ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

ELABORAZIONE DEL LINGUAGGIO NATURALE CON METODI PROBABILISTICI E RETI NEURALI

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Presentata da
Michele Pio Prencipe

Seconda Sessione di Laurea
Anno Accademico 2020 – 2021

PAROLE CHIAVE

Natural Language Processing

Deep Learning

Recurrent Neural Networks

Word Embedding

Language Model

Dedico questa mia tesi a chiunque mi sia stato vicino,
alle persone che hanno sempre creduto in me
e a tutti coloro che mi hanno aiutato nel raggiungimento di questo traguardo.

Sommario

L'elaborazione del linguaggio naturale (NLP) è il processo per il quale la macchina tenta di imparare le informazioni del parlato o dello scritto tipico dell'essere umano. La procedura è resa particolarmente complessa dalle numerose ambiguità tipiche della lingua o del testo: ironia, metafore, errori ortografici e così via.

Grazie all'apprendimento profondo, il Deep Learning, che ha permesso lo sviluppo delle reti neurali, si è raggiunto lo stato dell'arte nell'ambito NLP, tramite l'introduzione di architetture quali Encoder-Decoder, Transformers o meccanismi di attenzione. Le reti neurali, in particolare quelle con memoria o ricorrenti, si prestano molto bene ai task di NLP, per via della loro capacità di apprendere da una grande mole di dati a disposizione, ma anche perché riescono a concentrarsi particolarmente bene sul contesto di ciascuna parola in input o sulla *sentiment analysis* di una frase.

In questo elaborato vengono analizzate le principali tecniche per fare apprendere il linguaggio naturale al calcolatore elettronico; il tutto viene descritto con esempi e parti di codice Python. Per avere una visione completa sull'ambito, si prende come riferimento il libro di testo "Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow" di Aurélien Géron, oltre che alla bibliografia correlata.

Introduzione

Oggi l'**Intelligenza Artificiale** (IA) fa sempre più parte delle nostre abitudini quotidiane. Abbiamo avuto qualche tempo per abituarci a questa tecnologia, probabilmente anche grazie al fatto che da ormai molto tempo l'IA ha scatenato la fantasia degli scrittori di fantascienza che, con incredibile visione premonitoria, hanno immaginato scenari diversi, alcuni positivi e felici, mentre altri distopici e da incubo.

La necessità di studiare e implementare tecniche di intelligenza artificiale risiede intrinsecamente nella natura umana, in quanto l'uomo resta affascinato dalla possibilità di creare macchine in grado di simulare il proprio cervello.

Al giorno d'oggi i sistemi intelligenti sono presenti in ogni campo: nella vita di tutti giorni si possono trovare moderni termostati per il riscaldamento o arie condizionate in grado di anticipare il cambio di temperatura, gestire i bisogni degli abitanti e di interagire con altri dispositivi; oppure, nell'ambito ludico e videoludico i.e. la macchina (Deep Blue) che è riuscita a battere il campione del mondo di scacchi Garry Kasparov; fino ad arrivare alle missioni più ardue come i viaggi spaziali in cui la NASA ha utilizzato un programma chiamato Remote Agent in grado di gestire dalla terra le attività relative a un sistema nello spazio; o ancora, in ambito economico dove vengono sviluppati sistemi in grado di prevedere l'andamento degli indici di borsa. Questi sono solo alcuni dell'infinità degli esempi in cui hanno preso piede le tecniche di intelligenza artificiale. Pertanto, risulta essere di fondamentale importanza capirne a fondo i principi e le teorie che stanno dietro una tecnologia di questo genere.

Scendendo più nel dettaglio, l'intelligenza artificiale viene definita come l'abilità di una macchina di mostrare capacità tipiche dell'intelletto umano, doti uniche e inimitabili del nostro cervello dovute a migliaia di anni di evoluzione. Il ragionamento, l'apprendimento, la pianificazione e la creatività, la capacità di provare emozioni sono solo alcune delle capacità dell'Homo Sapiens che possono differenziarlo dall'automa.

Nel 1950 fu ideato il cosiddetto test di Turing, dal nome del suo inventore Alan Turing, con il quale si cercò di valutare la capacità di una macchina di eguagliare le caratteristiche dell'intelligenza umana. Il test può riguardare molti aspetti dell'intelletto umano come: il riconoscimento di oggetti nelle immagini,

il senso di orientamento in uno spazio, ma senz'altro la caratteristica più sensazionale è la padronanza del linguaggio. In questa tesi verranno utilizzati i principi del **Deep Learning**, un insieme di tecniche basate su **Reti Neurali**, per scoprire come un computer apprende e impara le caratteristiche del parlato o dello scritto umano. Successivamente, verranno analizzate le caratteristiche del **Natural Language Processing** (NLP) e i principali **modelli del linguaggio**, sia in maniera teorica che pratica, attraverso definizioni ed esempi di codice in **Python**.

La scelta di questo linguaggio risulta essere particolarmente pertinente, in quanto è uno dei più intuitivi e semplice da utilizzare, particolarmente adatto nell'ambito della *Computer Science* per via delle sue librerie ottimizzate nella manipolazione di grandi quantità di dati e semplice costruzione di modelli di apprendimento.

Indice

Elenco delle figure

Capitolo 1

Intelligenza Artificiale, Machine Learning, Deep Learning e Transfer Learning

In questo primo capitolo viene approfondito il contesto in cui è inserito lo studio della tesi, focalizzandosi sulla definizione di Intelligenza Artificiale con i suoi rami del *Machine Learning* e *Deep Learning*,analizzando gli step evolutivi per raggiungere lo stato dell'arte nel settore, per poi arrivare a toccare con mano anche l'argomento del *Transfer Learning*.

1.1 Intelligenza Artificiale

1.1.1 Definizione

Per Intelligenza Artificiale si intende la capacità di una macchina di agire e pensare umanamente e razionalmente. Questa disciplina è un ambito molto dibattuto sia tra scienziati che tra filosofi, in quanto manifesta aspetti etici, oltre che teorici e pratici. Le grandi menti mondiali hanno menzionato a più riprese, nei loro interventi, i pericoli di un'intelligenza artificiale mal gestita: Stephen Hawking, nel 2014, la considerò una minaccia per la sopravvivenza umana; Elon Musk, nello stesso anno, la definì più pericolosa del nucleare. Tuttavia, i vantaggi di un utilizzo consapevole dell'intelligenza artificiale come ambito di sviluppo e di progresso sembrano aver superato i rischi, tant'è che al giorno d'oggi è diventata parte del quotidiano.

1.1.2 Storia

Si può dire che l'AI nasce con l'avvento dei computer, nella seconda metà degli '50. Il primo programma di un sistema intelligente riguardava solo la dimostrazione di alcuni teoremi matematici partendo da determinate informazioni. Successivamente, molte università e aziende americane come l'IBM si cimentarono nello sviluppo di programmi e software in grado di pensare come gli esseri umani.

Col passare degli anni vennero sviluppati software sempre più complicati dal punto di vista matematico, ma contrariamente a quello che si sperava, l'intelligenza artificiale sembrava non riuscire a riprodurre le caratteristiche intellettuali umane. Il progresso definitivo si ebbe con l'avvento delle reti neurali, che hanno permesso ai sistemi intelligenti di migliorare sempre di più le proprie capacità di comportamento. Ora, questi sistemi sono in grado di prendere decisioni senza l'intervento umano ed effettuare scelte a seconda del contesto in cui sono inseriti.

1.2 Machine Learning

1.2.1 Definizione

Il Machine Learning è un insieme di metodi utilizzati per fare previsioni sulla base di una certa quantità di dati. Per fare ciò vengono usati algoritmi di regressione o di classificazione per comprendere i dati a disposizione. Questa disciplina viene chiamata anche apprendimento automatico in quanto, il modello dovrebbe imparare sulla base della propria esperienza, senza l'inserimento di nuove istruzioni.

1.2.2 Le modalità dell'apprendimento

L'apprendimento può essere effettuato in 4 diverse modalità:

- **Apprendimento supervisionato**: vengono presentati al modello una serie di esempi ideali costituiti dalla coppia input-output, in modo che riesca a capire la correlazione tra le entrate e le uscite.

- **Apprendimento non supervisionato**: al contrario dell'apprendimento precedente il modello riceve solo gli input. Deve capire da solo l'output da generare senza potersi confrontare con gli esempi.

- **Apprendimento per rinforzo**: l'obiettivo di questo tipo di apprendimento è quello di creare modelli in grado di fare azioni. Al programma

viene fornito un feedback per ogni azione che svolge. Un feedback positivo che si tradurrà in una ricompensa indica un'azione svolta correttamente, al contrario una punizione indicherà un'azione sbagliata.

- **Apprendimento semi-supervisionato**: vengono fornite informazioni incomplete sotto forma di esempi come nell'apprendimento supervisionato e il modello cercherà di prevedere anche quali sono i risultati mancanti.

1.2.3 Gli algoritmi

Un'altra caratteristica sulla quale si può classificare il Machine Learning è l'algoritmo utilizzato per l'addestramento. Gli algoritmi, in questo ambito, di certo non mancano. Partono tutti dal concetto di **Data Mining** (estrazione dai dati), con il quale si cerca di ricavare informazioni importanti dai dati a disposizione:

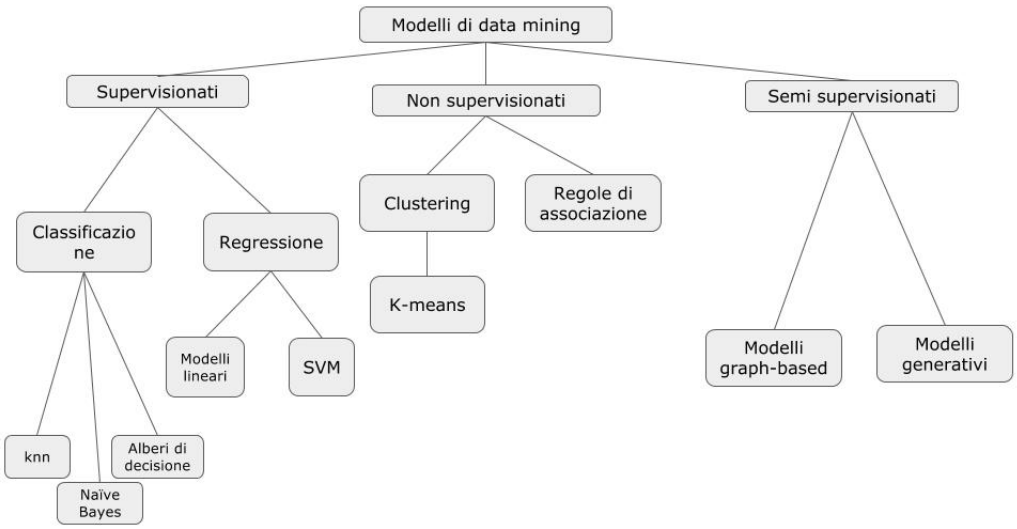

Figura 1.1: Schema riassuntivo degli algoritmi di machine learning

Ecco le principali famiglie di algoritmi:

- **Regressione**: questo algoritmo mira a stimare una funzione lineare o non-lineare in base ai dati in input. In questo modello esistono **variabili dipendenti** (o *target*) che variano in funzione delle **variabili indipendenti** (le *features*). Graficamente, l'algoritmo genera una funzione che cerca di approssimare nel modo migliore possibile i dati. Durante l'addestramento vengono aggiornati ad ogni iterazione i parametri della funzione da trovare.

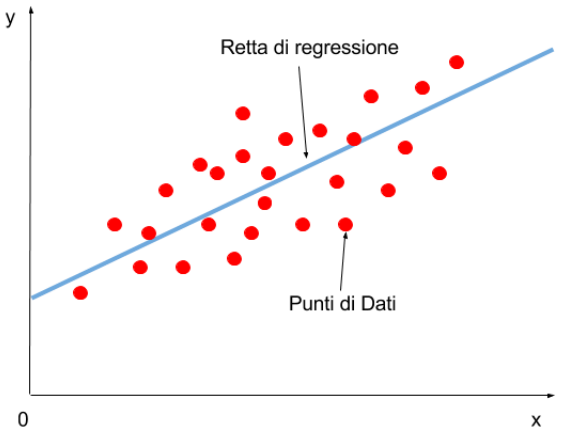

Figura 1.2: Regressione Lineare

- **Classificazione**: in questo caso, invece, si vogliono suddividere i dati di partenza in una serie di classi di appartenenza. Graficamente, classificare significa individuare una funzione (**iperpiano**) che massimizzi la separazione tra le classi.

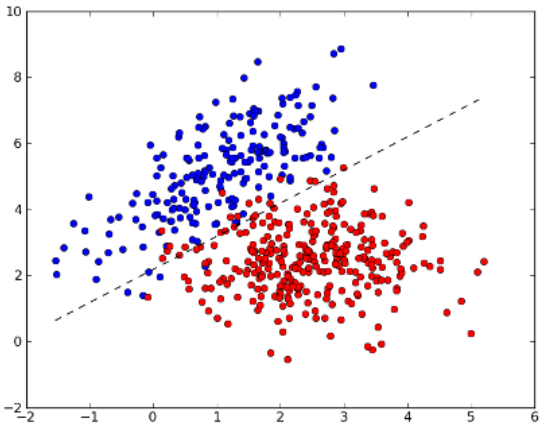

Figura 1.3: Classificazione

- **Clustering**: è un algoritmo che si basa sull'apprendimento non supervisionato che mira ad individuare raggruppamenti dei dati in base alle loro caratteristiche. A prima vista può sembrare simile ad un problema di classificazione, tuttavia presenta delle differenze sostanziali. Infatti, il clustering prova ad estrarre dalle istanze in input delle aeree di densità, mentre il compito della classificazione è quello di differenziare in varie classi le predizioni effettuate.

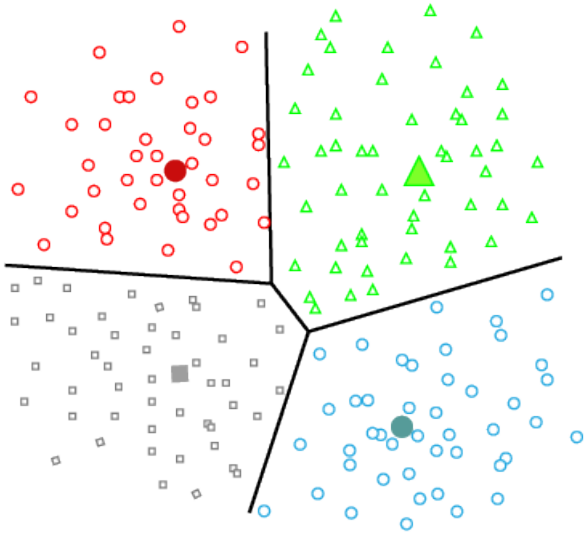

Figura 1.4: Esempio di clustering

1.2.4 La discesa del gradiente

Per valutare ed addestrare il modello, il primo passo è quello di calcolare la *cost function*, ovvero una funzione che rappresenti in qualche modo l'errore quadratico medio di tutti gli output (inteso come differenza tra output e valore atteso). La **discesa del gradiente** si pone come obiettivo quello di minimizzare quanto possibile la funzione di costo. Il problema risulta essere quello di trovare il minimo globale della funzione, ovvero il punto più basso. Il processo si basa su 4 step:

1. Si parte da un punto casuale A.
2. Si valuta la funzione ed il suo gradiente nel punto A (dove con gradiente si intende il vettore delle derivate parziali della funzione che indica la direzione di massima inclinazione in un punto).

3. Partendo da A viene sottratto un vettore, proporzionale al gradiente, per ottenere un nuovo punto.

4. Si esegue l'iterazione del punto 2, ripetendo fino ad una convergenza ad un minimo B.

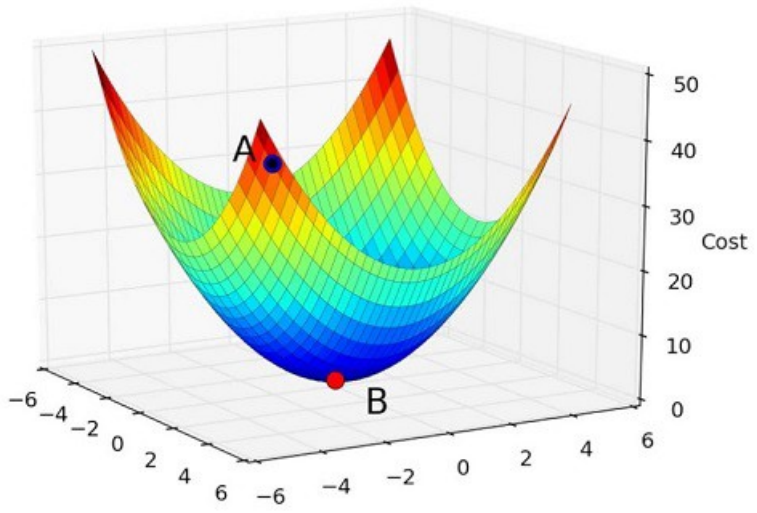

Figura 1.5: Metodo della discesa del gradiente

1.2.5 Limitazioni

Un problema tipico del Machine Learning è l'**overfitting** che può influenzare in maniera negativa un algoritmo di apprendimento. Il sistema va in overfitting se spende troppo tempo nell'addestramento e i dati non vengono generalizzati abbastanza bene, dove per **generalizzazione dei dati** si intende l'abilità di una macchina di portare a termine in maniera accurata esempi o compiti nuovi, che non ha mai affrontato, dopo aver fatto esperienza su un insieme di dati di apprendimento. Un altro problema, è quello inverso, ovvero l'**underfitting**. In questo caso il modello è troppo carente e non riesce ad addestrarsi bene sui dati presenti.

