Uno studio sui Word Embeddings per documenti di ambito medico: Il caso di studio della collezione PubMed

Beschi Andrea

Relatore: Prof. Giorgio Maria Di Nunzio

Corso di Laurea Magistrale in

Ingegneria Informatica

2 luglio 2018 Anno Accademico 2017/2018

Ringraziamenti

Sono tante le persone che hanno reso possibile tutto questo, ognuno con un contributo diverso, ma al tempo stesso fondamentale.

Prima di tutto vorrei ringraziare il Professor Di Nunzio per avermi guidato nella stesura di questo lavoro di tesi.

Ringrazio la mia famiglia, i miei genitori Emanuela e Angelo, mia sorella Silvia e tutti i parenti, che sempre mi sono stati accanto, mi hanno sostenuto e hanno creduto in me e nel mio futuro, di cui questo traguardo costituisce solida fondamenta. Questa Laurea è anche vostra.

Ringrazio Anita, che mi ha accompagnato e soprattutto sopportato in questo percorso, e che spero continui a sopportarmi per tanto tempo ancora. Saperti accanto in questa giornata rende tutto più speciale.

Ringrazio i miei amici, compagni insostituibili con i quali in questi anni ho vissuto momenti indimenticabili.

Infine, vorrei ringraziare in modo particolare mia Nonna Elda, che tanto avrebbe voluto partecipare a questo traguardo. Persona fondamentale che tanto mi ha insegnato, che tanti valori mi ha trasmesso e che mai ha dimenticato di ricordarmi con affetto "Studia!". A lei è dedicata questa tesi.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati codice

Sommario

Questo elaborato ha come obbiettivo quello di studiare i Word Embeddings, in relazione all'applicazione di tale strumento alla collezione di ambito biomedicale di PubMed[1]. In particolare l'interesse è quello di sviluppare una versione del modello Skip-Gram [12] con la quale addestrare i Word Embeddings relativi alla collezione PubMed sfruttando una piattaforma con poche risorse di calcolo come può essere un Notebook o un Personal Computer.

Questa tesi si colloca quindi all'interno del campo di ricerca della Natural Language Processing, o Elaborazione del Linguaggio Naturale. In tale ambito il problema della rappresentazione delle parole ricopre grande importanza. Si tratta cioè di risolvere il problema di come le parole e le loro informazioni di carattere lessicale, sintattico e grammatico, possano essere codificate all'interno di particolari strutture che possono essere utilizzate per processare le informazioni testuali in modo automatico, in modo analogo con quello che viene effettuato su immagini e informazioni sonore. Le principali applicazioni pratiche in cui si ritrova questo problema sono ad esempio i sistemi di reperimento dell'informazione, la Sentiment Analysis, la traduzione automatica e gli assistenti vocali. Non è certamente un problema di banale soluzione: tale complessità è certamente dovuta in modo particolare alle caratteristiche intrinseche di ambiguità dei linguaggi naturali.

Una possibile soluzione a tale problematica è rappresentata dai Word Embeddings, un insieme di strumenti, modelli del linguaggi e tecniche di apprendimento che permettono la rappresentazione di parole e frasi attraverso l'uso di vettori a componenti reali. Attraverso l'utilizzo di tecniche di Machine Learning e Deep Learning è possibile mettere in pratica modelli che permettano in modo automatico e senza supervisione umana, dato un corpus di documenti testuali di addestramento, l'addestramento di tali Word Embeddings.

Nei capitoli seguenti verranno analizzati gli aspetti principali dei Word Embeddings nell'ambito del Natural Language Processing (NPL), le motivazioni che hanno portato al loro sviluppo e al recente successo in vari ambiti di ricerca. Verranno inoltre presentate alcune applicazioni dei Word Embeddings rese possibili dai recenti risultati ottenuti nell'ambito delle reti neurali, del Machine Learning (ML), in particolar modo del Deep Learning (DL). In seguito verrà presentato quello che è diventato il modello di riferimento per quanto riguarda i Word Embeddings, ovvero il modello Word2Vec proposto da Mikolov et al. in [12], con particolare attenzione al modello Skip-Gram. Di tale modello è stata sviluppata

[1] https://www.ncbi.nlm.nih.gov/pubmed/

un'implementazione in linguaggio Python che fa uso del framework di Machine Learning TensorFlow[2]. Il modello Skip-Gram è stato dunque applicato alla collezione PubMed, che comprende più di 28 milioni di citazioni da letteratura biomedicale, giornali di scienze naturali e libri online. Saranno presentate le metodologie seguite per l'ottenimento della collezione di addestramento, per il pre-processing di tale collezione, per la costruzione del modello Skip-Gram e il suo addestramento. Verranno infine analizzati i risultati ottenuti.

