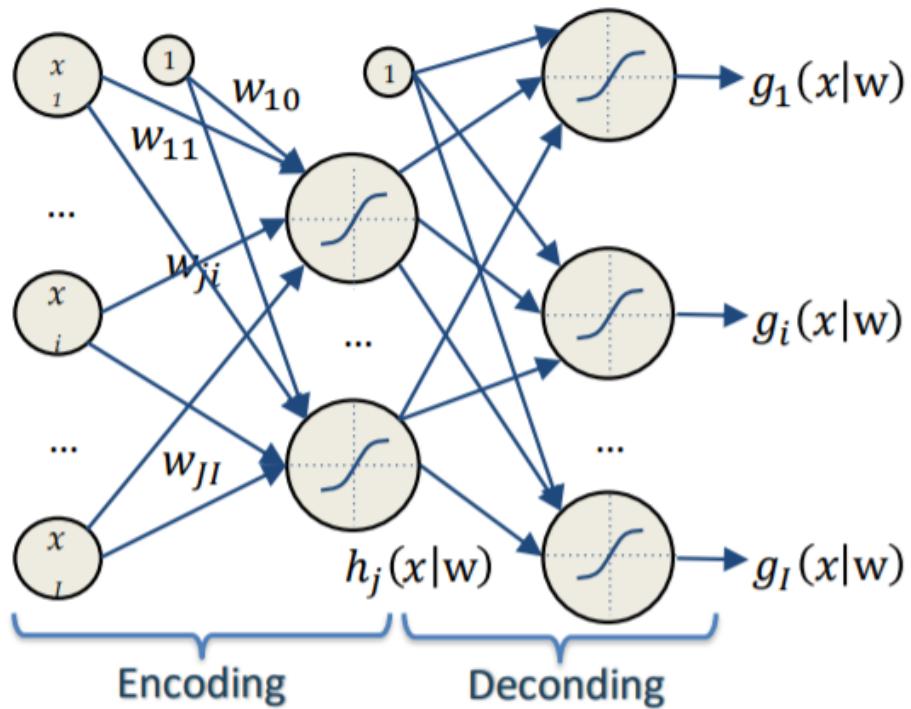


Figura 6.2: Autoencoder feedforward

6.2 Utilizzo delle reti autoencoder

6.2.1 Riduzione dimensionale

Una rete autoencoder è usata principalmente come algoritmo di **riduzione dimensionale** (o compressione) con determinate proprietà:

- **Data specificità:** l'autoencoder è adatta a comprimere solo dati simili a quelli su cui è allenata, o addirittura solo questi ultimi. Apprende features specifiche, quindi un'autoencoder allenata a comprimere immagini di caratteri scritti a mano non sarà adatta a comprimere foto generiche.
- **Lossy:** l'output non sarà esattamente identico all'input, si ha quindi una degradazione.
- **Self-supervised.**

Una caratteristica spesso cercata nella rappresentazione latente è la **sparsità**. Questa proprietà può essere raggiunta aggiungendo un termine di penalità proporzionale alla rappresentazione latente. Tipicamente viene utilizzata la norma L1. Riprendendo la notazione formale precedente la loss function diventa:

$$L = \|X - (\psi \circ \phi)X\|^2 + \lambda \sum_i |f_i| \quad (6.2)$$

Minimizzando il secondo termine imponiamo che la rappresentazione latente abbia la somma dei valori assoluti delle singole componenti minima ma contemporaneamente minimizzando il primo termine imponiamo che la rappresentazione latente sia in grado di separare significativamente le features. Idealmente, se non considerassimo la dimensionalità, una rappresentazione efficace a minimizzare questa loss function sarebbe la *one-hot encode*. Questa considerazione suggerisce che la rete tenderà comunque ad una rappresentazione in cui la maggior parte delle componenti f_i siano tendenti a zero e solo il minimo necessario di componenti siano con modulo significativamente diversi da zero.