1.3 Deep Learning

1.3.1 Definizione

Il Deep Learning è un sottoinsieme del Machine Learning che si occupa di analizzare i dati in maniera profonda, solitamente attraverso una rete di apprendimento che prende decisioni e giunge a conclusioni in base ai dati forniti. Questo metodo è particolarmente adatto a grandi set di dati, in quanto molto più preciso, anche se molto più dispendioso in termini di risorse computazionali e temporali rispetto all'apprendimento automatico.

Il modello classico del Deep Learning è caratterizzato da gerarchie di caratteristiche comuni che mira ad andare in profondità nell'albero gerarchico tra i vari strati (per questo motivo viene detto "apprendimento profondo"). Tra le architetture di apprendimento profondo si annoverano le reti neurali profonde, la convoluzione di reti neurali profonde e le reti neurali ricorsive, le quali vengono applicate nella visione artificiale, nel riconoscimento automatico del discorso, nell'elaborazione del linguaggio naturale, nel riconoscimento audio e nella bioinformatica.

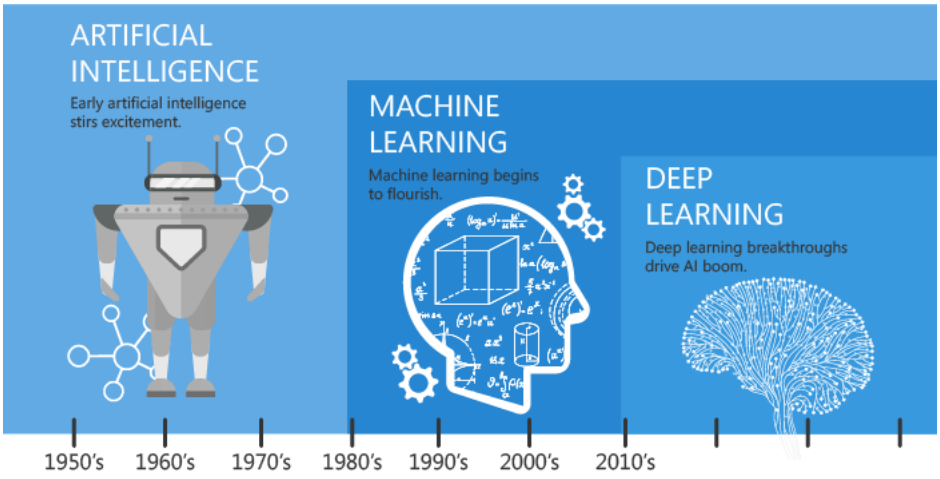

Figura 1.6: Evoluzione dei sistemi intelligenti

1.4 Transfer Learning

1.4.1 Definizione

Il Transfer Learning [1] è una tecnica di Machine Learning che focalizza il proprio apprendimento sulla memorizzazione di conoscenze già acquisite da

problemi diversi, ma di dominio simile i.e. nell'ambito della visione artificiale, le conoscenze apprese nel riconoscimento dei cani si possono sfruttare anche nel riconoscimento dei gatti. Questo tipo di algoritmo può risultare utile, quando si hanno dei dati non etichettati. Infatti, attraverso le conoscenze apprese da altri modelli riesce a sopperire al problema di insufficienza di dati etichettati. Come si può intuire dal nome, quindi, vi è una sorta di passaggio di conoscenza da un modello all'altro.

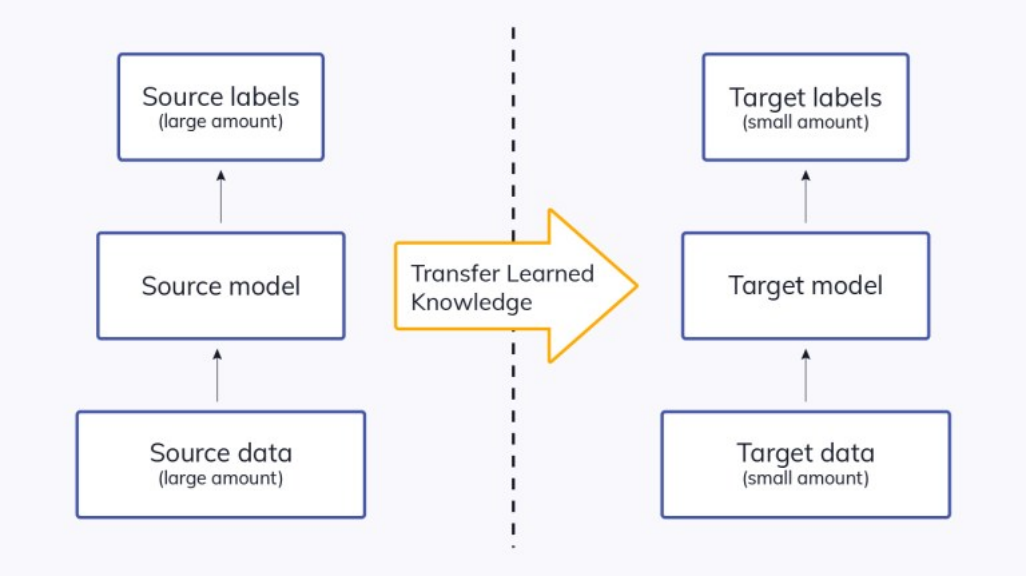

Figura 1.7: Transfer Learning

1.4.2 Applicazioni

Il modello è utile in quanto viene usato in svariate applicazioni molto diffuse. Tra le altre vengono citate:

- **Classificazione e riconoscimento delle immagini**: viene insegnato alla macchina a "vedere" cos'è rappresentato all'interno di un'immagine, attraverso reti convoluzionali.
- **Classificazione del testo tramite rappresentazioni vettoriali**: viene addestrata la macchina per capire la semantica del linguaggio umano. Le parole all'interno di un testo vengono rappresentate con una notazione vettoriale, in modo tale da poterne misurare la similarità ed effettuare classificazione o **sentiment analysis**. In questo caso il *training* richiede molto tempo, così sono nati modelli pre-addestrati come **GloVe** (*Global Vector for Word Reprenstation*), che sarà illustrato nel Capitolo 4.

Capitolo 2

Reti Neurali

In questo capitolo viene posta l'attenzione sulle caratteristiche dell'apprendimento profondo tramite reti neurali. In particolare, si partirà dallo studio di una rete neurale biologica, dalla quale hanno preso ispirazione le tecniche fondamentali di Deep Learning, fino ad arrivare ad architetture più complicate come **Transformer** ed **Encoder-Decoder**.

2.1 Reti Neurali Artificiali (ANN)

2.1.1 Rete Neurale Biologica

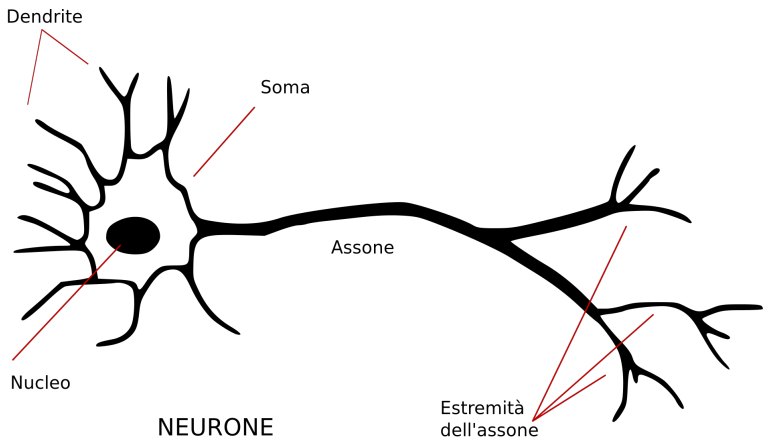

Figura 2.1: Esempio schematico di un neurone

Una rete neurale artificiale è la perfetta reincarnazione della rete di neuroni che compongono il cervello umano.

Un neurone è un'unità che si occupa di compiere operazioni sulla base di stimoli esterni ed è composto da 3 parti principali:

- il **soma**: è il corpo cellulare. Si occupa di integrare tra loro i vari input corrispondenti agli stimoli esterni.

- l'**assone**: l'unica linea di uscita del neurone che si dirama in migliaia di parti. Il soma restituisce un risultato che, nel caso in cui supera una certa soglia, il neurone si attiva e il cosiddetto "potenziale d'azione" (impulso elettrico) viene trasportato dall'assone. Al contrario, se il risultato non supera il valore di soglia il neurone rimane in uno stato di riposo.

- il **dendrite**: linea di entrata del neurone che riceve segnali in ingresso da altri assoni tramite le sinapsi, le quali a loro volta consentono la comunicazione con altri neuroni.

Questa unità sembra funzionare in maniera abbastanza semplice, tuttavia i neuroni sono organizzati in una rete di miliardi di singole componenti connesse tra loro. Infatti, essa si deve occupare di rispondere a tutti gli stimoli provenienti dall'esterno che costantemente interagiscono con il nostro corpo. Una volta elaborati gli stimoli la rete fornisce una risposta che si traduce in un'azione, un movimento di una parte del corpo, oppure un'idea finale che sta dietro un ragionamento.

2.1.2 Il Perceptron

McCulloch e Pitts proposero, nel 1943, un modello di un neurone artificiale che presentava uno o più ingressi binari ed un'unica uscita sempre binaria, in modo da essere compatibile con la logica booleana delle macchine.

Il neurone, in questo caso, si attiva quando entrambi gli input sono veri. Combinando tra loro i neuroni e attraverso le operazioni della logica booleana (AND, OR, NOT) si può realizzare una rete che riesce a calcolare espressioni logiche molto complesse. Le reti di McCulloch-Pitts furono inizialmente progettate per svolgere compiti aritmetico-logici specifici. Sostanzialmente, si possono definire come automi a stati finiti in grado di realizzare la logica delle proposizioni simulando il comportamento dei meccanismi cerebrali.

Nel 1958 nasce il modello **Perceptron** [2], considerato una delle prime architetture di reti neurali artificiali che riprende la logica utilizzata nel modello di McCulloch-Pitts, ma leggermente più complesso.

Gli ingressi e le uscite sono valori numerici e non più binari ed ogni input è associato ad un peso. La singola unità, chiamata unità logica di soglia (TLU), è costituita da una serie di variabili attivatrici $x_1, ..., x_n$ e da una serie di variabili inibitorie dette bias $b_1, ..., b_n$. É possibile assegnare agli input i pesi $w_1, ..., w_n$ per ponderare le variabili attivatrici, dando peso maggiore a quelle con più

rilevanza e viceversa. Il valore della soglia di attivazione viene chiamato T. L'output viene calcolato come somma pesata dei suoi input:

$$z = w_1x_1 + w_2x_2 + ... + w_nx_n \tag{2.1}$$

oppure, nel caso in cui viene considerato il vettore degli input $X = (x_1, ..., x_n)$ e il vettore dei pesi $W = (w_1, ..., w_n)$:

$$z = X^TW \tag{2.2}$$

La funzione di attivazione, detta **Heaviside**, è a scalino in quanto separa linearmente gli insiemi di ingresso e suddivide lo spazio in due partizioni. La funzione assume i seguenti valori:

$$heaviside(z) = \begin{cases} 1, z >= T \\ 0, z < T \end{cases} \tag{2.3}$$

Una singola unità può essere utilizzata ad esempio per la classificazione binaria lineare: nel caso in cui il risultato considerato come combinazione lineare degli input superi una certa soglia, viene considerato della classe positiva o viceversa.

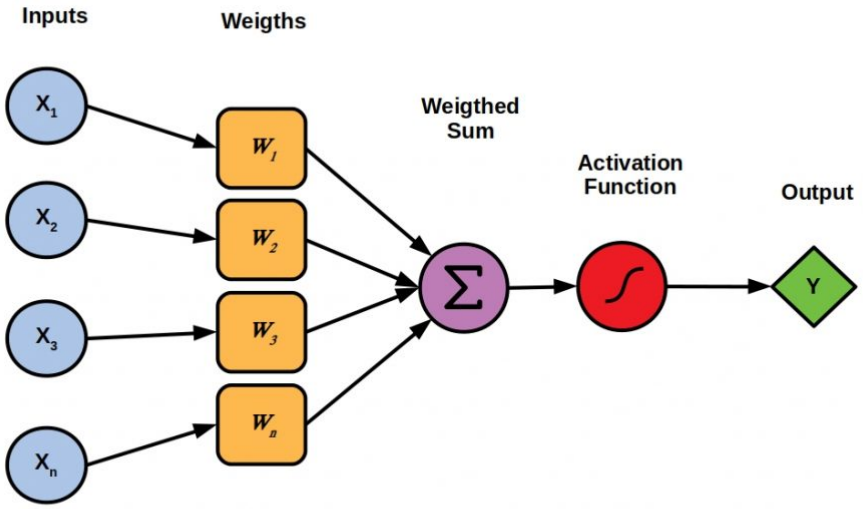

Figura 2.2: Struttura del Perceptron

2.1.3 Costruire una rete neurale e addestrarla

Il Perceptron può essere considerato come il progenitore delle reti neurali, in quanto presenta struttura e caratteristiche simili alle **Artificial Neural Network** (ANN). Un ANN è formata da n neuroni ciascuno dei quali:

- riceve valori in input da altri neuroni o dall'esterno.
- calcola un valore di output che manderà ad altri neuroni o all'esterno. Quest'ultimo sarà il risultato finale della rete, proprio come avviene nel neurone umano.

La matematica che sta dietro alla rete è molto semplice: l'i-esimo output y_i viene generato da un j-esimo neurone della rete a partire da un corrispondente input x_i. L'output finale viene considerato come somma di degli output di ciascun neurone, alla quale viene applicata la funzione di attivazione σ (di solito *sigmoide* o *tanh*):

$$y_i = \sigma(\sum_i w_{j,i} x_i + b_j) \tag{2.4}$$

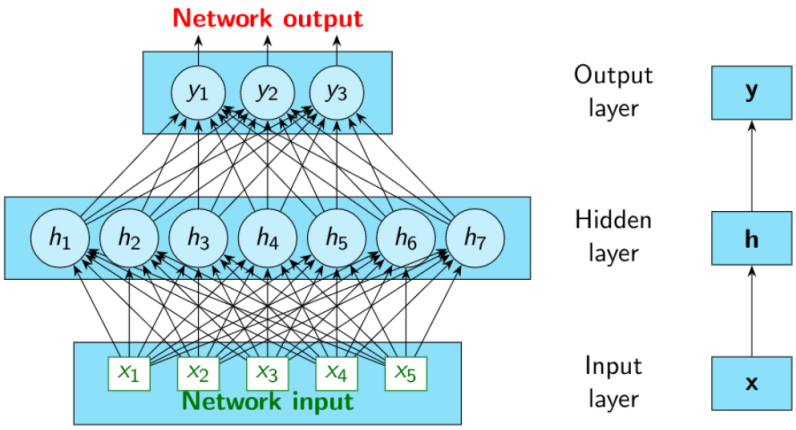

Figura 2.3: Struttura di una rete neurale con più layer

Tutti questi neuroni vengono organizzati in strati (*layer*). Se ogni nodo è connesso a tutti gli altri nodi di uno strato adiacente, allora quello strato viene detto *dense*. I *layer* vengono suddivisi in tre gruppi:

- **Input Layer**: è lo strato che riceve gli input dall'esterno.
- **Hidden Layer**: è lo strato in cui vengono elaborati i dati dello strato precedente.

- **Output Layer**: è lo strato contenente il risultato finale.

L'output di un intero *layer* è dato da:

$$y = \sigma(Wx + b) \tag{2.5}$$

dove W sono i pesi e b i bias, considerabili come parametri del *layer*. Sono questi parametri che vanno ad incidere il comportamento dell'intera rete. Una rete può arrivare ad avere milioni di parametri che vengono inizializzati in maniera random per l'addestramento.

Procedendo con l'addestramento i parametri vengono aggiustati fino a quando non raggiungono il comportamento desiderato. Si utilizza, quindi, il metodo della **discesa stocastica del gradiente** in combinazione con la ***backpropagation***. La discesa stocastica è un'estensione dell'algoritmo della discesa del gradiente, in cui, al posto di calcolare esattamente il gradiente per ogni iterazione, viene usato quello di uno dei campioni scelto casualmente. La retro propagazione (*backpropagation*) è un algoritmo per il calcolo del gradiente di una funzione di perdita rispetto alle variabili della rete: le informazioni partono dall'output finale per poi fluire all'indietro tra gli strati della rete, al fine di poter calcolare il gradiente per ogni strato. I parametri, nel corso delle iterazioni, vengono aggiornati nel seguente modo:

$$w \leftarrow w - \eta \frac{\delta(g_i - y_i)^2}{\delta w} \tag{2.6}$$

Le iterazioni proseguono finché non viene soddisfatto un certo criterio di terminazione oppure viene raggiunto il minimo della funzione di costo.