[2]https://www.tensorflow.org/

Capitolo 1

Natural Language Processing e Word Embeddings

In questo capitolo verranno presentati i concetti fondamentali del Natural Language Processing (NPL), con particolare attenzione agli aspetti relativi ai documenti scritti, ai cosiddetti Word Embeddings e alle loro applicazioni.

1.1 Natural Language Processing

Con il termine Natural Language Processing (detto anche NPL, o Elaborazione del Linguaggio Naturale) viene indicata un'area di ricerca che interseca informatica, intelligenza artificiale e linguistica che si occupa del processo di trattamento automatico delle informazioni scritte o parlate in una lingua naturale ad uno o più livelli linguistici con l'obbiettivo di ottenere un'elaborazione del linguaggio simile a quella umana per tutta una serie di applicazioni pratiche. Si tratta di un processo difficile e complesso a causa delle caratteristiche di ambiguità del linguaggio umano, tanto che il problema della perfetta comprensione del linguaggio è stato definito AI-completo [22], termine coniato in analogia con i termini NP-difficile e NP-completo dalla teoria della complessità, ad indicare quelle classi di problemi dell'intelligenza artificiale ritenuti più complessi.

La complessità dell'elaborazione del linguaggio naturale è da ritrovare principalmente in due fattori:

- Nel linguaggio naturale sono spesso presenti convenzioni e forme idiomatiche fortemente dipendenti dal contesto, anche in presenza di regole e strutture relativamente precise.

- La comunicazione umana è ridondante e ambigua.

Il processo di elaborazione del linguaggio naturale viene quindi suddiviso in fasi diverse, rispecchiando la struttura dell'elaborazione di un linguaggio di programmazione:

- **Analisi lessicale**: processo che si occupa della scomposizione delle sequenze di caratteri in ingresso nelle parole che le compongono. Tali parole vengono chiamate 'token'.
- **Analisi grammaticale**: processo che si occupa dell'associazione di ciascun token precedentemente estratto in una categoria lessicale. In questo caso la valutazione del contesto è fondamentale, in quanto ci sono parole che possono appartenere a categorie lessicale distinte in base al loro utilizzo in contesti diversi.
- **Analisi sintattica**: processo che ha l'obbiettivo di organizzare i token in una struttura sintattica utilizzando le regole di una certa grammatica.
- **Analisi semantica**: processo che si occupa di attribuire ai token un certo significato.

1.1.1 Storia del Natural Language Processing

La storia del Natural Language Processing viene solitamente fatta partire alla fine degli anni Quaranta del Novecento con il memorandum del matematico statunitense Warren Weaver dal titolo "Translation" nel quale veniva proposta per la prima volta l'idea dell'utilizzo di un calcolatore per la traduzione di un testo da una lingua naturale ad un'altra. Il Machine Translation (MT) [21] è stata dunque la prima applicazione informatica nel campo del NLP. I primi sistemi di questo tipo facevano la semplicistica assunzione che le differenze tra diverse lingue derivassero dalle parole e dal loro ordine. La traduzione in tali sistemi avveniva quindi utilizzando il look-up su dizionari e riordinando le parole tradotte per rispecchiare l'ordine della lingua nella quale si voleva tradurre, il tutto senza considerare l'ambiguità lessicale intrinseca del linguaggio naturale. Chiaramente questo tipo di sistemi produceva scarsi risultati, spingendo gli studiosi a indagare meglio la teoria del linguaggio. In questo periodo si svilupparono altre applicazioni, come il riconoscimento vocale. Tuttavia, a causa dell'inadeguatezza dei sistemi esistenti, della mancanza di dati facilmente fruibili e dalla limitatezza di risorse dei calcolatori disponibili, che portò al report dell'ALPAC(Automatic Language Processing Advisory Committee of the National Academy of Science) del 1966[1] nel quale si concludeva che la traduzione automatica non fosse raggiungibile in tempi brevi, ci fu un rallentamento e una graduale diminuzione di interesse per quest'area di ricerca. Negli anni Ottanta si ha invece una rapida crescita, anche e soprattutto perché vennero a mancare quegli ostacoli che ne avevano limitato lo sviluppo in passato [10]: aumentò infatti la disponibilità di testi elettronici di una certa dimensione, aumentarono le risorse in termini di potenza e memoria a disposizione dei calcolatori e ci fu la nascita di Internet. In particolare gli approcci di tipo statistico hanno avuto successo in vari problemi come nell'identificazione di parti del discorso, disambiguazione del senso delle parole, nei sistemi di reperimento dell'informazione e nei sistemi di Question-Answering.