6.2.2 Eliminazione del rumore

È possibile usare le reti autoencoder per rimuovere e/o attenuare il rumore nei dati. Per far ciò, durante il training, ai dati di input viene aggiunto del **rumore** e la rete tenterà di ricostruire l'input originale. Oltre a poter utilizzare l'autoencoder per rimuovere il rumore dai dati, aggiungere del rumore in input permette alla rete di apprendere una **rappresentazione latente** più robusta, in quanto è forzata ad estrarre features idealmente indipendenti dalle fluttuazioni dovute al rumore. Questo può migliorare le prestazioni anche negli altri casi d'uso.

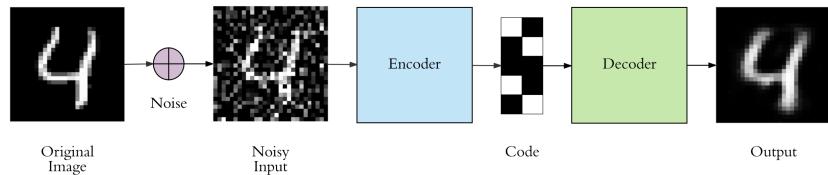


Figura 6.3: Schema denoising autoencoder

6.2.3 Inizializzatore di pesi

Gli autoencoder possono essere utilizzate per **inizializzare** reti neurali con altri compiti, come per esempio un classificatore. In particolare quando si hanno pochi dati già classificati per il training. Per far questo si procede in questo modo:

1. Si progetta la rete per classificare normalmente. Ricordando la struttura delle reti CNN (senza perdere generalità) per la classificazione ricordiamo che essa è strutturata in due parti. La prima parte è incaricata di estrarre le features mentre la seconda usa le features per classificare effettivamente. Sostituiamo la seconda parte della rete con il simmetrico della prima parte. In questo modo abbiamo una struttura che produce un output della stessa dimensione dell'input. Di fatto abbiamo ottenuto una rete autoencoder. Alleniamo l'autoencoder ottenuto con i dati privi di etichette.
2. Scartiamo il decoder e manteniamo l'encoder con i pesi ottenuti dall'allenamento.
3. Connettiamo la parte precedentemente sostituita dal decoder (classificatore).
4. Usiamo la tecnica del fine tuning per completare l'allenamento con i pochi dati etichettati.

Le reti autoencoder forniscono una buona inizializzazione, riducendo anche il rischio di overfitting, perché dovendo approssimare la funzione identità apprendono una rappresentazione latente su dati simili ma non uguali a quelli effettivamente usati nel training.

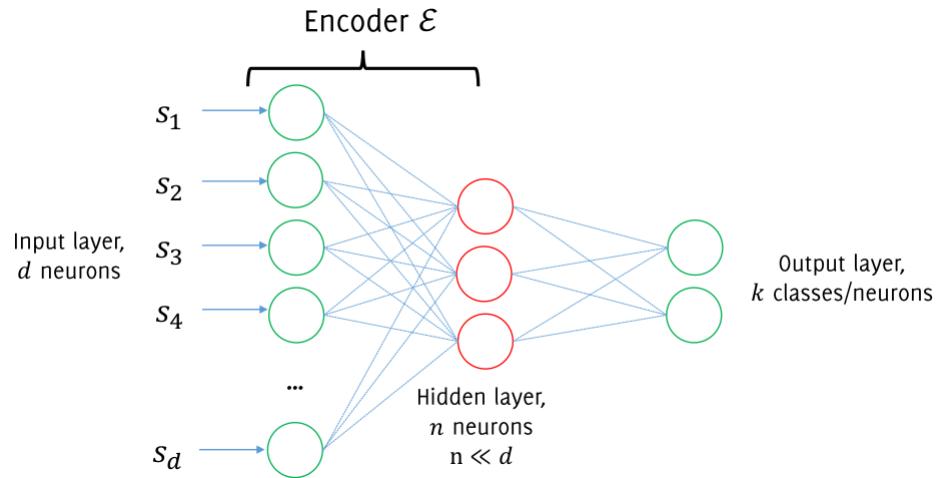


Figura 6.4: Classificatore inizializzato con autoencoder

6.2.4 Autoencoder generativi

I **Generative Models** o modelli generativi permettono di creare dati come foto, film, musica, testi e usarli negli ambiti più diversi come **data augmentation**, simulazioni e pianificazioni. È possibile tecnicamente usare autoencoder per generare nuovi dati, ma non senza qualche criticità. Un'opzione per generare nuovi dati potrebbe essere la seguente:

1. Alleniamo l'autoencoder su un insieme di dati X in modo da costruire una rappresentazione latente F .
2. Scartiamo l'encoder
3. Generiamo dei vettori casuali nello spazio F e li usiamo come input del decoder che produrrà in output un nuovo dato.