Durante l'addestramento è solito iterare sul **training set** (una suddivisione del dataset iniziale per permettere l'addestramento su una parte di dati) in più mandate dette **epoche**. Il costo ad ogni epoca viene decrementato fino a che non raggiunge un ottimo locale oppure non soddisfa i criteri di terminazione. In un set a parte, detto **validation set** (la restante parte del dataset non utilizzata per l'addestramento, impiegato per la valutazione del modello), si può valutare l'accuratezza della rete. Nelle reti ***feedforward***, come queste, i *layer* assumono una sequenza temporale lineare, dall'input all'output, cosa che non avviene per le reti neurali ricorrenti.

2.2 Reti Neurali Ricorrenti (RNN)

Le reti ricorrenti, così come si può intuire dal nome, hanno la stessa forma delle reti *feedforward*, tuttavia le loro connessioni sono cicliche in modo da mantenere in memoria lo stato tra le sequenze di input. In sostanza, viene

inserito un ciclo di *feedback* che consente all'output di essere reinserito come input nella sequenza temporale successiva. L'aggiunta del ciclo consente alla rete di elaborare, apprendere e prevedere dati sequenziali. Le reti *feedforward* tradizionali non possono farlo perché gli input e gli output sono vettori di dimensioni fisse, che si presume siano indipendenti l'uno dall'altro. I dati sequenziali, invece, possono variare in lunghezza e dimensioni ed i valori non sono indipendenti tra loro. Le attività che si possono svolgere con questi particolari tipi di dato possono essere molteplici: variano dal prevedere la parola successiva in una frase alla previsione del mercato azionario; dalla composizione musicale alle *chatbot*.

2.2.1 Struttura di una rete neurale ricorrente

La struttura di una **rete neurale ricorrente** (RNN) risulta essere simile alle reti *feedforward*, ma, come abbiamo detto in precedenza presenta delle connessioni cicliche per favorire la memorizzazione dell'output in ogni fase temporale.

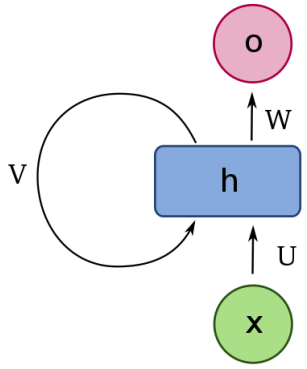

Figura 2.4: Struttura di un neurone ricorrente

Nell'immagine (Fig.2.4) è riportata una rete formata da un singolo neurone. Il neurone, così come nelle reti *feedforward*, riceve degli input X e restituisce output O. Come si può notare, l'uscita torna in parte o in toto all'ingresso del neurone stesso.

Ciò significa che ad ogni passo temporale t (chiamato *frame*), l'uscita dipende sia dalla funzione di attivazione sugli input sia in parte dal valore $V(T)$ che, per effetto del ciclo, corrisponde all'output $O(t-1)$, che a sua volta dipendeva sia da $X(t-1)$ che $V(t-1)$, il quale corrispondeva all'uscita $O(t-2)$ e così via. Perciò la rete può essere "srotolata" attraverso i vari *frame* temporali ed un semplice ciclo permette alla rete di avere memoria e mantenere lo stato

dell'output precedente. Proprio per questo motivo le reti ricorrenti hanno preso l'appellativo di **reti con memoria** [3] [4].

Lo stato viene resettato ad ogni nuova sequenza in input. Partendo dal singolo neurone ora è possibile creare un intero *hidden layer* di neuroni ricorrenti, dove ognuno riceve sia il vettore degli input $X(t)$ che il vettore degli output $O(t-1)$ dato dallo step temporale precedente. In questo modo l'output $O(T)$ di uno strato all'interno di una rete è dato da:

$$O(t) = \phi([X(t) \cdot O(t-1)] \cdot W + b) \text{ con } W = \begin{bmatrix} W_x \\ W_o \end{bmatrix} \tag{2.7}$$

dove:

- W_o è il vettore dei pesi dello strato di output.
- ϕ è una funzione di attivazione come la *ReLU* oppure la *tanh*.
- W_x è il vettore dei pesi per gli input.
- b è il bias.

2.2.2 Addestrare una rete neurale ricorrente

Una rete neurale ricorrente può essere addestrata tramite l'algoritmo della **Backpropagation Through Time (BPTT)**: il metodo è uguale a quello visto con la *backpropagation* delle reti *feedforward*, ma in questo caso l'errore commesso all'istante t dipende dall'errore all'istante *t-1* e così via. Per questo motivo per calcolare l'errore vengono ripercorsi all'indietro tutti gli archi che percorrono temporalmente gli step della rete.

L'output di ogni *layer* è valutato secondo una specifica funzione di costo $C(Y(0), Y(1), ..., Y(T))$ che è calcolata per ogni istante temporale partendo da 0 fino ad arrivare all'ultimo chiamato T. Si noti, inoltre, come i gradienti di tale funzione di costo vengono propagati all'indietro attraverso la rete srotolata. Infine, i parametri del modello vengono aggiornati utilizzando i gradienti calcolati durante la BPTT.

Il problema principale nell'addestramento di questo tipo di reti rimane la *dissolvenza\esplosione* del gradiente che risiede nelle funzioni di attivazione utilizzate:

- Se vengono utilizzate funzioni del tipo *sigmoid* o *tanh*, i valori dei loro gradienti restano compresi tra [0..1]. Visto che l'algoritmo di *backpropagation* prevede che i gradienti vengano moltiplicati lungo i *layer*, è chiaro che il prodotto di un elevato numero di valori compresi tra 0 ed 1 porta il valore complessivo a diminuire velocemente lungo la catena di neuroni (**dissolvenza del gradiente**).

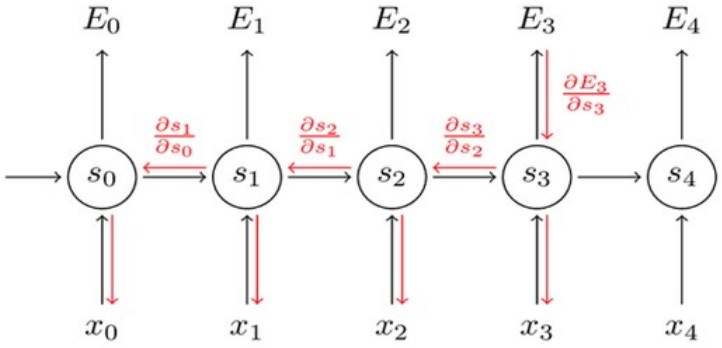

Figura 2.5: Funzionamento della BPTT

- Se, al contrario, vengono utilizzate funzioni lineari come la $ReLU$ o la lineare pura, i valori dei gradienti possono essere anche superiori ad 1 e quindi il valore complessivo può crescere enormemente lungo la catena di neuroni (**esplosione del gradiente**).

Esistono anche in questo caso varie soluzioni per sopperire ai problemi:

- **Gradient Clipping**: aggiornare il gradiente dentro un range prefissato.
- **Truncate Backpropagation**: fissare il gradiente dentro una finestra temporale specifica. Questa soluzione comporta la perdita dei riferimenti agli altri istanti temporali.
- **Batch Normalization** (BN): si può utilizzare solo nei *layer* e non tra passaggi temporali: è possibile aggiungere uno strato BN ad una cella di memoria, ma ciò da buoni risultati solo quando viene applicato tra gli strati ricorrenti e non all'interno di uno strato, come dimostrato in [5].
- **Layer Normalitazion** (LN): Simile alla BN, ma invece che normalizzare tra la dimensione delle batch, si normalizza tra la dimensione delle *features*. Come la BN, la *Layer Normalization* [6] apprende una scala e un parametro di offset per ogni input.

La propagazione dell'errore per ogni istante temporale equivale a fare moltiplicazione delle derivate parziali che rappresentano i vari archi. Tuttavia per addestrare una RNN con sequenze molto lunghe è necessario costruire una rete molto profonda, che può portare a tempi di addestramento troppo lunghi, oppure a gradienti instabili. Inoltre, man mano che si scende in profondità la rete comincia a dimenticarsi gradualmente i primi input della sequenza. Per

risolvere questi problemi vengono generalmente utilizzati alcuni "trick" comuni a quelli delle reti *feedforward*: ottimizzatori veloci, buona inizializzazione dei parametri, *dropout*.

2.2.3 Long Short Term Memory (LSTM)

Per ovviare al problema della perdita graduale dei primi input è stata introdotta la *Long Short Term Memory (LSTM)* [7] [8]. L'idea di questa tecnica può essere riassunta secondo i seguenti principi:

- Una RNN può avere più *layers* di celle LSTM, ognuna delle quali memorizza un valore in un vettore.
- il flusso di informazione tra ogni cella è gestito da particolari ponti detti *gates*, che evitano la lettura o la scrittura in memoria.
- lo stato ha un'impostazione predefinita, in questo modo viene evitata anche la *dissoluzione/esplosione* del gradiente.

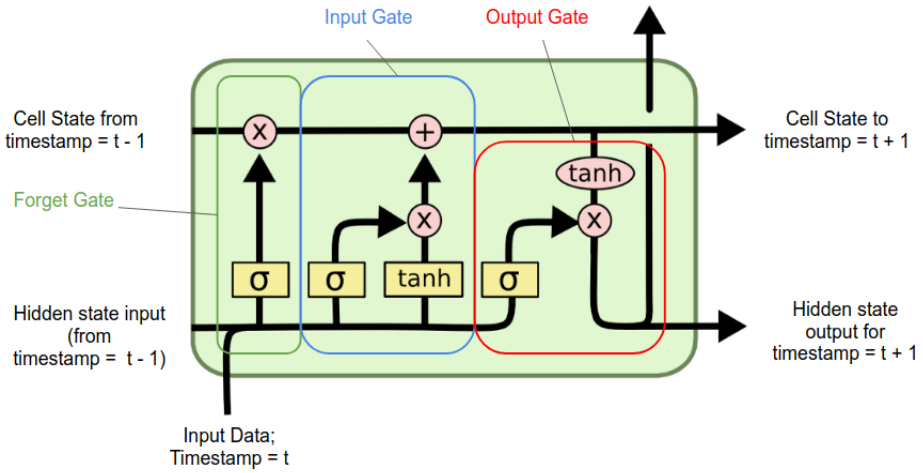

Figura 2.6: Struttura di una cella LSTM

Questa particolare cella impara a decidere cosa conservare nella memoria a lungo termine scartando i parametri inutili e leggendo dalla memoria quelli più rilevanti. La cella si comporta in maniera particolare a seconda dello stato in cui è (vedi Figura 2.6).

- **Memoria a lungo termine** da C_{t-1} (stato della cella all'istante di tempo precedente $t-1$) a C_t (stato da destinare all'istante successivo

$t+1$): attraversa la rete da sinistra a destra come nelle reti RNN, ma deve passare attraverso il *forget gate* dove perde alcuni ricordi inutilizzati e l'*input gate* dove acquista nuove informazioni. Una volta finito il processo il risultato viene direttamente passato a $C(t)$.

- **Passaggio da lungo a breve termine**: Durante il passaggio dall'*input gate* viene copiata l'informazione contenuta nella memoria a lungo termine e passata alla funzione *tanh*. Il risultato produce la memoria a breve termine $h(t)$.

- **Memoria a breve termine** da h_{t-1} (stato nascosto proveniente da $t-1$) a h_t (stato nascosto attuale da destinare a $t+1$): h_{t-1} viene inserito nei vari gate per prelevare le informazioni rilevanti dalla memoria a lungo termine. Il risultato $h(t)$ è uguale a $y(t)$ di questo neurone in questo arco temporale.

La cella al suo interno si compone di 3 gates:

- **Forget Gate**: controlla quale parte della memoria a lungo termine debba essere rimossa e quale lasciata. Per fare questo viene utilizzata la funzione $f(t)$:
$$f(t) = \sigma(x_t W_{xf^T} + h_{t-1} W_{hf^T} + b_f) \tag{2.8}$$
La funzione si basa sulla *sigmoide* che restituisce un vettore in cui ogni elemento vale:
  - 1 se l'informazione deve essere ripresentata all'istante corrente t.
  - 0 se l'informazione deve essere scordata.

- **Input Gate**: controlla quale parte di una certa proposta di informazione $\tilde{C}_t$ debba essere aggiunta allo stato di lungo termine:
$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C) \tag{2.9}$$
La nuova informazione si basa sull'apprendimento di un iperpiano con parametri W_C e b_C sui dati $[h_{t-1}, x_t]$. Il risultato è compreso tra $[-1, 1]$ per via della tangente iperbolica; in questo modo si evita il *dissolvimento/esplosione* del gradiente. Il vettore i_t normalizzato indica quali sono le informazioni giuste da utilizzare. Tale vettore, come prima, viene calcolato tramite la *sigmoide* per apprendere l'iperpiano dei pesi di $\tilde{C}_t$. Il calcolo di i_t viene svolto nel seguente modo:
$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \tag{2.10}$$
Quindi il nuovo stato C_t deriva da:
$$C_t = f_t \otimes C_{t-1} \oplus i_t \otimes \tilde{C}_t \tag{2.11}$$

- **Output Gate**: controlla quale parte della memoria a lungo termine deve essere letta e quindi diventare l'output $h(t)$ della cella nell'istante corrente. Il compito di questo gate si suddivide in 3 passi:

  1. Recupera un vettore o_t delle informazioni rilevanti dalla memoria a lungo termine, scartando quelle che potrebbero diventare importanti in futuro.
  2. l'output o_t appreso tramite la solita *sigmoide* sull'iperpiano dato da $W_o[h_{t-1}, x_t] + b_o$.
  3. l'output finale della cella y_t è dato dalla moltiplicazione con la tangente iperbolica delle informazioni della memoria a lungo termine:

$$y_t = h_t = o_t * tanh(C_t) \tag{2.12}$$

2.2.4 Varianti

In una normale cella LSTM, i *gate controller* si focalizzano solo sull'input $x(t)$ e sul precedente stato a breve termine h_{t-1}. Sembrerebbe una buona idea quella di fornire un contesto alla cella, permettendo di guardare anche nella memoria a lungo termine.

Da qui nasce la teoria dello spioncino (*peephole* [9]) che permette di "sbirciare" nello stato a lungo termine. Lo stato C_{t-1} viene aggiunto come dato in ingresso al *forget gate* e all'*input gate*, mentre lo stato C_t viene aggiunto come ingresso dell'*output gate*.

Un'altra variante dell'unità LSTM è la struttura *Gated Recurrent Unit* (GRU) [10], progenitrice dell'architettura Encoder-Decoder. GRU è una versione semplificata di LSTM che cerca di mantenerne i vantaggi, riducendo parametri e complessità. Le principali semplificazioni sono:

- le memorie vengono unite in un unico vettore h_t
- viene usato un *update gate* (unione di *input gate* e *forget gate*) e un *reset gate* (che controlla lo stato dell'output) per quantificare quanto dimenticare e quanto aggiungere: se necessario aggiungere prima deve dimenticare.

GRU in generale funziona meglio rispetto a LSTM. Quest'ultimo, infatti, può arrivare a richiedere grandi quantità di dati per migliorare la propria accuratezza.

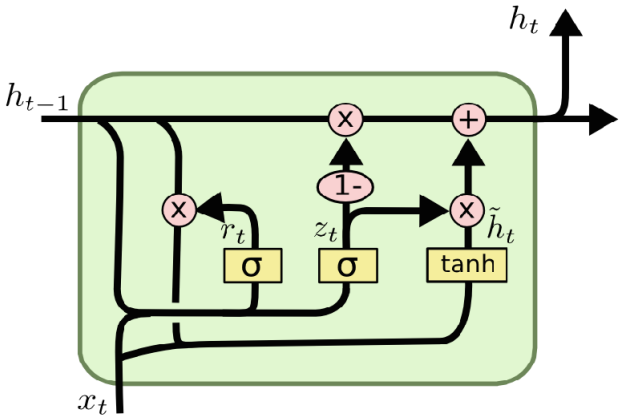

Figura 2.7: Struttura del *Gated Recurrent Unit* (GRU)

2.2.5 Usi e applicazioni

Le reti neurali ricorrenti possono essere usate in vari ambiti come:

- **Time Series Forecasting**: le serie temporali rappresentano banche dati storiche di eventi di varia natura. E' possibile addestrare reti ricorrenti usando queste *time series* in modo tale che, presentando agli ingressi del modello la sequenza di dati nota fino al tempo t, il modello sia in grado di predire quali saranno i prossimi valori dal tempo $t + 1$ in avanti. Queste previsioni possono servire ad anticipare specifici eventi (ad esempio quando è il momento giusto di comprare o vendere azioni in borsa), ma anche a rilevare anomalie in determinati andamenti come nella previsione di terremoti.

- **NLP**: il linguaggio viene inteso come il susseguirsi delle parole scritte in sequenze precise determinate dalla sintassi e grammatica della lingua in uso e determinano un contesto che può variare drasticamente il significato di una parola. Per questo motivo le reti ricorrenti si prestano molto bene ai task di NLP, come la *text classification*, la *machine translation*, *chatbot*.

- **Speech-Recognition**: il linguaggio parlato a voce composto da parole in sequenza, ogni parola è composta da lettere in sequenza, ogni lettera corrisponde ad un suono. Quindi il parlato è una sequenza di suoni emessi in ordine tale da formare parole e frasi. Anche in questo caso le reti ricorrenti tornano utili per decodificare le sequenze di suoni in sequenze di lettere, quindi parole e frasi.