[1] http://www.mt-archive.info/ALPAC-1966.pdf

1.1.2 Livelli del Natural Language Processing

Un modo intuitivo per capire cosa avviene all'interno di un sistema di Natural Language Processing è utilizzare il cosiddetto approccio per 'livelli del linguaggio':

- **Fonologia**: questo livello ha a che fare con l'interpretazione dei suoni del discorso all'interno e attraverso le parole. Sono presenti tre tipi di regole nell'analisi fonetica:

  1. regole fonetiche, per i suoni delle parole.
  2. regole fonemiche, per le variazioni della pronuncia quando le parole vengono utilizzate insieme.
  3. regole prosodiche, per l'oscillazione degli accenti e dell'intonazione all'interno di una frase.

  In un sistema di NLP che accetti il suono come input, le onde sonore vengono analizzate e codificate in un segnale digitale per essere interpretate attraverso le varie regole o comparate con un particolare modello che si vuole utilizzare.

- **Morfologia**: questo livello ha a che fare con la natura delle parole, che sono composte da morfemi, le più piccole unità di significato. Per esempio la parola *preautorizzazione* può essere morfologicamente analizzata in tre differenti morfemi: il prefisso *pre*, la radice *autorizz* e il suffisso *azione*. Dato che il significato di ogni singolo morfeme si mantiene, gli essere umani possono prendere una parola sconosciuta, scomporla nei morfemi per ottenere il suo significato. In modo analogo un sistema NLP può riconoscere il significato di ogni morfeme per ottenere una rappresentazione del significato delle parole.

- **Lessicale**: in questo livello un sistema NLP interpreta il significato di ogni singola parola. La natura della rappresentazione può variare secondo la teoria semantica utilizzata dal sistema. Questo livello può avere necessità di un dizionario più o meno complesso: nel caso più semplice può essere composto da coppie *(parola, parte del discorso)*, fino ad arrivare a casi più complessi dove troviamo informazioni sulla classe semantica delle parole, definizioni del senso nella rappresentazione semantica.

- **Sintattico**: questo livello pone l'attenzione sull'analisi delle parole in una frase con l'obbiettivo di rilevarne la struttura grammaticale. Il risultato di tale livello può essere una rappresentazione della frase che enfatizzi le relazioni di dipendenza tra le parole. Sono state sviluppate nel tempo vari tipi di grammatiche che possono essere utilizzate a questo scopo.

- **Semantico**: a questo livello viene determinato il possibile significato di una frase ponendo l'attenzione sulle interazioni tra i significati a livello delle parole. Questo livello di elaborazione può includere il processo di disambiguazione di parole con più significati.

- **Discorso**: il livello del discorso, a differenza di quello sintattico e semantico, si concentra sulle proprietà del testo come una singola entità che ottiene significato in base alle relazioni e alle connessioni tra le varie frasi che lo compongono.
- **Pragmatico**: questo livello ha a che fare con l'utilizzo del linguaggio nelle situazioni e utilizza il contesto e il contenuto del testo per la comprensione.

Nei moderni sistemi NLP si ha la tendenza ad implementare moduli per ottenere soprattutto i livelli più bassi dell'elaborazione appena esposta. Questo è dovuto ad una serie di ragioni: in primo luogo perché una particolare applicazione non richiede l'interpretazione a livelli più alti; inoltre i livelli più bassi sono stati indagati in modo più approfondito e quindi sono meglio implementati; infine, i livelli più bassi di tale schema hanno a che fare con le unità più piccole di analisi, per esempio morfemi, parole, frasi, che sono meglio governate da regole grammaticali, semantiche, lessicali.

1.1.3 Approcci al Natural Language Processing

Gli approcci al NLP possono essere divisi in quattro categorie: simbolico, statistico, connessionista e ibrido. I primi due approcci coesistono fin dalla nascita di questo campo di ricerca. I primi lavori di tipo connessionista appaiono all'inizio degli anni Sessanta. L'approccio simbolico ha dominato il campo del NLP fino agli anni Ottanta, quando l'approccio statistico ha preso finalmente piede grazie alla maggiore disponibilità di risorse computazionali e al bisogno di sviluppare sistemi che avessero a che fare con contesti reali.