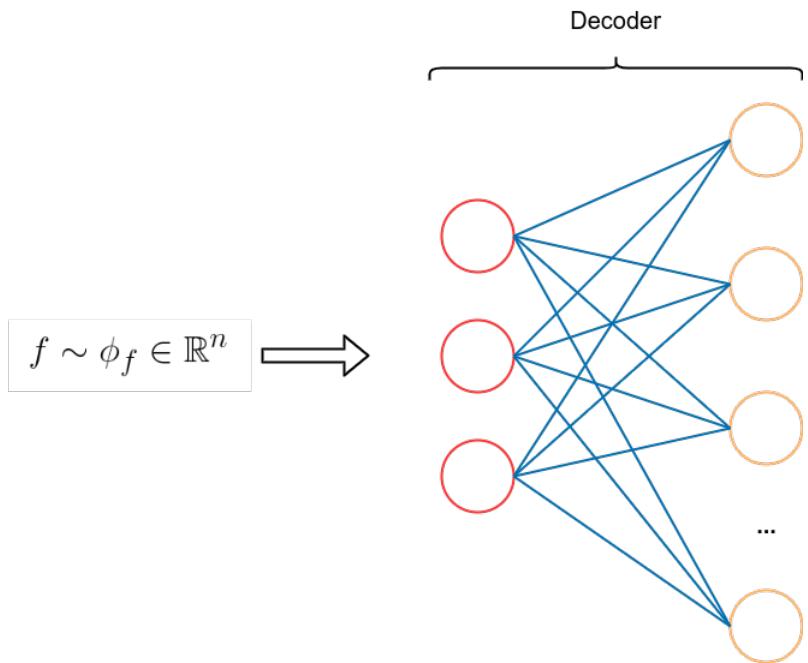


Figura 6.5: Generazione con autoencoder

Mentre i punti 1 e 2 non creano eccessivi problemi, il punto 3 è critico. Non sappiamo nulla a priori della distribuzione che assume lo spazio latente F quindi non siamo in grado di generare con alta probabilità dei vettori di input per il decoder che producano un output sensato.

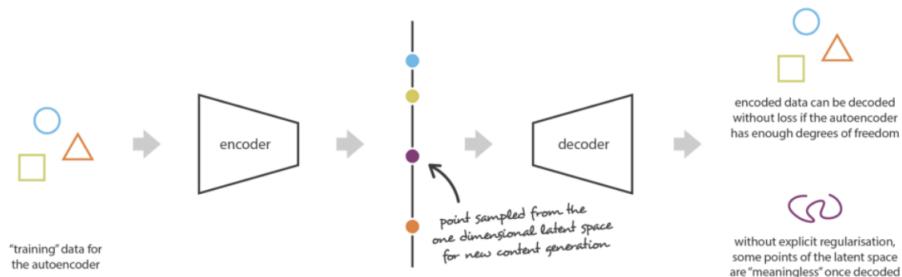


Figura 6.6: Spazio latente irregolare

Osservando l'esempio qui sopra vediamo una rete autoencoder allenata a rappresentare figure geometriche chiuse. Nel momento in cui diamo in input

un vettore casuale (punto viola) notiamo che il decoder genera una figura senza significato per il contesto. Questo avviene perché non campioniamo con la funzione di probabilità costruita dalla rete. Questa distribuzione (ϕ_f) è estremamente difficile da stimare a priori e devono essere applicate tecniche sofisticate come i **variational autoencoder**.

Bibliografia

- [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.
- [2] Slides delle lezioni
- [3] Appunti di PoliMI Data Scientists del corso di Soft Computing