2.3 Architettura Encoder-Decoder

Sebbene i modelli di Deep Learning come le Deep Neural Network (DNN) funzionino bene su set di dati di grandi dimensioni, non sono in grado di effettuare una *seq2seq prediction*, una famiglia di approcci di apprendimento sull'elaborazione del linguaggio. In questa famiglia vengono incluse svariate applicazioni come la traduzione da lingua a lingua, le didascalie nelle immagini, riempimento di parole mancanti, la *sentiment analysis*. In tutte queste applicazioni vi è un comune denominatore: un **architettura Encoder-Decoder** [11]. Si tratta di un modello end-to-end, ovvero addestrabile direttamente su frasi di origine e di destinazione. Il modello è capace di gestire testi in input di lunghezza variabile.

2.3.1 Struttura

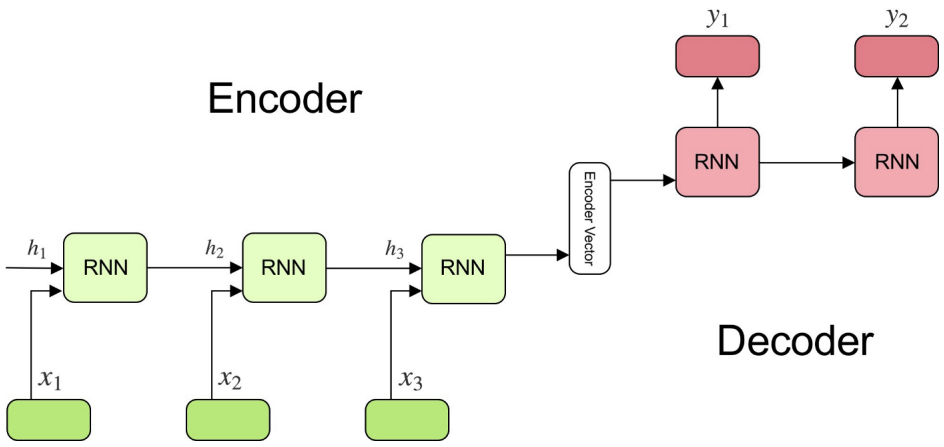

Figura 2.8: Architettura Encoder-Decoder LSTM

Osservando la Figura 2.8, si può notare come l'architettura sia basata su celle RNN che a loro volta solitamente si compongono di celle LSTM, infatti spesso prende il nome di Encoder-Decoder LSTM.

Questa architettura è composta da due modelli: uno per leggere la sequenza di input e codificarla in un vettore a lunghezza fissa, e un secondo per decodificare il vettore a lunghezza fissa ed emettere la sequenza prevista.

Il codificatore e il decodificatore del modello proposto sono addestrati congiuntamente per massimizzare la probabilità condizionata di una sequenza target data una sequenza sorgente.

Per l'Encoder (la parte a sinistra), il numero di passi temporali è uguale alla lunghezza della frase da tradurre. Ad ogni passo, c'è una pila di LSTM in

cui lo stato nascosto dell'LSTM precedente viene dato in ingresso al successivo. L'ultimo livello dell'ultima fase temporale genera un vettore che rappresenta il significato dell'intera frase. Il vettore viene quindi immesso in un'altra cella LSTM (il decodificatore) che produce le parole nella lingua di destinazione. Nel decodificatore (la parte destra) il testo viene generato in modo sequenziale; ogni passaggio produce una parola che viene inserita come input per il passaggio temporale successivo.

2.3.2 Vantaggi e svantaggi

Uno dei principali vantaggi nell'utilizzo di un architettura Encoder-Decoder è che la lunghezza delle sequenze in input e in output può essere differente.

Due punti, però, rendono questa architettura non soddisfacente: in primo luogo, l'intero significato della frase deve essere compresso nello stato nascosto tra l'Encoder e il Decoder. In secondo luogo, le LSTM in realtà non conservano le informazioni per più di circa 20 parole.

La soluzione a questi problemi esiste e si chiama Bi-LSTM (*Bidirectional* LSTM), che esegue due LSTM in direzioni opposte. In una Bi-LSTM il significato è codificato in due vettori, uno generato eseguendo una LSTM da sinistra a destra, e un altro da destra a sinistra. Ciò consente di raddoppiare la lunghezza della frase senza perdere troppe informazioni. Il modello Encoder-Decoder è il seme per meccanismi di attenzione e trasformatori.

2.3.3 Il meccanismo di attenzione

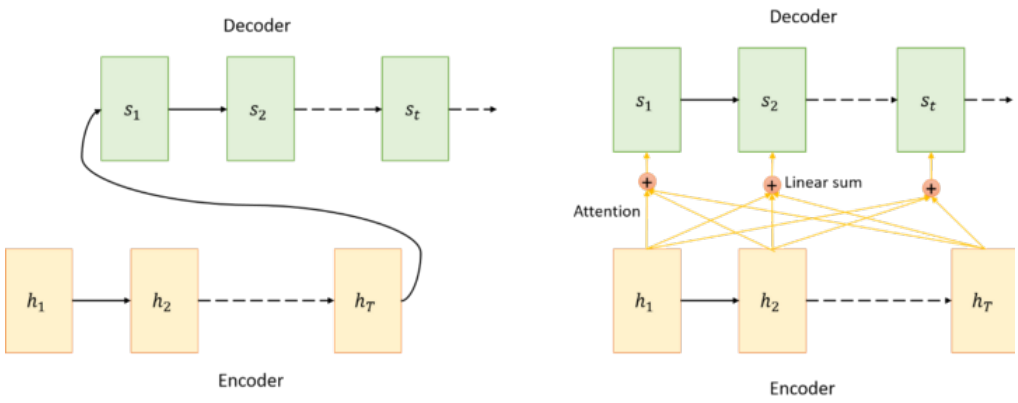

Figura 2.9: Architettura Encoder-Decoder tradizionale(a sinistra), meccanismo di attenzione(a destra)

Per ovviare al problema della compressione dei dati in uno stato nascosto, viene in soccorso il meccanismo di attenzione che aiuta a considerare tutti gli stati nascosti del codificatore per fare previsioni.

Il meccanismo riprende l'architettura Encoder-Decoder, con l'aggiunta di una somma pesata per ogni stato nascosto e per ogni nodo del codificatore in ogni fase temporale; dopodiché la previsione è fatta sullo stato nascosto più informativo. Per decidere quali stati nascosti siano più o meno utili alla previsione viene utilizzata una rete neurale.

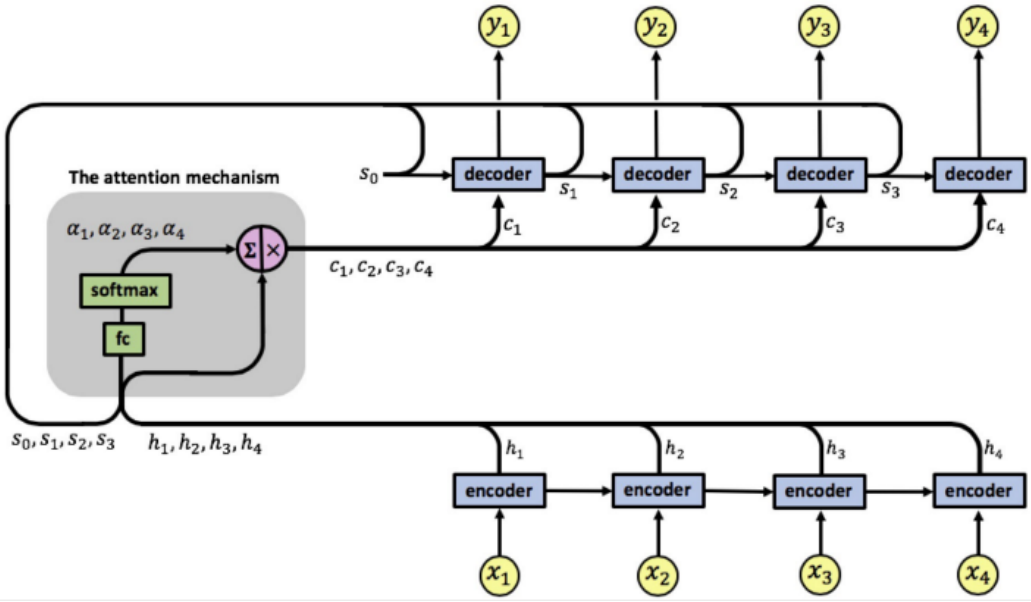

Figura 2.10: Architettura completa del meccanismo di attenzione

La Figura 2.10 è un riassunto chiaro di come agisce il meccanismo di attenzione. Tale processo si può suddividere in 4 passaggi:

1. **Codifica**: Ogni codificatore stabilisce il proprio input x_j per ogni passo temporale t e quindi restituisce un output h_j che sarà passato alla rete neurale del meccanismo di attenzione.

2. **Calcolo dei pesi dell'attenzione**: Per ogni step temporale la quantità di attenzione da prestare all'unità h_j è $\alpha_{t,j}$ ed è calcolata nel seguente modo, considerando sia l'output del codificatore nascosto corrente h_j sia lo stato nascosto del decodificatore allo step precedente s_{t-1}:

$$\alpha_{t,j} = softmax(e_{t,j}) = \frac{exp(e_{t,j})}{\sum_{k=1}^{T} exp(e_{t,k})} \tag{2.13}$$

dove:

$$e_{t,j} = \mathbf{a}(h_j, s_{t-1}), \forall j \in [1, T] \tag{2.14}$$

3. **Creazione del vettore contesto**: Il vettore contesto c_t è una semplice combinazione lineare dei valori degli output dei codificatori h_j di ogni fase temporale ponderati dai valori di attenzione $\alpha_{t,j}$:

$$c_t = \sum_{j=1}^{T} \alpha_{t,j} h_j \tag{2.15}$$

4. **Decodifica/Traduzione**: il vettore contesto c_t viene utilizzato insieme allo stato nascosto precedente del decodificatore s_{t-1} per calcolare il nuovo stato nascosto s_t e il nuovo output y_t.

2.3.4 "Attention is all you need"

Negli ultimi anni, i meccanismi di attenzione hanno trovato ampia applicazione specialmente nel *seq2seq learning*, in cui una sequenza di elementi (ad esempio un sequenza di parole in una frase) viene trasformata in un'altra diversa da quella di partenza. Questo torna particolarmente utile nella traduzione da lingua a lingua, tanto che questa modalità di apprendimento è diventata il cuore pulsante di Google Translate. Infatti, nel giugno 2017, Google ha pubblicato il documento ufficiale "*Attention is all you need*" (L'attenzione è tutto ciò di cui hai bisogno") [12], che oltre a trattare il *seq2seq learning*, basato sui meccanismi di attenzione visti in precedenza, ha presentato anche una nuova tecnologia: il **transformer** .

2.4 Transformer

Il Transformer [13] è un'architettura che si occupa di trasformare una sequenza in un'altra con l'aiuto di due parti (Encoder e Decoder), ma differisce dai modelli sequenza-sequenza precedentemente descritti perché non implica alcuna rete ricorrente. Le reti ricorrenti sono state, fino ad ora, uno dei modi migliori per catturare le dipendenze temporali in sequenze. Tuttavia, il team che ha presentato il documento ha dimostrato che un'architettura con solo meccanismi di attenzione senza RNN può migliorare i risultati. Un miglioramento sulle attività in linguaggio naturale (NLT) è stato presentato da **BERT**, che verrà analizzato dopo aver descritto la struttura del Transformer.

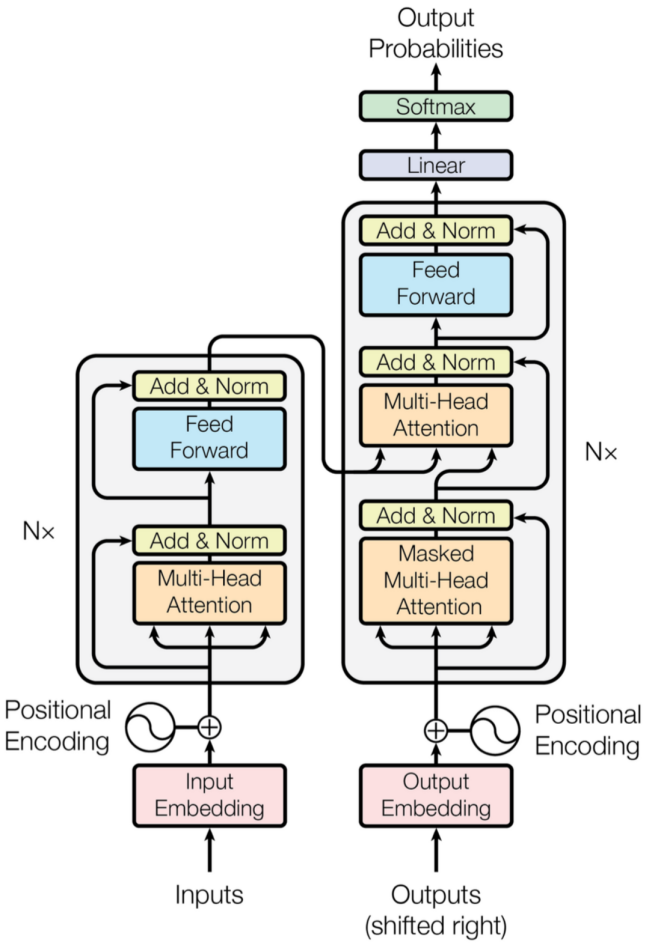

Figura 2.11: Struttura di un Transformer

2.4.1 Struttura del Transformer

Osservando la Figura 2.11, si nota come la struttura si possa suddividere in due parti: sulla sinistra si trova l'Encoder con la maggior parte dei moduli costituiti da strati **Multi-Head Attention** e *feedforward*; mentre sulla parte destra vi è il Decoder, con la sua parte superiore impilata N volte, dove arrivano gli output provenienti dall'*Encoder Stack*. Proprio come prima, il decodificatore emette una probabilità per ogni possibile parola successiva, ad ogni passaggio temporale.

2.4.2 Multi-Head Attention

Tutti questi livelli sono distribuiti nel tempo, quindi ogni parola viene trattata indipendentemente da tutte le altre, ma per la traduzione parola per

parola all'interno di una frase entra in gioco la *Multi-Head Attention*.

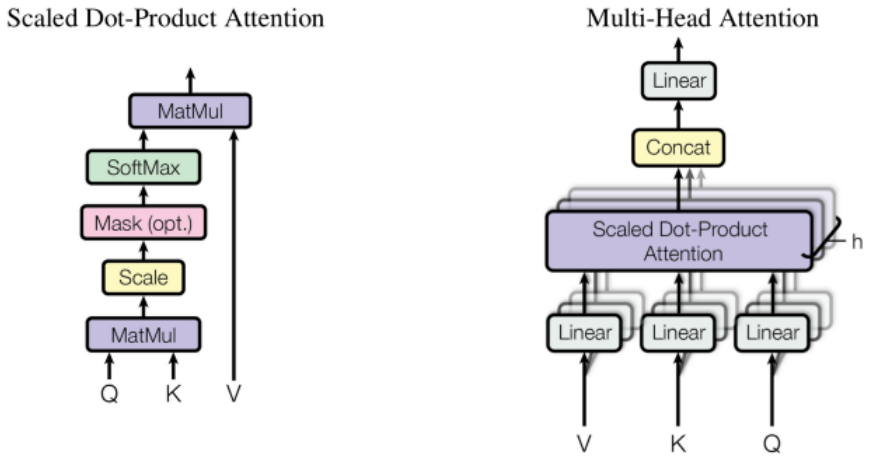

Figura 2.12: *Scaled Dot-Product Attention* (sulla sinistra) e *Multi-Head Attention* (sulla destra).

Il livello *Multi-Head Attention* dell'Encoder codifica la relazione di ogni parola con ogni altra parola nella stessa frase, prestando maggiore attenzione a quelle più rilevanti. Ad esempio, l'output di questo livello per la parola "capitale" nella frase "Parigi è la capitale della Francia" dipenderà da tutte le parole nella frase, ma probabilmente presterà maggiore attenzione alle parole "Parigi" e "Francia" che alle parole "è" o "la". Questo meccanismo di attenzione è chiamato auto-attenzione (la frase presta attenzione a sé stessa). Il livello *Masked Multi-Head Attention* del decodificatore fa la stessa cosa, ma ogni parola può occuparsi solo delle parole che si trovano prima di essa, le successive non vengono prese in considerazione. Infine, il livello di *Multi-Head Attention* superiore del decodificatore è il punto in cui viene prestata attenzione alle parole nella frase di input. Ad esempio, il decodificatore probabilmente presterà molta attenzione alla parola "capitale" nella frase di input quando sta per produrre la traduzione di questa parola. Come rappresentato nella Figura 2.12, il livello di *Multi-Head Attention* si basa sul livello **Scaled Dot-Product Attention**. L'utilizzo del *Dot-Product*, che di solito serve nel calcolo della similarità tra due vettori, non è casuale. Supponendo che l'Encoder abbia analizzato la frase "Loro hanno giocato a scacchi" e sia riuscito a distinguere le varie parti del predicato: "Loro" è il soggetto, "hanno giocato" è il verbo, e così via; e supponendo anche che il Decoder si trovi a dover tradurre il verbo, dopo aver

già tradotto il soggetto, verrà prodotto un dizionario chiave-valore costruito nella seguente maniera: {"soggetto": "loro", "verbo":"hanno giocato", ...}. Quindi, il Decoder deve cercare il valore corrispondente alla chiave "verbo". Il modello, tuttavia, non riesce a rappresentare le chiavi, ma ha una concezione vettoriale delle parti del discorso (apprese durante il training). Perciò, la chiave della ricerca (query) non corrisponderà ad una chiave nel dizionario ed occorrerà utilizzare la funzione *softmax* per convertire i punteggi ottenuti dal *Dot-Product* in pesi. Se la chiave rappresenta il "verbo" è probabile che sia simile alla query e il peso sarà vicino a 1. Nella Figura 2.12 si può osservare, infatti, la presenza di 3 matrici:

- **Q**: matrice contenente le query sulle righe, ovvero la rappresentazione vettoriale di una parola nella sequenza. La sua forma sarà , quindi, $n \times d$ dove n è il numero di query, mentre d è la lunghezza della chiave.
- **K**: è la matrice contenente tutte le chiavi, ovvero le rappresentazioni vettoriali di tutte le parole nella sequenza.
- **V**: è la matrice contenente i valori, che sono a loro volta le rappresentazioni vettoriali di tutte le parole nella sequenza.