Approccio simbolico

L'approccio simbolico esegue una approfondita analisi dei fenomeni linguistici basata su un'esplicita rappresentazione dei fatti attraverso schemi e algoritmi fondati su regole e dizionari sviluppati da essere umani esperti di tale campo. Un esempio di applicazione di tale approccio può essere quello dei sistemi basati sulla logica o su regole. Nei sistemi basati su logica la struttura del linguaggio è solitamente nella forma di proposizioni logiche, mentre i sistemi basati su regole consistono in una serie di regole, un motore di inferenza e della memoria di lavoro. Gli approcci di tipo simbolico sono stati utilizzati per diversi decenni [10] in una serie di aree di ricerca e applicazione come l'estrazione di informazione, la categorizzazione di testi, la risoluzione di ambiguità e acquisizione di lessico. Le tecniche più utilizzate comprendono l'apprendimento explanation-based, l'apprendimento basato su regole, programmazione logica induttiva, alberi decisionali, clustering, algoritmi K-nearest-neighbour.

Approccio statistico

L'approccio statistico utilizza una vasta gamma di tecniche matematiche e testi di grandi dimensioni per sviluppare un modello generalizzato dei fenomeni lin-

guistici basati su esempi reali di tali fenomeni messi a disposizione con i testi di addestramento senza l'aggiunta di ulteriore conoscenza linguistica. Uno dei modelli statistici più utilizzati è l'Hidden Markov Model (HMM) [3], un automa a stati finiti con un insieme di stati ai quali sono associate delle probabilità di transizione tra stati diversi. Gli approcci statistici vengono tipicamente utilizzati in applicazioni come lo speech recognition, l'acquisizione lessicale, il parsing e l'apprendimento di grammatiche.

Approccio connessionista

Si tratta di un approccio simile a quello statistico, in quanto troviamo nuovamente l'obbiettivo di sviluppare modelli generalizzati partendo da esempi di fenomeni linguistici. La differenza si trova nel fatto che l'approccio connessionista combina l'apprendimento statistico dei modelli statistici con varie teorie della rappresentazione. In generale, un modello connessionista è un'interconnessione di unità semplici di elaborazione. In letteratura troviamo [18]:

- **Modelli locali**, nei quali ogni unità rappresenta un particolare concetto. Le relazioni tra concetti sono codificate nei pesi delle connessioni tra tali concetti. La conoscenza in tali modelli è sparsa attraverso la rete, e le connessioni tra unità di elaborazione riflettono le loro relazioni strutturali.

- **Modelli distribuiti**, nei quali, a differenza dei modelli presentati in precedenza, i concetti sono rappresentati in funzione dell'attivazione simultanea di più unità di elaborazione.

1.1.4 Applicazioni del Natural Language Processing

Il Natural Language Processing trova applicazione in una vasta gamma di applicazioni: di fatto ogni applicazione che utilizzi testo è candidabile per l'elaborazione del linguaggio naturale. Le principali applicazioni sono:

- **Reperimento dell'informazione**, area che si occupa di gestire la rappresentazione, la memorizzazione, l'organizzazione e l'accesso ad oggetti contenenti informazioni come documenti testuali, pagine web, cataloghi online e oggetti multimediali. Lo scopo finale del reperimento dell'informazione è soddisfare il cosiddetto 'bisogno informativo dell'utente'. Un esempio di sistema di reperimento dell'informazione è costituito dai motori di ricerca, come Google[2], Yahoo[3], Bing[4].

- **Estrazione dell'informazione**, processo che si occupa dell'estrazione automatica di informazioni strutturali da documenti non strutturati o semistrutturati.

[2]https://www.google.it/
[3]https://it.yahoo.com/
[4]https://www.bing.com

- **Sistemi di Question-Answering**, che si occupano di rispondere in modo automatico ad una domanda espressa dall'utente in un linguaggio naturale.
- **Sentiment analysis**, processo con cui vengono automaticamente raccolte e interpretate le opinioni presenti in un testo, che può essere una recensione di un prodotto, di un film, di un evento. Task tipici di questo campo sono l'analisi di popolarità e di soggettività.
- **Traduzione automatica**, area che si occupa della traduzione di testi da una lingua naturale ad un'altra in modo automatizzato.
- **Assistenti vocali**, come Siri[5] di Apple, Google Assistant[6] di Google, Cortana[7] di Microsoft.