Tutte e tre le matrici Q, K e V hanno dimensione uguale alla lista di parole nella sequenza in input, il ché significa che ogni parola nella frase verrà confrontata con ogni parola nella stessa sentenza, compresa sé stessa. Applicando, quindi, il meccanismo di attenzione alle matrici, si trova che:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.16}$$

Il prodotto punto-punto QK^T rappresenta, pertanto, la somiglianza di ogni query con le chiavi. L'output finale è una matrice che contiene il punteggio di somiglianza tra query e chiavi. Nell'Encoder, l'equazione viene applicata ad ogni frase di input del batch in questione. Nel Decoder, il procedimento è analogo, ma questa volta viene utilizzata una maschera per evitare il confronto con le parole che si trovano nelle posizioni successive ad essa. Il decodificatore, al momento dell'inferenza, avrà accesso solamente alle parole già tradotte precedentemente. L'addestramento, perciò, richiede il mascheramento dei futuri token di output. Inoltre, la funzione *softmax* viene applicata ai pesi per avere una distribuzione tra 0 e 1. Tali pesi vengono poi applicati a tutte le parole della sequenza che vengono introdotte in V.

L'immagine a destra nella Figura 2.12 descrive come questo meccanismo di attenzione può essere parallelizzato, ovvero essere utilizzato uno a fianco all'altro (da qui prende il nome la *Multi-Head Attention*). L'attenzione viene ripetuta più volte al fine di apprendere le diverse sfumature di *Q, K, V*.

Le rappresentazioni lineari si ottengono moltiplicando le tre matrici per le rispettive matrici di pesi trovate. Alla fine sia nell'Encoder che nel Decoder vi sarà una piccola rete *feedforward* con parametri identici per ogni posizione, che possono essere descritti come la trasformazione lineare di ciascun elemento della sequenza data. Da tale trasformazione prende il nome il Transformer [14].

2.4.3 Addestramento e inferenza

Durante **l'addestramento**, viene presa in input la frase di destinazione, spostata di un passo a destra, in quanto viene inserito un token di inizio sequenza all'inizio (il SOS, *start-of-sequence*). Perché? Una ragione è che il modello non deve imparare a copiare l'input del nostro decodificatore durante l'addestramento, ma deve imparare a prevedere la prossima parola data la sequenza dell'Encoder e una particolare sequenza del Decoder, che è già stata vista dal modello. Visto lo spostamento a destra, quindi, viene aggiunta la SOS, altrimenti quel posto sarebbe vuoto. Durante l'**inferenza**, ovvero la fase di risposta in cui il modello deve prevedere nuovi output, il Decoder non può ricevere i target, quindi vengono fornite le parole di output precedentemente trovate dall'Encoder (iniziando con un SOS). Quindi il modello deve essere chiamato ripetutamente, prevedendo una parola in più ad ogni round (che viene inviata al decodificatore al round successivo, fino a quando non viene emesso il token di fine sequenza (l'EOS, *end-of-sequence*).

2.5 Modelli nati da Transformer: il caso BERT

2.5.1 Definizione

Le tecniche basate sui Transformer rappresentano lo stato dell'arte per quanto riguarda i modelli di *Deep Learning* basati sui meccanismi di attenzione. Google, nel 2018, sviluppa e pubblica il modello **BERT** (*Bidirectional Encoder Representations from Transformers* [15]). BERT nasce come un modello pre-addestrato con dati non etichettati estratti tra 2 miliardi e mezzo di parole provenienti da *English Wikipedia*. La nascita di BERT ha permesso di raggiungere lo stato dell'arte in svariate applicazioni nell'ambito dell'NLP:

- **GLUE** (*General Language Understanding Evaluation*): sono una collezione di dataset per addestrare, valutare, analizzare modelli di NLP. La collezione consiste di 9 diversi task per testare la comprensione del linguaggio del modello.
- **SQuAD** (*Stanford Question Answering Dataset*): consiste in un set di domande e risposte per testare l'accuratezza dei modelli NLP

- **SWAG** (*Situations With Adversarial Generations*): questo tipo di dataset su larga scala unisce l'inferenza del linguaggio naturale con il ragionamento fisicamente fondato, attraverso il senso comune.

Gli autori di BERT proposero due attività principali per il suo pre-addestramento:

- **Masked Language Model** (MLM): Il mascheramento delle parole assegna una probabilità ad ogni parola di una frase di essere mascherata. Il modello si addestrerà cercando di predire quale parola ci dovrebbe essere sotto la maschera.

  Ad esempio, si supponga di avere in input la frase "Oggi vado al cinema per vedere un film". Vengono mascherate alcune parole e la frase diventa "Oggi <MASK> al cinema per vedere un <MASK>". Il modello deve predire esattamente le parole "vado" e "film", mentre gli altri output, che hanno una probabilità minore di essere nella frase, saranno ignorati.

- **Next Sentence Prediction** (NSP): Il modello viene addestrato per predire se due frasi siano consecutive oppure no. Ad esempio "Il cane dorme" e "Sta russando" sono consecutive, mentre "Il cane dorme" e "Il sole splende" non lo sono. Questo è abbastanza impegnativo per il modello, ma nel caso riuscisse a superare questo tipo di addestramento, allora migliorerà di gran lunga le sue performance.

Sicuramente BERT rappresenta una pietra miliare nella comprensione del linguaggio. Google ha affermato che BERT verrà utilizzato per tutte le sue query di ricerca, con un grande passo avanti nel capire non solo il contesto della query di ricerca, ma anche il suo significato implicito. Gli esempi di queste query non mancano. Nella ricerca "Can you get medicine for someone pharmacy," la sequenza "someone" e la sua relazione con le altre parole contenute nella query sono importanti per comprenderne il significato. Nel periodo avanti BERT, Google non avrebbe capito l'importanza di questa connessione e avrebbe restituito i risultati sulle prescrizioni mediche. Come spiega Google: con BERT, la funzione di ricerca ora è in grado di cogliere questa sfumatura e sapere che tutte le parole (anche le più comuni come "someone") sono di primaria importanza per la comprensione della semantica della frase.

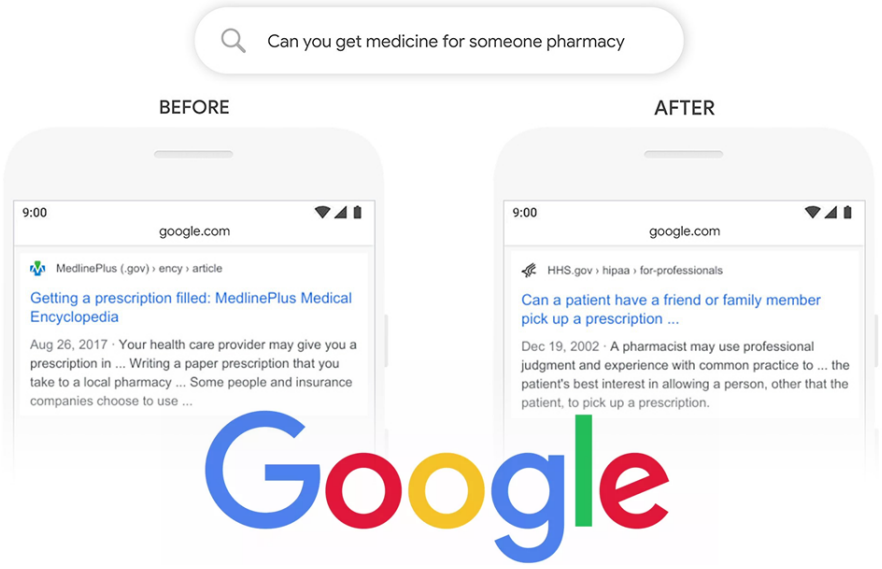

Figura 2.13: Esempio di ricerca prima e dopo la nascita di BERT

2.5.2 BERT in Python

In questa sezione verrà implementato un semplice programma per emulare le ricerche correlate di Google in Python tramite BERT [16]. Quando viene effettuata una ricerca sul browser, quest'ultimo restituirà la risposta più coerente a quello che viene cercato. Google cerca di individuare il topic della ricerca e interpretare la semantica della frase. Dopodiché stila una classifica dei termini più coerenti, per aiutare l'utente a raffinare la propria ricerca. Google, pertanto, si appoggia sull'attività di mascheramento MLM di BERT per dare i propri suggerimenti.

Si parte scaricando e importando la libreria `transformers`, la quale contiene molte architetture general-purpose tra cui BERT, GPT-2, RoBERTa, DistilBert, XLNet e molte altre per la comprensione del linguaggio naturale (NLU). Il modulo `pipeline` risulta essere utile per l'inferenza del modello scelto tra quelli presenti in `transformers`. Viene importata anche la libreria `pprint` per la stampa dei risultati.

```
pip install transformers

import transformers
from transformers import pipeline
import pprint
```

Successivamente, viene inizializzato un `unmasker`, ovvero una parola target

che il modello deve predire. Dentro la `format print` vi è `unmasker.tokenizer.mask_token` che si occupa di restituire un token per ogni predizione del modello. Il Transformer in questo caso è una variante più performante di BERT, ma con le stesse caratteristiche, chiamato Distil-BERT.

```
unmasker = pipeline("fill-mask")
pprint(unmasker(f"If you are not part of the
    {unmasker.tokenizer.mask_token}, then you are part of the
    problem"))
```

L'output si presenta come una lista delle soluzioni meglio rispettano il contesto della frase. In ogni elemento della lista viene stampato, inoltre, anche lo *score* che misura la probabilità che la predizione sia quella adatta alla frase, il predicato completo con la parola mancante, il codice del token della parola mancante e il token corrispondente sotto forma di stringa.

```
[{'score': 0.7982092499732971,
  'sequence': 'If you are not part of the solution, then you are part
     of the '
              'problem',
  'token': 2472,
  'token_str': ' solution'},
 {'score': 0.13441292941570282,
  'sequence': 'If you are not part of the problem, then you are part
     of the '
              'problem',
  'token': 936,
  'token_str': ' problem'},
 {'score': 0.0032895938493311405,
  'sequence': 'If you are not part of the answer, then you are part
     of the '
              'problem',
  'token': 1948,
  'token_str': ' answer'},
 {'score': 0.0030515999533236027,
  'sequence': 'If you are not part of the equation, then you are part
     of the '
              'problem',
  'token': 19587,
  'token_str': ' equation'}]
```

2.5.3 Valutazione del modello

La previsione di di BERT è inaffidabile se si vuole analizzare la correttezza fattuale, tuttavia se il modello viene pre-addestrato su un corpus con un vocabolario veramente rappresentativo della frase di input, allora l'accuratezza risulta essere coerente con la correttezza reale. Per esempio, supponendo di avere in input la frase: "<MASK> è usata per curare il raffreddore", non è detto che la previsione rispecchia quello che ci si aspettava, ma sicuramente BERT calcolerà che i migliori risultati saranno in gran parte dei farmaci o medicinali. Questa robusta capacità viene usata per valutare il modello. Vengono effettuati due tipi di test: **quantitativo** e **qualitativo**. Il primo valuta quanti sono i vettori sensibili al contesto, il secondo valuta la qualità del vettore [CLS], (un array che ha per elementi tutti i token di un certo cluster), esaminando a sua volta la qualità del vettore della frase mascherata predetta. Il test quantitativo più utilizzato è sicuramente l'**F1-measure** (che nella classificazione è una misura di valutazione del modello sulla base di *precision*, il numero di previsioni corrette per la classi A rispetto al numero di elementi della classe A, e *recall*, ovvero tutte le previsioni corrette fratto il numero di previsioni). Tale misura viene inserita in una **Matrice di confusione** (una visualizzazione semplice dei dati predetti).

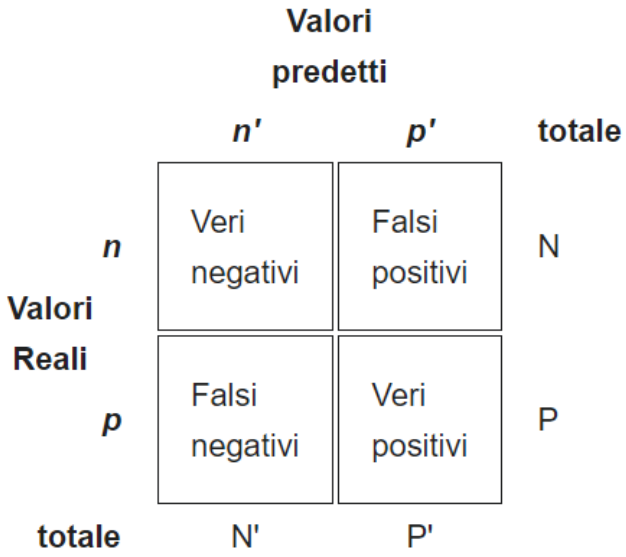

Figura 2.14: Esempio di matrice di confusione

Capitolo 3

Il linguaggio naturale

*"Il **linguaggio naturale** è la facoltà dell'uomo di comunicare ed esprimersi per mezzo di suoni articolati, organizzati in parole, atte a individuare immagini e a distinguere rapporti secondo convenzioni implicite, varie nel tempo e nello spazio"* [17]

Il linguaggio come forma articolata di comunicazione compare per la prima volta circa 100mila anni fa ed è fondamentale perché distingue l'*Homo Sapiens* da altre specie. Quando si parla di elaborazione del linguaggio naturale (NLP), si intende il processo di comprensione del dato letterale da parte di una macchina. Lo studio dell'NLP è cruciale per 3 ragioni:

- **comunicazione** uomo-macchina senza l'utilizzo di linguaggi formali;
- **apprendimento** della macchina dalla grande mole di dati disponibili sul linguaggio naturale;
- **comprensione** in modo scientifico e semantico del linguaggio.

Lo studio del linguaggio, tuttavia, non è particolarmente semplice, in quanto presenta numerose ambiguità, ridondanze, significati impliciti, dipendenze dal contesto, tanto che Alan Turing lo definì come una delle peculiarità dell'intelligenza artificiale che la rendono tale. In questo capitolo verranno esaminate nel dettaglio quali sono le tecniche principali di elaborazione del linguaggio naturale (*Natural Language Processing*, NLP), a partire dal testo (l'unico modo in cui una macchina può percepire il linguaggio).

3.1 L'analisi del testo

La macchina può apprendere il linguaggio naturale solamente attraverso un dato di tipo testuale. Tutti gli altri tipi di dato, specialmente l'audio, vengono

dapprima trasformati in forma scritta per poi essere appresa. Il testo, quindi, è un'informazione fondamentale nell'apprendimento. Si tratta di un tipo di **dato non strutturato**, in quanto non è provvisto di uno schema in grado di spiegarne canonicamente la semantica. Al contrario, i **dati strutturati** sono, per esempio, lo *score* o il *rating* di un certo prodotto in vendita, da cui si può ricavare facilmente la media e la posizione in classifica. Nel caso in cui viene analizzato un commento per un certo prodotto, bisogna stabilire un criterio di valutazione per il dato destrutturato. Infatti, per questo tipo di dato non si possono estrarre canonicamente *score* o *rating*, ma bisogna analizzare a fondo il testo del commento per capire se è negativo o positivo. Tuttavia il testo rileva delle trappole e delle insidie dovute alle sue caratteristiche intrinseche:

- In generale, ogni testo è all'interno di un contesto, quindi bisogna interpretarlo, in quanto potrebbe avere più di un significato. Contestualizzare vuol dire applicare il significato pertinente al concetto. Bisogna prestare attenzione anche a sarcasmo, ironia e metafore, difficilmente individuabili dalla macchina.
- Il testo dipende dalla lingua in cui è scritto. Ognuna delle quali presenta alfabeti, regole strutturali, sintassi e parole differenti.
- Potrebbe contenere errori ortografici o semantici e potrebbe utilizzare elementi di uso comune non previsti dalla lingua come abbreviazioni/emoticon.