1.2 Word Embeddings

Con il termine Word Embeddings viene indicato un insieme di strumenti, modelli del linguaggio e tecniche di apprendimento all'interno dell'area del Natural Language Processing che permettono di rappresentare parole e frasi di testi scritti attraverso vettori di numeri reali. Si tratta quindi di una funzione:

$$V \to \mathbb{R}^D : w \mapsto \vec{w} \tag{1.1}$$

Una parola w del dizionario V viene quindi mappata in un vettore a valori reali $\vec{w}$ in uno spazio di dimensione D.

Prima di affrontare in modo approfondito l'argomento dei Word Embeddings, viene presentato brevemente il problema della rappresentazione delle parole.

1.2.1 La rappresentazione delle parole

Un problema fondamentale per tutte le applicazioni del NLP è quello di rappresentare le parole dei testi all'interno del calcolatore: abbiamo bisogno di rappresentare il testo scritto in modo tale da poterlo processare in modo automatico, in modo analogo a quanto avviene con le immagini e le onde acustiche, rappresentate da segnali analogici o digitali.

One-hot Vectors

L'approccio più semplice per la rappresentazione delle parole è quello di utilizzare i cosiddetti 'one-hot embeddings', attraverso cioè vettori di grandi dimensioni(tante quante la dimensione del dizionario) con componenti tutte uguali a zero, tranne per quella relativa all'indice della parola rappresentata. Per esempio,

[5]https://www.apple.com/it/ios/siri/
[6]https://assistant.google.com/intl/it_it/
[7]https://www.microsoft.com/it-it/windows/cortana

partendo da un dizionario $D = \{uomo, donna, ragazzo, ragazza, re, regina\}$ con $|D| = 6$ otteniamo i vettori in Figura 1.1.

	0	1	2	3	4	5
uomo	1	0	0	0	0	0
donna	0	1	0	0	0	0
ragazzo	0	0	1	0	0	0
ragazza	0	0	0	1	0	0
re	0	0	0	0	1	0
regina	0	0	0	0	0	1

Figura 1.1: Esempio di one-hot vectors

Si tratta di una rappresentazione molto semplice e facile da costruire, ma presenta tutta una serie di svantaggi che la rendono inutilizzabile in un sistema reale: innanzitutto la dimensione dello spazio vettoriale generato dipende dalla dimensione del dizionario, che solitamente è molto grande. Inoltre questo tipo di rappresentazione non fornisce informazioni sulle relazioni semantiche tra le diverse parole, dato che si tratta di vettori ortogonali tra loro, e che quindi non permettono l'estrazione di un valore di similarità.

Count Vector

Consideriamo un Corpus C composto da D documenti $\{d_1, d_2, ..., d_D\}$ e siano N i token estratti dal corpus che formano il dizionario. Registrando l'informazione sulla frequenza di ciascun token all'interno di ogni documento possiamo generare la matrice M composta dai Count vector: M ha dimensione DxN, e la riga i-esima contiene le frequenze del token all'interno del documento d_i, con $0 \leq i \leq D$, mentre la colonna j-esima contiene il Count vector relativo al token j-esimo, $0 \leq j \leq N$.

TF-IDF Vector

Il modello TF-IDF [11] è simile al precedente ed è quindi basato sulla frequenza delle parole all'interno del corpus in esame, ma invece di considerare le occorrenze di una parola in un singolo documento, si concentra sulle occorrenze all'interno dell'intera collezione. L'idea alla base è quella di penalizzare i pesi relativi a quelle parole che appaiono spesso, ma in tutti i documenti (ad esempio come gli articoli, i pronomi, le preposizioni semplici). Questa tipologia di parole non danno un'informazione precisa sul documento. Viene dato invece maggior peso a quelle parole che appaiono con una certe frequenza solo in alcuni documenti, particolarità che le rende portatrici di contenuto informativo per quel dato documento. Sia quindi:

- $TF(t, d) = \frac{\text{numero occorrenze termine t nel documento d}}{\text{numero di termini nel documento}}$
- $IDF(t) = \log \frac{N}{n}$, con N=numero documenti e n=numero documenti in cui appare t.

Il peso di ciascun termine viene ottenuto moltiplicando i valori di TF e IDF.