3.2 Le tecniche di pre-processing

Esiste una fase di **pre-processamento** del testo che consiste nel disambiguare le varie criticità: il testo viene trasformato in una forma più interpretabile dal calcolatore. In questa fase vengono applicate varie operazioni da usare dipendentemente dal contesto:

- **Segmentazione (*tokenitation*)**: consiste nell'analizzare lessicalmente il testo, scomponendolo in elementi sintattici di base (*token*), solitamente frasi o parole, ma anche l'eliminazione degli elementi di punteggiatura o delle abbreviazioni. Questi elementi vengono poi classificati in una fase di analisi sintattica successiva detta *Part of Speech*.
- **Part of Speech (POS)**: i *token* individuati nella fase precedente vengono classificati secondo la grammatica in: verbo, nome, aggettivo, ecc. A seconda delle esigenze si possono suddividere anche in plurale/singolare, nomi comuni/propri ecc. Ad ogni POS viene assegnata un'etichetta per tenere conto del significato della parola stessa.

- **Filtraggio delle parole**: le parole vengono normalizzate e uniformate per eliminare le differenze di forma:
  - **Casefolding**: tutte le parole vengono convertite minuscolo/maiuscolo.
  - **Rimozione Stopword**: alcune parole non particolarmente indicative sull'argomento del testo come: "e", "di", "non" vengono rimosse.
  - **Lemmatizzazione**: consiste nel sostituire ciascuna parola con il lemma, la sua forma base: ad esempio le parole "dormi", "dormisse", "dormano" sono da convertire nel loro lemma "dormire".
  - **Stemming**: simile alla lemmatizzazione ma viene ricavata la radice morfologica: "nuotare" diventa "nuot". In questo caso potrebbe risultare delle parole senza senso e può comportare una perdita di precisione.

Aldilà della forma e della struttura, apprendibili dal calcolatore tramite algoritmi, la caratteristica sicuramente più affascinante dell'NLP è il riconoscimento della semantica: contestualizzare una frase, conoscere il significato delle parole e metterle in relazione tra di loro. Per fare ciò, il contenuto di un testo viene rappresentato in modo strutturato: le parole più indicative, etichettate secondo le regole di *pre-processing* viste precedentemente, vengono inserite in un multiset detto **Bag of Words (BOW)**. Si tratta di tutte le parole distinte con il numero di occorrenze all'interno del testo. In realtà le parole vengono suddivise in *n-gram*, ovvero sequenze di parole. Se tutte le parole sono distinte tra loro e non sono in sequenza si tratta *unigram*, mentre parole come "New York" compongono il *bi-gram*, e così via. Questa tecnica non è particolarmente efficace in quanto la maggior parte degli *n-gram* non sono significativi. Viene quindi implementato un **Vector Space Model** rappresentante un insieme di documenti come vettori. Sostanzialmente, si tratta di una matrice $D \times N$ dove D è il numero di documenti e N è il numero di occorrenze dei termini.

Attraverso la rappresentazione vettoriale si può misurare la similarità dei documenti. Visto che la distanza euclidea tra vettori può risultare alquanto onerosa (i *token* di un testo possono raggiungere tranquillamente l'ordine di 10^6 elementi), si usa molto spesso la similarità coseno, che misura il coseno dell'angolo compreso tra i due vettori (più l'angolo è piccolo, più i due vettori sono simili):

$$cos(a, b) = \frac{a * b}{||a|| * ||b||} = \frac{\sum_{i=1}^{n} a_i * b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} * \sqrt{\sum_{i=1}^{n} b_i^2}} \tag{3.1}$$

3.2.1 Il TF-IDF

Il numero di occorrenze, tuttavia, non è l'unico modo per "pesare" la significatività di un termine. Un modo più efficace consiste nel calcolare il **tf-idf** (term frequency - inverse document frequency), ovvero il rapporto tra il valore del termine rispetto al numero di apparizioni nell'intera collezione di documenti:

- **term frequency**: è il fattore locale, il numero di apparizioni del termine in un documento.
- **inverse document frequency**: è il fattore globale, pesa ciascun termine all'interno dell'intera collezione. Se un termine compare in meno documenti è più significativo, perché in grado di distinguerli meglio.

Il *tf-idf* dato un termine t ed un documento d si calcola:

$$tf.idf(t,d) = log(f(t,d)) * log(\frac{|D|}{|d \in D : t \in d|}) \tag{3.2}$$

3.3 Word Embedding

3.3.1 Definizione

Finora si è considerato ogni termine a sé stante e indipendente dagli altri. Lo studio svolto fino a questo momento si occupa di **Latent Sentiment Analysis (LSA)** che affida ai documenti una dimensione corrispondente ad un livello semantico. Altri modelli, invece, si occupano delle similarità a livello semantico delle parole: è il caso del **Word Embedding**.

Quando due parole sono semanticamente simili allora il loro contesto è simile. Risulta importante, quindi, mappare ogni parola in uno spazio multidimensionale in modo che due parole semanticamente simili siano vicine in tale spazio. In questo modo, se nello spazio ci sono due parole con contesto simile, dovrebbero essere rappresentate da vettori simili (con dimensioni uguali a quelle dello spazio di appartenenza). Questi due vettori vengono chiamati, appunto, Word Embedding (WE) e fanno parte della famiglia degli algoritmi non supervisionati.

La tecnica del WE viene insegnata a modelli che fanno parte della famiglia dei **Word2Vec** [18], che si compongono di una semplice rete neurale artificiale a due strati che restituisce la distribuzione semantica delle parole. Ogni parola rappresenta un vettore multidimensionale, quindi un punto nello spazio considerato. In tale spazio, più due punti sono vicini, più i vettori sono simili e le due parole hanno una probabilità maggiore di riferirsi allo stesso contesto.

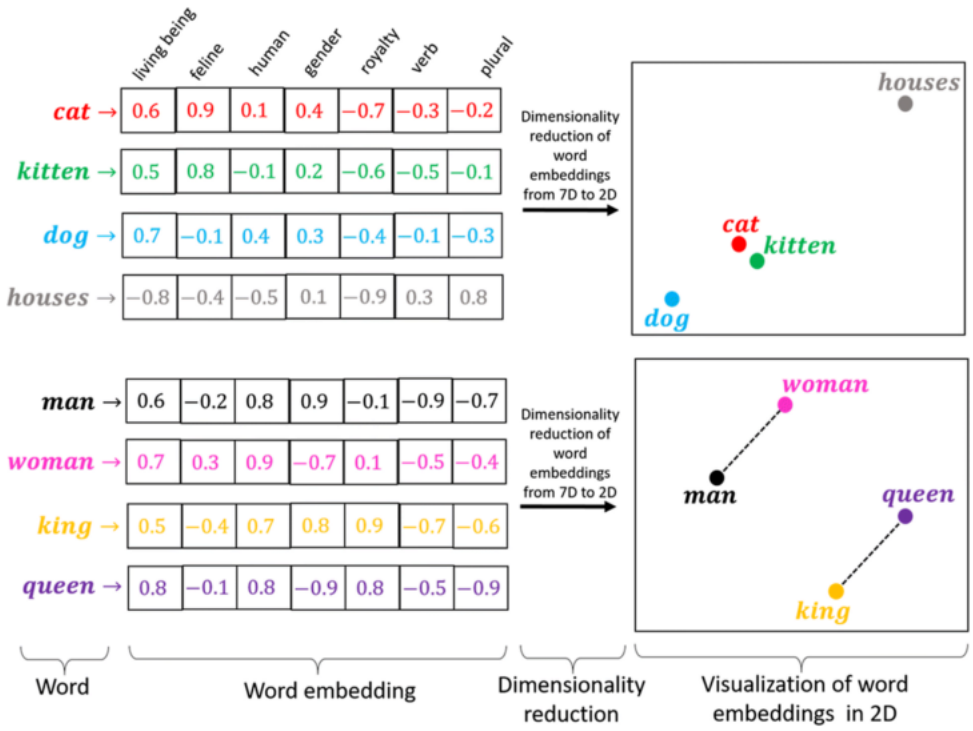

Figura 3.1: Tabella del peso trovato di ogni parola rispetto alle varie classi grammaticali (sinistra). Rappresentazione 2D dei vettori (destra).

La trasformazione da parola a vettore si può considerare come una funzione che mappa le parole in uno spazio *n-dimensionale*:

$$W : words \rightarrow R^n \tag{3.3}$$

La rete addestrata restituisce una probabilità per ogni parola del vocabolario (esclusa la parola in input) che essa sia contestualmente vicina all'input.

3.3.2 Addestramento di un'architettura Word2Vec

Supponendo di avere in input la seguente frase:

"Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura"

ogni parola viene dapprima codificata in un vettore secondo **One-Hot** (tutti i numeri del vettore sono a 0 tranne un unico numero a 1):

nel	1	0	0	0	0	0	0	0	0	0	0	0	0
mezzo	0	1	0	0	0	0	0	0	0	0	0	0	0
del	0	0	1	0	0	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
selva	0	0	0	0	0	0	0	0	0	0	0	1	0
oscura	0	0	0	0	0	0	0	0	0	0	0	0	1

La rappresentazione vettoriale dei token sarà circoscritta per forza in uno spazio a 13 dimensioni visto che sono contenute 13 parole nella frase in input.

La rete apprende i pesi da assegnare ai rispettivi token tramite il metodo di *backpropagation* visto in precedenza. La sua struttura è abbastanza semplice e si compone dei tre strati soliti: input, hidden, output. Nell'ultimo strato viene applicata la funzione *softmax*. L'output finale è una matrice W di grandezza $N \times V$ contenente la rappresentazione WE dei token V_i. Per N, invece, si intende la dimensione del WE. Da questo modello nascono le due strutture **CBOW** e **Skip-Gram**. Entrambe utilizzano una finestra di contesto la cui dimensione è denotata con C, che include le parole immediatamente precedenti e successive del termine con ampiezza uguale a C. Ad esempio con C=*2* si ha:

token corrente	**finestra di contesto**
nel	(nel, mezzo) (nel, del)
mezzo	(nel, mezzo) (mezzo,del) (mezzo,cammin)
cammin	(mezzo, cammin)(del,cammin)(cammin,di)(cammin,nostra)
di	(del,di)(cammin,di)(di,nostra)(di,vita)
nostra	(cammin,nostra)(di,nostra)(nostra,vita)
vita	(di,vita)(nostra,vita)

Di seguito viene presentata la logica e l'architettura dei due modelli, i quali prendono spunto dai concetti di *Bag of Words* e *n-gram*, visti tra le tecniche di *pre-processing*:

- **Continuos Bag of Words (CBOW)**: ha lo scopo di predire il token corrente a partire dalla finestra di grandezza $\pm m$ parole del contesto:

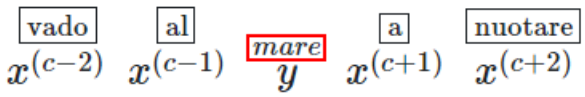

Figura 3.2: Esempio predizione CBOW

La rete viene addestrata nel seguente modo:

1. Prende in input le codifiche *One-Hot* e viene calcolata la media dei loro word-embedding della matrice V (che ha come colonne gli embedding delle parole di contesto). La moltiplicazione tra V e la media costituisce il contesto che sarà passato all'*hidden layer*.
2. In questo strato viene ottenuto il vettore z dalla moltiplicazione tra il contesto e l'i-esima riga della matrice U (contenente sulle righe gli embedding delle parole w_j, quando sono parole centrali).
3. Infine, viene calcolata la funzione *softmax* su z per ricavarne una distribuzione di probabilità che la parola centrale sia indicata per quella frase.

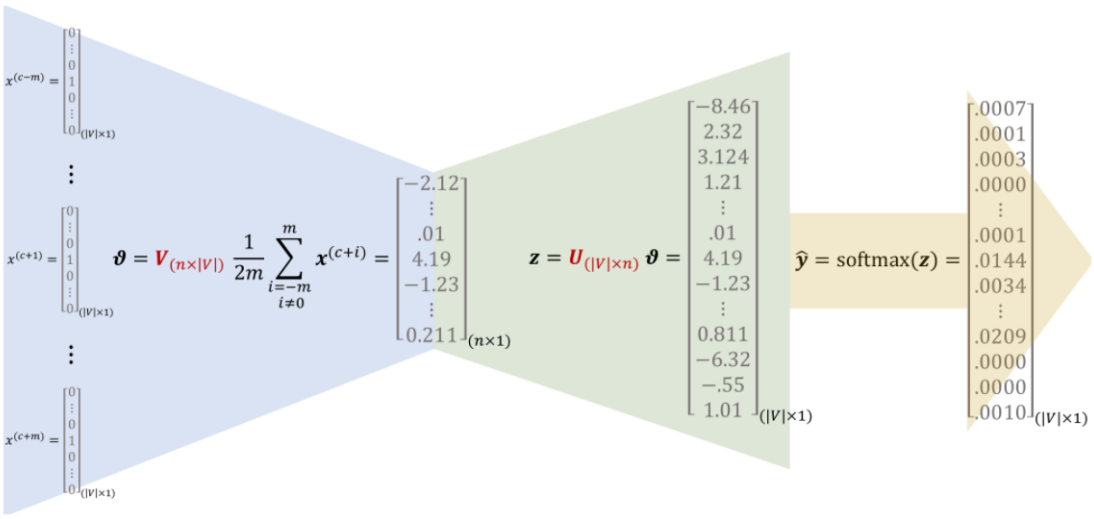

Figura 3.3: Addestramento del modello CBOW

- **Skip-Gram**: questo modello è speculare al precedente, ovvero cerca di predire il contesto data una parola centrale, come per esempio:

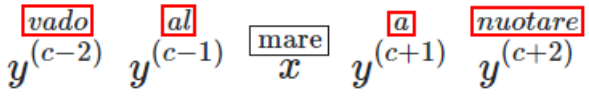

Figura 3.4: Esempio predizione Skip-Gram

Il modello viene, quindi, addestrato su una codifica *One-Hot* della parola centrale di dimensione V e le $2*m$ parole della finestra $(y^{(c-m)}, \ldots, y^{(c+m)})$ che saranno confrontate con la distribuzione di probabilità per l'aggiornamento dei parametri. Le matrici U e V in questo caso contengono rispettivamente gli embedding delle parole di contesto e gli embedding delle parole centrali. L'addestramento è simile al caso di CBOW, ma in questo caso non viene calcolata la media nell'input layer e, specularmente, si vanno ad apprendere 2 *word embedding* per ogni parola, a seconda che essa sia di contesto o centrale.

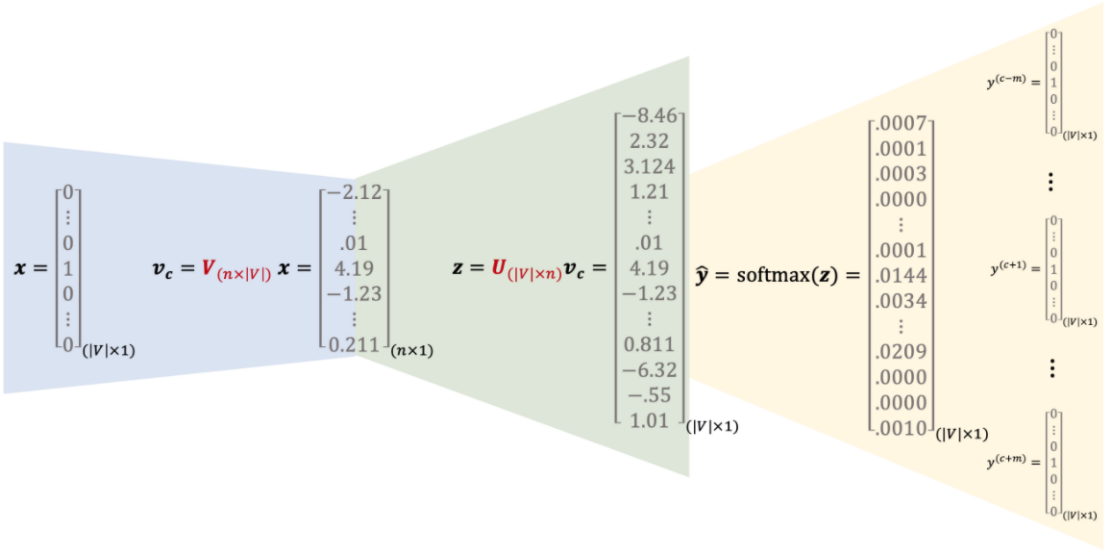

Figura 3.5: Addestramento del modello Skip-Gram

3.3.3 Limitazioni

Il *Word2Vec*, tuttavia non è perfetto, ma presenta delle limitazioni sostanziali:

- l'aggiunta o la rimozione di una parola comporta il riaddestramento completo del modello;
- non viene risolto il problema della polisemia (la capacità di una stessa parola di assumere diversi significati) tra i termini.

.

Capitolo 4

Applicazioni e modelli linguistici

Un modello linguistico associa una probabilità ad una sequenza di parole determinata a partire da un corpus. Le sue applicazioni sono molteplici e spaziano dal confronto sulla verosimiglianza di frasi al calcolo della probabilità della parola successiva in una frase; dall'autocompletamento alla correzione di parole fuori contesto in una frase; dall'individuazione dell'output più probabile nella *speech recognition* al supporto predittivo alla comunicazione aumentata e accessibile. I modelli linguistici hanno subito un'evoluzione che si adegua al trend dell'intelligenza artificiale. In questo capitolo vengono esplorati i principali modelli ed applicazioni del linguaggio, partendo dai più semplici per arrivare a quelli più complicati.