N-grams

Un approccio alternativo per la rappresentazione delle parole è costituito dal modello 'n-grams' [11], che invece di considerare le singole parole prende in considerazione sequenze contigue di n parole. Ad esempio, da un testo "il gatto è sul tavolo", utilizzando $n = 2$ otteniamo $\{(il, gatto), (gatto, è), (è, sul), (sul, tavolo)\}$. Un modello di questo genere permette di catturare la probabilità congiunta che una data sequenza di parole co-occorra all'interno del testo in analisi. Tuttavia, anche questo modello presenta degli svantaggi: quello più importante è sicuramente dovuto alla 'curse of dimensionality' [4]. Se ad esempio volessimo modellare la distribuzione congiunta di 5-grams con un dizionario di 100000 parole, il numero di combinazioni possibile sarebbe $100000^5 - 1 = 10^{25} - 1$, un numero decisamente troppo grande per essere utilizzabile in una applicazione pratica. La causa principale di questa 'curse of dimensionality' può essere ritrovata nel fatto che ci siamo concentrati su uno spazio discreto, e quindi il problema di generalizzazione non è di facile soluzione, e la modifica di anche una sola di queste variabili ha un impatto drastico sul valore della funzione che vogliamo stimare.

1.2.2 La rappresentazione distribuita delle parole: Word Embeddings

La soluzione alla 'curse of dimensionality' vista in precedenza può essere trovata passando da uno spazio discreto allo spazio continuo: l'approccio è quello di provare a rappresentare le parole di un testo non più attraverso vettori di grandi dimensioni a componenti discrete, ma attraverso vettori di dimensione inferiore, ma con componenti continue. E' questo il caso dei Word Embeddings, che sono quindi in grado di codificare le informazioni semantiche e sintattiche delle parole, dove l'informazione semantica ha a che fare con il significato, mentre quella sintattica ha a che fare con il ruolo all'interno della struttura del testo.
A questo punto è possibile sviluppare modelli linguistici e statistici che permettono, a partire da un corpus di documenti iniziale, di addestrare tramite reti neurali, tecniche di Machine Learning e Deep Learning questi vettori. La maggior parte di questi modelli cercano di ottimizzare una funzione di loss andando a minimizzare la differenza tra i valori della predizione e quelli reali.
In letteratura sono presenti vari approcci ai Word Embeddings, che possono essere classificati in base alla distribuzione dell'informazione delle parole [9]:

- Modelli paradigmatici, che pongono l'attenzione sul contesto in cui appaiono le parole nel testo.
- Modelli sintagmatici, che si focalizzano invece sulla co-occorrenza delle parole all'interno di un testo.

Una classificazione alternativa è basata sul metodo di generazione dei Word Embeddings:

- Modelli matrix factorization: basati sulla matrice di co-occorrenza delle parole. I Word Embeddings vengono generati tramite metodi di fattorizzazione e scomposizione di matrici.
- Modelli slide-window sampling: dove i dati in ingresso vengono generati da finestre che scorrono il testo per predirre il contesto.

Applicazioni dei Word Embeddings

Abbiamo visto nella sezione introduttiva sul Natural Language Processing varie applicazioni pratiche: in molte di queste trovano applicazione i Word Embeddings:

- **Analisi sintattica**, il cui obbiettivo è l'estrazione della struttura sintattica del testo.
- **Analisi semantica**, che cerca di mettere in relazione la struttura sintattica delle parole all'interno di un testo con il loro significato in modo indipendente dalla lingua utilizzata. In altre parole, cerca di mettere in risalto una qualche tipo di correlazione tra le parole. Ad esempio, il vettore('Madrid'-'Spagna'+'Francia') è 'vicino' al vettore('Parigi').
- **Part-of-Speech (POS) tagging**, ovvero l'analisi grammaticale del testo, dove si cerca di etichettare ogni parola in una determinata categoria (per esempio nome, avverbio, singolare, plurare, ecc..).
- **Named entity recognition**, dove l'obbiettivo è il riconoscimento di nomi di persone, organizzazioni, valori monetari, luoghi d'interesse.
- **Sentiment Analysis**, che ha l'obbiettivo di estrarre opinioni e sentimenti dal testo.
- **Traduzione automatica**, che cerca di sostituire il testo in una lingua con il testo corrispondente in un'altra lingua.

Capitolo 2

Deep Learning e Reti Neurali

In questo capitolo verranno presentati gli aspetti principali del Machine Learning, o apprendimento automatico, e del Deep Learning, campo del Machine Learning sul quale si basano le metodologie di apprendimento applicate nel modello per i Word Embeddings preso in considerazione in questo lavoro di tesi. Verranno inoltre presentati i concetti fondamentali delle reti neurali per poter introdurre i cosiddetti 'Neural Language Model', ovvero i modelli neurali del linguaggio.