4.1 I modelli probabilistici

I precursori dei modelli più complessi sono sicuramente le architetture basate su metodi probabilistici che devono la loro nascita al calcolo della probabilità. La probabilità di determinare una singola parola corretta (*uni-gram*) all'interno di una frase è data semplicemente dal rapporto tra il numero di occorrenze $C(w)$ della parola fratto la dimensione m della frase:

$$P(w) = \frac{C(w)}{m} \tag{4.1}$$

La cosa si complica se si vanno a considerare gli *n-gram*: la probabilità di trovare un *n-gram* corretto diventa il rapporto tra le occorrenze dell'*n-gram* C_1^N fratto tutte le occorrenze degli altri *n-gram* della frase le cui $n-1$ prime parole sono w^{n-1}.

$$P(w_n|w_1^{n-1}) = \frac{C_1^n}{\sum(C(w_1^{n-1*w})} \tag{4.2}$$

Se, invece, si vuole trovare la probabilità di una frase (intesa come sequenza ordinata di parole) all'interno di un testo allora si calcola $P(w_1, w_2, ..., w_n)$. Questo è il classico esempio di probabilità condizionata che si calcola nel seguente modo:

$$P(w_1^n) = P(w_1) \prod_{i=2}^{n} P(w_i | w_1^{i-1}) \tag{4.3}$$

Per esempio, la probabilità della frase "ieri ero al cinema" è data da:

$$P(ieri\, ero\, al\, cinema) = P(ieri) \cdot P(ero|ieri) \cdot P(al|ieri\, ero) \cdot P(cinema|ieri\, ero\, al) \tag{4.4}$$

4.1.1 Approssimazione di Markov

Con l'aumentare del testo bisogna valutare tutte le possibili frasi precedenti e ciò non permetterebbe al modello di generalizzare. Si formula, quindi, la cosiddetta **approssimazione di Markov**, in cui:

$$P(w_k | w_{k-m+1}^{k-1}) \approx P(w_k | w_{k-n+1}^{k-1}) \tag{4.5}$$

ovvero che la probabilità di una parola che sia condizionata solo dalle n parole precedenti. Partendo dall'esempio precedente, la probabilità risultante cambia nel seguente modo:

$$P(ieri\, ero\, al\, cinema) = P(ieri) \cdot P(ero|ieri) \cdot P(al|ero) \cdot P(cinema|al) \tag{4.6}$$

4.1.2 Esempio con i delimitatori

Finora non venivano considerate informazioni importanti relative al contesto di inizio e fine frase. A questo scopo vengono aggiunti dei **delimitatori** $< d >$ che hanno il compito di far considerare alla rete anche la prima e l'ultima parola così come le altre.

$$\begin{aligned} P(< d > \; ieri\, ero\, al\, cinema < d >) = P(ieri| < d >) \cdot P(ero|ieri) \\ P(al|ero) \cdot P(cinema|al) \cdot P(< d > |cinema) \end{aligned} \tag{4.7}$$

Definiti i criteri verrà costruita una matrice con le occorrenze degli *n-gram* e quella delle probabilità di una frase, considerando come frasi d'esempio:

Oggi vado al mare
Al mare oggi piove
Se piove vado al parco

$C(w_i\|w_{i-1})$	$< d >$	oggi	vado	al	mare	piove	se	parco
$< d >$	0	1	0	1	0	0	1	0
oggi	0	0	1	0	0	1	0	0
vado	0	0	0	2	0	0	0	0
al	0	0	0	0	2	0	0	1
mare	1	1	0	0	0	0	0	0
piove	1	0	1	0	0	0	0	0
se	0	0	0	0	0	1	0	0
parco	1	0	0	0	0	0	0	0

Tabella 4.1: Tabella occorenze parole

$P(w_i\|w_{i-1})$	$< d >$	oggi	vado	al	mare	piove	se	parco
$< d >$	0	0.33	0	0.33	0	0	0.33	0
oggi	0	0	0.5	0	0	0.5	0	0
vado	0	0	0	1	0	0	0	0
al	0	0	0	0	0.67	0	0	0.33
mare	0.5	0.5	0	0	0	0	0	0
piove	0.5	0	0.5	0	0	0	0	0
se	0	0	0	0	0	1	0	0
piove	1	0	0	0	0	0	0	0

Tabella 4.2: Tabella probabilità *uni-gram*

Per calcolare la probabilità di una qualunque frase w_1^n, inseriti gli opportuni delimitatori, verrà effettuata una produttoria delle probabilità:

$$P(< d > \; oggi\, piove \; < d >) = P(oggi| < d >) \cdot P(piove|oggi) \cdot P(< d > |piove) = \\ = 0.33 \cdot 0.50 \cdot 0.50 = 0.0825 \tag{4.8}$$

4.1.3 Limiti e osservazioni

Uno dei limiti principali del modello di Markov è la gestione delle parole sconosciute. Una possibile soluzione al problema è considerare il numero di occorrenze medio k del testo e sostituire tutte le parole non presenti nel corpus con il nuovo token $< unk >$. Riprendendo l'esempio di prima e consinderando $k = 2$, le ultime 2 parole ("se" e "parco") vengono sostituite con $< unk >$. Viene trovata, quindi, una nuova matrice delle probabilità con la sostituzione delle ultime due righe di "se" e "parco" con la nuova riga $< unk >$.

$P(w_i\|w_{i-1})$	$< d >$	oggi	vado	al	mare	piove	$< unk >$
$< unk >$	0.50	0	0	0	0	0.50	0

Ad esempio inserendo le parole sconosciute "Quando" e "museo", la probabilità verrà calcolata nel modo seguente:

$$P(< d > \ Quando\, piove\, vado\, al\, museo \ < d >) = 0,33 \cdot 0,50 \cdot 0,50 \cdot 1 \cdot 0,33 \cdot 0,50 \approx 0,014 \tag{4.9}$$

Un altro limite è la grande quantità di *n-gram* con probabilità a 0 che rendono nulla ogni altra probabilità. Questo limita la generalizzazione del modello. Per ovviare al problema esistono 3 approcci principali:

- **K-smoothing**: questo approccio consiste nel ricalcolare la matrice delle probabilità, riconsiderando tutte le probabilità con la seguente formula:

$$P(w_i|w_1^{n-1}) = \frac{C(w_1^n) + k}{C(w_1^{n-1}) + k \cdot |V|} \tag{4.10}$$

  In questo modo anche con un semplice *smoothing* di $k = 0.1$ tutte le probabilità diventano diverse da 0 e il modello è in grado di generalizzare tutte le parole.

- **Backoff**: Viene utilizzato un coefficiente di riduzione $d = 0.4$ che viene moltiplicato per l'*(N-1)-gram* corrispondente al *N-gram* corrente. Il processo viene ripetuto finché non si trova una parola con probabilità a 0. Ipotizzando di avere un *tri-gram*: "Leggo un libro " con valore di probabilità nullo, viene calcolato $d \cdot P(libro|un)$; nel caso anche quest'ultimo allora si calcola $d^2 \cdot P(libro)$

- **Interpolazione**: Simile all'approccio precedente, ma si va a ponderare con opportuni coefficienti decrescenti per tutti gli *N-gram* via via di ordine inferiore:

$$\hat{P}(w_{n-N+1}^n) = \sum_{i=1}^{N} \lambda_i P(w_{n-N+i})\, con \sum_{i=1}^{N} \lambda_i = 1 \tag{4.11}$$

Nell'esempio si ha che:

$$\hat{P}(libro|leggo\, un) = \lambda_1 \cdot P(libro|leggo\, un) + \lambda_2 \cdot P(libro|un) + \lambda_3 \cdot P(libro) \tag{4.12}$$

4.1.4 Part of Speech Tagging con modello di Markov

L'etichettatura delle parti del discorso (nome, aggettivo, verbo, etc...) è uno dei task più classici dell'NLP. Questo compito è reso particolarmente difficile

dalla varietà di caratteristiche che può assumere una parola: ad esempio la parola inglese "set" può assumere la funzione di verbo, nome, aggettivo a seconda del contesto. Il task utilizza il modello di Markov con gli stati nascosti, il quale può ottenere risultati con accuratezza apprezzabile, senza l'utilizzo del word embedding. Il modello di Markov con gli stati nascosti presenta le stesse caratteristiche del modello visto precedentemente, ma con l'aggiunta degli stati nascosti. Gli stati corrispondono alle parti lessicali del discorso e si considerano nascosti in quanto non è conosciuta a priori la sequenza di stati corrispondenti alle caratteristiche del discorso. Nel modello viene aggiunto anche lo stato π che corrisponde al delimitatore, il quale rimane l'unico noto a priori. Le **transizioni** sono i passaggi di stato e corrispondono alla probabilità di quanto una parola successiva di uno stato corrente S_i sia riconducibile allo stato successivo S_j. Tale probabilità è data da:

$$P(S_j|S_i) = \frac{C(S_i, S_j) + \epsilon}{C(S_i) + \epsilon \cdot N} \tag{4.13}$$

Dove $C(S_i, S_j)$ è il conteggio del numero di occorrenze nella sequenza dei due stati e $C(S_i)$ è quello dello stato corrente. ϵ è una piccola costante di *smoothing* per quelle transizioni con probabilità nulle, mentre N è il numero totale di stati. Tutte queste probabilità vengono inserite in una matrice $A_{N\times N}$. Le **emissioni**, invece, sono associate alle rispettive probabilità del testo di training:

$$P(w_j|S_i) = \frac{C(S_i, w_j) + \epsilon}{C(S_i) + \epsilon \cdot |V|} \tag{4.14}$$

dove $C(S_i, w_j)$ è il numero di occorrenze della parola S_i e $|V|$ è il numero di parole nel dizionario V, che corrisponde a tutte le parole che hanno una frequenza superiore ad un certo k, con l'aggiunta dei delimitatori $< d >$ e delle parole sconosciute $< unk >$ dei rispettivi stati (si ha quindi $< unk_{Stato} >$). Le probabilità vengono aggiunte in una matrice di emissione $B_{N\times|V|}$.

Si suppone di avere 3 frasi con 3 stati: Articolo (A), Verbo (V), Nome (N).

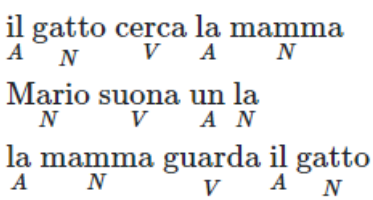

Figura 4.1: Esempio di frasi con i rispettivi stati

La forma della matrice con le occorrenze delle transizioni sarà:

$$\begin{pmatrix} C(S_i, S_j) & \pi & A & N & V & C(S_i) \\ \pi & 0 & 2 & 1 & 0 & 3 \\ A & 0 & 0 & 5 & 0 & 5 \\ N & 3 & 0 & 0 & 3 & 6 \\ V & 0 & 3 & 0 & 0 & 3 \end{pmatrix} \tag{4.15}$$

mentre la matrice con le occorrenze delle emissioni avrà la seguente forma:

$$\begin{pmatrix} C(S_i, w_j) & il & gatto & \dots & guarda & < unk > & < d > \\ \pi & 0 & 0 & \dots & 0 & 0 & 4 \\ A & 2 & 0 & \dots & 0 & 0 & 0 \\ N & 0 & 2 & \dots & 0 & 0 & 0 \\ V & 0 & 0 & \dots & 1 & 0 & 0 \end{pmatrix} \tag{4.16}$$

Partendo da queste due matrici vengono calcolate anche la matrici di transizione A (considerando $\epsilon = 0.01$ e la probabilità $P(\pi|\pi) = 0$):

$$\begin{pmatrix} P(S_i, S_j) & \pi & A & N & V \\ \pi & 0 & 0.663 & 0.333 & 0.003 \\ A & 0.002 & 0.002 & 0.994 & 0.002 \\ N & 0.498 & 0.002 & 0.002 & 0.498 \\ V & 0.003 & 0.990 & 0.003 & 0.003 \end{pmatrix} \tag{4.17}$$

e di emissione B (considerando sempre $\epsilon = 0.01$ tranne per π e $< d >$):

$$\begin{pmatrix} P(w_j, S_i) & il & gatto & \dots & guarda & unk & < d > \\ \pi & 0 & 0 & \dots & 0 & 0 & 1 \\ A & 0.394 & 0.002 & \dots & 0.002 & 0.002 & 0 \\ N & 0.002 & 0.330 & \dots & 0.002 & 0.002 & 0 \\ V & 0.003 & 0.003 & \dots & 0.326 & 0.003 & 0 \end{pmatrix} \tag{4.18}$$

Se si effettua il prodotto delle matrici $A \cdot B$ si troverà la probabilità $P(w_j t + 1|S_i^t)$ che la parola successiva allo stato S_i^t sia w_j^{t+1}. Nella Figura 4.2 vengono rappresentati i vari stati con le rispettive transizioni (frecce rosse) ed emissioni (frecce verdi) con sopra indicate le probabilità di ciascuna freccia.

Esistono ottimizzazioni al processo markoviano come **l'algoritmo di Viterbi** [19], che si occupa di scegliere il percorso più vicino alla sequenza in input all'interno del campo con tutte le possibilità per quella sequenza. Ad ogni passo l'algoritmo elimina i percorsi meno probabili secondo un certo criterio (distanza di Hamming rispetto all'input o distanza euclidea). Alla fine rimarrà un solo superstite che sarà il candidato principale per lo stato da assocciare alla parola.

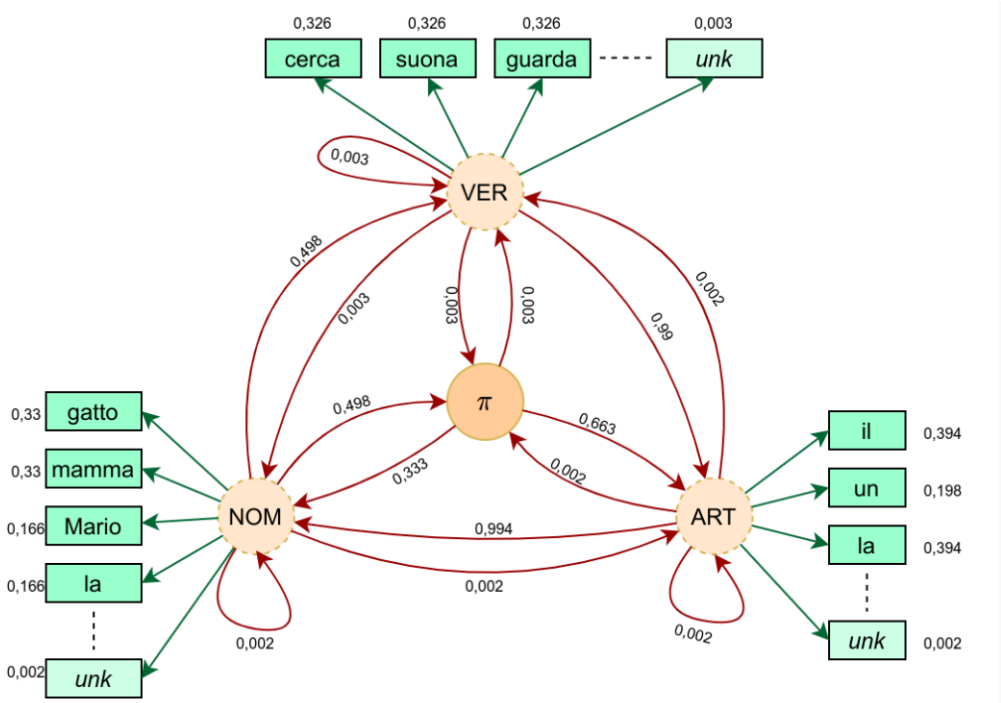

Figura 4.2: Esempio di predizione con le matrici di transizione ed emissione

4.2 Operazioni nello spazio vettoriale

Una volta capito come effettuare la traduzione delle parole in vettori (Word2Vec) è possibile operare algebricamente su tali vettori per sfruttare alcune delle loro caratteristiche per ricercare risultati semanticamente rilevanti come similitudini, analogie e relazioni tra significati delle parole.

4.2.1 Le principali operazioni vettoriali in Python

In questa sezione viene implementato del codice Python per osservare a livello pratico come vengono effettuate le principali operazioni tra vettori. Infine, verranno analizzati i risultati.

Si parte prendendo come esempio il Dataset "reddit_worldnews" da Kaggle che fungerà da corpus per alcuni esempi. In questo Dataset sono presenti le principali notizie mondiali dal social network *Reddit*.