2.1 Machine Learning

In questa sezione vedremo una introduzione al Machine Learning, per poter in seguito introdurre il Deep Learning. Verrà presentata una definizione di algoritmo di apprendimento automatico e verrà proposto l'esempio dell'algoritmo di regressione lineare. Infine verrà introdotta la metodologia di ottimizzazione basata sul gradiente.

2.1.1 Algoritmi di apprendimento

Un algoritmo di apprendimento automatico è un algoritmo che è in grado di imparare dai dati forniti in ingresso. Una possibile definizione formale di algoritmo di apprendimento è la seguente [6]: "Un algoritmo di apprendimento è in grado di imparare da una certa esperienza E rispetto a una classe di task T e misura di performance P, se la performance nei task in T, misurata secondo P, aumenta con l'esperienza E".

Vedremo adesso una descrizione dei vari componenti della definizione sopra enunciata.

Il task T

I task nel Machine Learning sono solitamente descritti in termini di come il sistema debba elaborare un esempio. Un esempio in questo caso è una collezione di feature che siano state quantitativamente misurate da un oggetto o un evento e che vogliamo che il sistema processi. Generalmente un esempio è rappresentato

attraverso un vettore $x \in \mathbb{R}^n$ dove ogni componente x_i rappresenta una feature. Per esempio, le feature di un'immagine sono i valori assunti dai pixel di tale immagine.
Attraverso il Machine Learning è possibile affrontare tutta una serie di task, alcuni dei più noti sono i seguenti:

- **Classificazione**: in questo task è richiesto di assegnare un qualche input ad una delle k categorie alle quali può appartenere. L'algoritmo deve quindi produrre una funzione $f : \mathbb{R}^n \to \{1, ..., k\}$. Quando $y = f(x)$, il modello assegna all'input descritto da x una categoria identificata dal codice numerico y. Esistono delle varianti del problema di classificazione, dove f produce una distribuzione di probabilità sulle classi. Un esempio di classificazione è il riconoscimento di oggetti, dove l'input sono delle immagini(generalmente descritte come un insieme di pixel) e l'output del sistema è un codice che identifica l'oggetto nell'immagine. Un altro esempio sono i sistemi di riconoscimento facciale.

- **Classificazione con input mancanti**: la classificazione diventa di più complicata risoluzione se non è garantito che sia fornita ogni misurazione del vettore di input. L'algoritmo di apprendimento non può quindi basarsi su una sola funzione che mappi gli input nelle categorie, ma deve imparare un insieme di funzioni: ogni funzione equivale alla classificazione di x con un diverso sottoinsieme degli input mancanti. Questa situazione si ritrova frequentemente nelle diagnosi mediche, dove alcuni tipi di esami sono dolorosi e invasivi, e quindi utilizzati solo nei casi necessari. Per definire un insieme così vasto di funzioni si può addestrare una distribuzione di probabilità sulle variabili rilevanti, e in seguito risolvere la classificazione marginalizzando le variabili mancanti. Con n variabili in input, possiamo ottenere 2^n funzioni di classificazione diverse, ma l'algoritmo ha bisogno di imparare una singola funzione che descriva la distribuzione di probabilità congiunta.

- **Regressione**: in questa tipologia di problema, è richiesta la predizione di un valore numerico dato un certo input. L'algoritmo di apprendimento deve quindi produrre una funzione $f : \mathbb{R}^n \to \mathbb{R}$. Questo task è simile alla classificazione, la differenza sta nel valore prodotto in output.

- **Trascrizione**: in questo task è richiesto al sistema di apprendimento automatico di osservare la rappresentazione relativamente non strutturata di un certo tipo di dati in ingresso e trascrivere l'informazione in una forma testuale discreta. Per esempio, nel riconoscimento ottico di caratteri (OCR, Optical Character Recognition), viene fornita in ingresso una immagine del testo e viene restituito in output il testo in forma di sequenza di caratteri. Il sistema di Google Street View[1] utilizza tecniche di Deep Learning per processare i numeri civici. Un altro esempio di applicazione è il riconoscimento

[1] https://www.google.it/streetview/

vocale, dove viene fornito in input l'informazione audio e viene prodotto in output il testo delle parole pronunciate.

- **Traduzione automatica**: nel problema della traduzione automatica viene fornito in ingresso la sequenza di simboli in una certa lingua, e l'algoritmo converte la sequenza in input in un'altra lingua.