Per iniziare viene scaricata la libreria **`gensim`**, utile nell'NLP ed in particolare nell'indicizzazione di documenti e nel recupero di similarità partendo da corpus di grandi dimensioni.

```
pip install gensim
```

Vengono inoltre importate le più classiche librerie per l'elaborazione del linguaggio e dei dati: **Pandas**, **Numpy**, **NLTK** e dalla libreria Gensim [20] viene scaricato il modello Word2Vec (visto in precedenza).

```
from gensim import Word2Vec, KeyedVectors

import numpy as np
import pandas as pd
import nltk
```

Con la seguente operazione viene caricato e visualizzato il Dataset.

```
import os.path
if not os.path.exists("reddit_worldnews.csv"):
    from urllib.request import urlretrieve
    urlretrieve("/bitbucket.org/michtor99/tesi/reddit_worldnews.csv",
        "reddit_worlnews.csv")
df = pd.read_csv("reddit_worldnews.csv", encoding="latin1")
df.head(10)
```

	time_created	date_created	up_votes	down_votes	title	over_18	author
0	1201232046	2008-01-25	3	0	Scores killed in Pakistan clashes	False	polar
1	1201232075	2008-01-25	2	0	Japan resumes refuelling mission	False	polar
2	1201232523	2008-01-25	3	0	US presses Egypt on Gaza border	False	polar
3	1201233290	2008-01-25	1	0	Jump-start economy: Give health care to all	False	fadi420
4	1201274720	2008-01-25	4	0	Council of Europe bashes EU&UN terror blacklist	False	mhermans
...	...	...	...	...	...	...	...
509231	1479816764	2016-11-22	5	0	Heil Trump : Donald Trump s alt-right white...	False	nonamenoglory
509232	1479816772	2016-11-22	1	0	There are people speculating that this could b...	False	SummerRay
509233	1479817056	2016-11-22	1	0	Professor receives Arab Researchers Award	False	AUSharjah
509234	1479817157	2016-11-22	1	0	Nigel Farage attacks response to Trump ambassa...	False	smilyflower
509235	1479817346	2016-11-22	1	0	Palestinian wielding knife shot dead in West B...	False	superislam

Figura 4.3: Le prime 10 righe del dataset.

Il Dataset presenta le seguenti colonne (come si può notare in Figura 4.3:

- **time_created**: *timestamp* unico della data della presentazione della notizia.
- **data_created**: data della notizia.
- **up_votes**: quante volte è stata votata positivamente la notizia.

- **down_votes**: quante volte è stata votata negativamente la notizia.
- **title**: il titolo della notizia.
- **over_18**: se è o meno adatto ai minorenni.
- **author**: l'autore dell'articolo.

L'unico dato testuale interessante è sicuramente il titolo da cui si possono ricavare importanti informazioni. Il testo, però, deve essere pre-processato in modo da garantire la rimozione di ambiguità e permettere al modello di generalizzare. A questo scopo viene scaricato il modulo `"punkt"` (della libreria `nltk`) per la *tokenitation*. Il risultato viene messo in un vettore da passare come parametro al modello *Word2Vec*.

```
nltk.download("punkt")
titles = df["title"].values()
newsVec = [nltk.word_tokenize(title) for title in titles]
```

A questo punto si può costruire il modello *Word2Vec* e testarlo con qualche operazione interessante. La funzionalità `most_similar` restituisce una lista di parole più vicine alla sequenza in input. Per esempio, si suppone di voler trovare la parola "queen" partendo dalle parole "king" e "man". Si applica una semplice sottrazione degli input per trovare il concetto di "royality", il quale viene sommato alla parola "woman" per trovare "queen".

```
model = Word2Vec(newsVec)
royality = model["king"] - model["man"]
queen = model["woman"] + royality
model.most_similar([queen])
```

oppure, equivalentemente, si può applicare al concetto di genere, per arrivare sempre alla parola "queen":

```
gender = model["woman"] - model["man"]
queen = model["woman"] + gender
model.most_similar([queen])
```

Entrambi danno lo stesso risultato, ovvero una lista di parole più simili (con affianco la probabilità che la parola in input e quella predetta siano dello stesso contesto) alla parola "regina" senza che essa sia stata inserita in input:

```
[('king', 0.9424921274185181),
 ('prince', 0.7830901145935059),
 ('King', 0.7793066501617432),
```

```
('monarchy', 0.7474737763404846),
('president', 0.7437374591827393),
('blogger', 0.7237984538078308),
('royal', 0.7165801525115967),
('ex-president', 0.7130229473114014)]
```

Una semplice operazione di somma è riuscita ad individuare nuovi contesti ed *embeddings*.

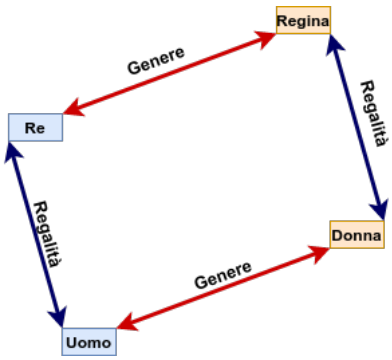

Figura 4.4: Esempio operazioni tra vettori

4.2.2 GloVe

A proposito di rappresentazioni vettoriali delle parole, nel 2014 viene presentato il modello **GloVe** (*Global Vector for Word Representation* [21]) che utilizza tecniche di *Latent Sentiment Analysis* con matrici di co-occorenza (matrice simmetrica nelle cui celle vi è il numero di volte che una parola all'interno di una finestra appare all'interno di un testo, in quanto parole con significati simili dovrebbero avere utilizzo simile). Ad esempio la seguente matrice, contiente delle probabilità che una parola sia corrispondente ad un determinato contesto:

Probability and Ratio	$k = SOLID$	$k = GAS$	$k = WATER$	$k = FASHION$
$P(k\|ice)$	$1.9 \cdot 10^{-4}$	$6.6 \cdot 10^{-5}$	$3.0 \cdot 10^{-3}$	$1.7 \cdot 10^{-5}$
$P(k\|steam)$	$2.2 \cdot 10^{-5}$	$7.8 \cdot 10^{-4}$	$2.2 \cdot 10^{-3}$	$1.8 \cdot 10^{-5}$
$P(k\|ice)/P(k\|steam)$	8.9	$8.5 \cdot 10^{-2}$	1.36	0.96

Come si può osservare nella tabella, le parole "ghiaccio" e "vapore" hanno una co-occorrenza maggiore con la parola condivisa "acqua", mentre hanno un numero di co-occorrenza minore con la parola "moda".

L'obiettivo formativo di GloVe è quello mappare le parole in uno spazio semanticamente correlato ad essa, in modo da apprendere vettori di parole. Per

fare ciò utilizza il loro prodotto scalare in modo che sia uguale al logaritmo della probabilità di co-occorrenza delle parole. Poiché il logaritmo di un rapporto è uguale alla differenza dei logaritmi, questo obiettivo associa il rapporto del logaritmo della probabilità della co-occorrenza alle differenze di vettori nello spazio vettoriale.

4.3 La traduzione

Partendo dalle trasformazioni lineari tra spazi vettoriali viste è possibile effettuare la traduzione da lingua a lingua. Le relazioni tra i termini nelle due lingue continueranno a valere:

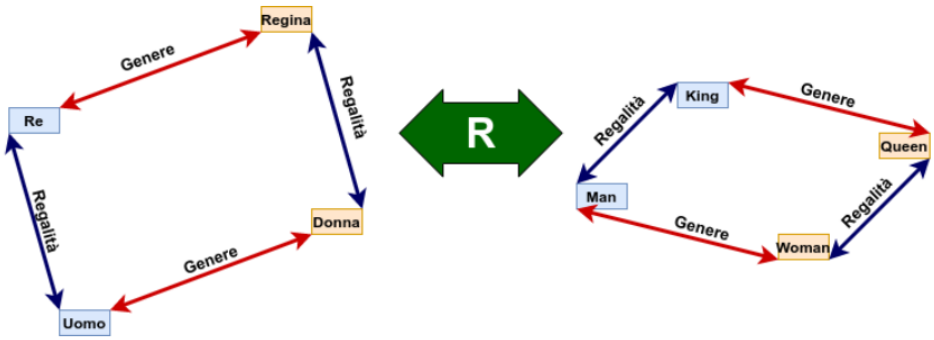

Figura 4.5: Traduzione come trasformazione lineare

Ogni parola di una lingua A corrisponde ad un vettore $x \in \mathbb{R}^n$, mentre nella lingua B il vettore sarà $y \in \mathbb{R}^m$. Nell'esempio si ha $n = m = 2$. La traduzione consiste, pertanto, nel associare un x di una lingua A ad un $\hat{y}$ che sia il più vicino possibile a y della lingua B. L'operazione si effettua applicando una trasformazione lineare, attraverso la moltiplicazione con un'opportuna matrice $R_{n \times m}$. Pertanto, ad esempio, la parola tradotta verrà trovata nel seguente modo:

$$x_{queen} \approx x_{regina} R \tag{4.19}$$

Per poter calcolare R è necessario avere k parole che abbiano una traduzione con i rispettivi embedding $x_1, ..., x_k$ per la lingua A e $y_1, ..., y_k$ per la lingua B. R deve poter minimizzare l'errore tra i vari embedding $x_i R$ e y_i. Il problema si può trovare come $XR \approx Y$, dove

$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_k \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_k \end{bmatrix} \tag{4.20}$$

Il problema del modello assomiglia molto ad un problema di minimizzazione di una funzione di costo. Si può, pertanto, applicare il metodo della discesa del gradiente, ponendo il gradiente analiticamente a zero:

$$X^T(XR - Y) = 0 \tag{4.21}$$

da cui:

$$R = (X^T X)^{-1} X^T Y \tag{4.22}$$

Ovviamente, al crescere di n diventa sempre più computazionalmente difficile trovare l'inversa di una matrice $n \times n$, quindi vengono solitamente usati embedding inferiori a 1000 componenti. Una volta trovata R, si può ottenere un vettore predetto $\hat{y} = Rx$ che sarà $\hat{y} \approx y$, il più vicino nella lingua B.

4.4 Autocorrezione

Un'altra della applicazioni più importanti di NLP è sicuramente l'autocorrezione dell'errore. Questa tecnica può essere integrata con gli algoritmi di *pre-processing*, ma esistono metodi differenti come la distanza di Levenshtein o metodi probabilistici. Questa applicazione risulta essere particolarmente importante nell'elaborazione del testo, ed è frequentemente usata digitazione da dispositivo mobile. Una delle tecniche più basilari per scoprire se vi è un errore di battitura è chiaramente verificare la presenza del token digitato tra i $|V|$ termine di un corpus di riferimento. Nel caso in cui la ricerca del termine non

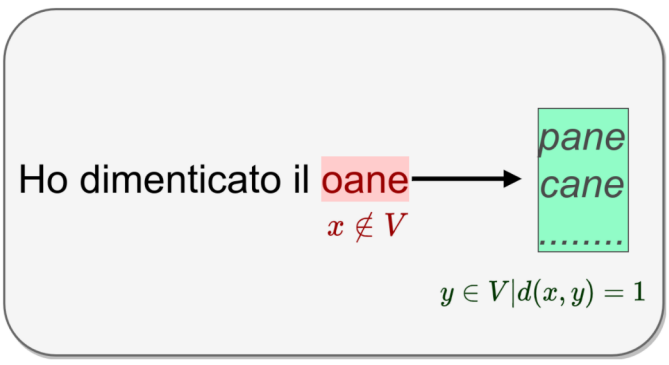

Figura 4.6: Esempio di autocorrezione

ha dato alcun risultato, viene generato un errore. Per correggerlo si procede con la generazione di possibili alternative in maniera iterativa. In particolare, si parte con la generazione di parole che hanno *distanza di edit* (numero di operazioni come inserimento, cancellazione, sostituzione, per trasformare un termine nel token presente nel corpus) pari a 1. Naturalmente, solo alcune delle

parole generate sono parole di senso compiuto e di queste si andrà a selezionare quelle che con più alta probabilità rispondono al contesto della restante parte della frase. La probabilità si calcola come frequenza relativa $P(w_k) = \frac{freq(w_k)}{|V|}$.

4.4.1 Distanza di Levenshtein

La distanza di edit, nota anche come distanza di Levenshtein, dal nome del suo inventore, offre una misura di differenza nei caratteri di due sequenze. Si supponga che le 3 operazioni abbiano costo computazionale unitario e si ipotizzi di avere le sequenze "neve" e "avere", rispettivamente di partenza e di destinazione. La matrice delle distanze verrà costruita nella maniera seguente:

	#	n	e	v	e
#	0	1	2	3	4
a	1	1	2	3	4
v	2	2	2	2	3
e	3	3	2	3	2
r	4	4	3	3	3
e	5	5	4	4	3

Tabella 4.3: Tabella distanze di Levenshtein

in cui una generica cella (i, j) è calcolata tenendo in considerazione la cella nella colonna precedente $(i, j-1)$, la cella della riga precedente $(i-1, j)$, la cella precedente in diagonale $(i-1, j-1)$. La distanza è calcolata secondo le tre operazioni:

- **Inserimento**: viene aggiunto un carattere alla substringa di destinazione calcolata dall'inizio fino alla colonna precedente, la distanza sarà $d(i, j-1)+1$
- **Cancellazione**: viene eliminato il carattere ne nella substringa di partenza dall'inizio fino alla riga precedente, la distanza sarà $d(i-1, j)+1$
- **Sostituzione**: viene sostituito un carattere rispetto allo stato della riga precedente e della colonna precedente. In particolare, se il carattere delle substringa di partenza è uguale a quello della substringa di destinazione, la distanza sarà $d(i-1, j-1)$, mentre se i caratteri sono diversi la distanza sarà $d(i-1, j-1)+1$, in quanto il nuovo carattere della stringa di destinazione è ottenuto sostituendo quello della stringa di partenza;

Nel caso trattato la distanza di Levenshtein è 3 che intuitivamente sono il numero di caratteri presenti sia nella sequenza di partenza sia nella sequenza di destinazione. [22]

Conclusioni e sviluppi futuri

Nello svolgimento di questa tesi sono stati toccati svariati argomenti nell'ambito di NLP e Deep Learning, soprattutto in maniera teorica.

Sono state viste le principali teorie che hanno portato allo stato dell'arte del Deep Learning per le applicazioni di *Natural Language Processing*, prendendo in considerazione la struttura di una rete neurale con memoria.

Successivamente sono state fornite le principali tecniche e *trick* di elaborazione del linguaggio naturale, pensando a come potrebbe apprendere il computer e a quali insidie potrebbero nascondersi dietro la forma di comunicazione umana più utilizzata.

Sono state esaminate le principali applicazioni e task di NLP, imparando le teorie che stanno dietro al loro funzionamento, e la logica delle loro strutture.

Con l'aggiunta della memoria, la macchina aggiunge un altro tassello nella somiglianza del cervello umano. Si tratta di un'evoluzione così ampia che neanche i film di fantascienza potevano prevedere. Questo pone lo sguardo anche sul futuro ed indaga su quali potranno essere le prossime scoperte ed invenzione in questo ambito.

Una volta studiata la teoria si potrà mettere in pratica tramite progetti e programmi atti all'apprendimento del linguaggio della macchina. Un'applicazione interessante che si potrebbe costruire con le tecniche e le teorie studiate in questa tesi è sicuramente la *chatbot*, che al giorno d'oggi sta prendendo sempre più piede e ha sempre più richiesta. La libreria *pipeline* dei Transformers propone, a questo proposito, un modulo domanda-risposta che potrebbe risultare particolarmente utile nel flusso della chat.

Ringraziamenti

Il primo ringraziamento va al relatore di questo lavoro, il Prof. Gianluca Moro, che ha reso possibile tutto ciò e ha acceso il mio personale interesse nei confronti di questa splendida disciplina. Ringrazio anche tutti i professori che mi hanno trasmesso le loro conoscenze e per avermi fatto appassionare al mondo dell'informatica in generale.

Il ringraziamento più grande va alla mia famiglia che mi ha sempre supportato e spronato a dare al massimo e che ha permesso la mia istruzione.

Un caloroso grazie ai miei amici di una vita, ma anche a quelli conosciuti in facoltà, che, con il loro entusiasmo, mi hanno aiutato a superare i momenti più difficili.

Bibliografia

[1] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, 2010.

[2] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[3] Cristiano Casadei. Le reti neurali ricorrenti. 2020. `https://www.developersmaggioli.it/blog/le-reti-neurali-ricorrenti/`.

[4] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[5] César Laurent, Gabriel Pereyra, Philémon Brakel, Ying Zhang, and Yoshua Bengio. Batch normalized recurrent neural networks, 2015.

[6] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[7] Gianluca Moro, Roberto Pasolini, Giacomo Domeniconi, and Vittorio Ghini. Deep neural trading: Comparative study with feed forward, recurrent and autoencoder networks. In Christoph Quix and Jorge Bernardino, editors, *Data Management Technologies and Applications - 7th International Conference, DATA 2018, Porto, Portugal, July 26-28, 2018, Revised Selected Papers*, volume 862 of *Communications in Computer and Information Science*, pages 189–209. Springer, 2018.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[9] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 3, pages 189–194. IEEE, 2000.

[10] Gianluca Moro., Andrea Pagliarani., Roberto Pasolini., and Claudio Sartori. Cross-domain & in-domain sentiment analysis with memory-based deep neural networks. In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management - KDIR,*, pages 127–138. INSTICC, SciTePress, 2018.

[11] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[13] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[14] Medium. inside-machine-learning. 2020. `https://medium.com/inside-machine-learning/what-is-a-transformer-d07dd1fbec04`.

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[17] Treccani. dizionario. `https://www.treccani.it/enciclopedia/linguaggio-naturale_%28Enciclopedia-della-Matematica%29/`.

[18] Christopher Manning, Richard Socher, Guillaume Genthial Fang, and Rohit Mundra. Cs224n: Natural language processing with deep learning1. 2017.

[19] Andrew J. Viterbi. Viterbi algorithm. *Scholarpedia*, 4(1):6246, 2009.

[20] Ichi Pro. Addestrare word2vec utilizzando gensim. `https://ichi.pro/it/addestrare-word2vec-utilizzando-gensim-22278944385252`.

[21] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[22] Enrico Giannini. Disordine. 2021. https://enricogiannini.com/8/word2vec-modelli-addestrabili-per/-la-rappresentazione-distribuita-delle-parole.