- **Anomaly detection**: in questo task, l'algoritmo passa in rassegna una serie di eventi e oggetti e segnala alcuni di loro come inusuali o atipici. Un esempio di sistema di questo tipo è l'individuazione di frodi per le carte di credito: un sistema effettua il monitoraggio dei movimenti della carta, e se si accorge che alcuni di questi potrebbero non essere stati effettuati dal legittimo proprietario, emette un qualche tipo di allarme per notificare l'anomalia.

- **Sintesi e campionamento**: l'algoritmo di apprendimento automatico viene addestrato per produrre in uscita nuovi esempi simili a quelli forniti in ingresso. Questo tipo di sistema trova applicazione in quei casi dove c'è la necessità di produrre grandi quantità di contenuto, e produrre queste quantità a mano sarebbe troppo costoso. Per esempio nel campo dei Videogames, dove sarebbe impensabile che vengano prodotti a mano i singoli pixel di un oggetti di grandi dimensioni o di un paesaggio. Un altro esempio di applicazione è la sintesi del discorso, dove in input abbiamo un testo scritto e vogliamo che il programma produca in output l'audio di tale testo.

- **Denoising**: in questo tipo di task, viene fornito all'algoritmo di apprendimento automatico un esempio corrotto $\tilde{x} \in \mathbb{R}^n$ ottenuto con un qualche processo corruttivo da un esempio $x \in \mathbb{R}^n$. L'algoritmo deve quindi essere in grado di ricostruire l'esempio originale x partendo dalla sua versione corrotta $\tilde{x}$, o più in generale predire la distribuzione di probabilità condizionata $\mathrm{P}[x \mid \tilde{x}]$.

La misura di performance P

Al fine di valutare le 'abilità' di un algoritmo di apprendimento automatico, dobbiamo mettere a punto una misura quantitativa per misurarne la performance. Generalmente questo tipo di misura P è specifica per il tipo di task T messo in atto dal sistema.
Per problemi come la classificazione e la trascrizione viene utilizzata la misura di accuratezza del modello: si tratta del rapporto tra gli esempi per i quali il modello produce l'output corretto e gli esempi totali. Da questo tipo di misura possiamo ottenere una misura dell'error rate, ovvero il rapporto inverso, tra gli esempi con output non corretti.
Generalmente si è interessati a misurare le performance del sistema su un insieme di dati che non siano stati utilizzati per l'addestramento: abbiamo quindi l'insieme di training, per addestrare l'algoritmo e il modello, e l'insieme di test, per testarne le performance.

L'esperienza E

Gli algoritmi di apprendimento automatico possono essere classificati come non-supervisionati e supervisionati in base al tipo di esperienza che possono avere durante il processo di addestramento:

- **Algoritmi di apprendimento non-supervisionati**: l'addestramento viene effettuato su un dataset per impararne proprietà utili. In altre parole, vengono osservati diversi esempi di una variabile aleatoria x e si cerca quindi di impararne in modo implicito o esplicito la distribuzione di probabilità $\mathrm{P}[x]$ di una serie di proprietà interessanti.
- **Algoritmi di apprendimento supervisionati**: in questo caso l'addestramento viene effettuato su un dataset dove ogni esempio è associato ad una etichetta. Vengono cioè osservati vari esempi di una variabile aleatoria x e del valore o della variabile associata y, cercando di imparare a predire y da x, andando a stimare $\mathrm{P}[y \mid x]$.

2.1.2 Esempio: la regressione lineare

Per rendere più concreti i concetti enunciati nel paragrafo precedente sul Machine Learning, verrà presentato un esempio di un semplice algoritmo di Machine Learning: la regressione lineare.
Come suggerisce il nome, il problema è quello di risolvere una regressione: in altre parole, l'obbiettivo è prendere un vettore $x \in \mathbb{R}^n$ come input e predire il valore di uno scalare $y \in \mathbb{R}$ come output. L'output della regressione lineare è chiaramente una funzione lineare nell'input. Sia quindi $\hat{y}$ il valore di predizione di y del nostro modello. Possiamo definire l'output come:

$$\hat{y} = w^T x \tag{2.1}$$

dove $w \in \mathbb{R}^n$ è il vettore dei parametri.
I parametri sono i valori che controllano il comportamento del sistema. In questo caso, w_i è il coefficiente che viene moltiplicato con la feature x_i prima di essere sommato con i contributi delle altre feature. Possiamo pensare a w come un insieme di pesi che determinano quale sia l'importanza di ogni feature nel contribuire alla predizione del valore y. Se una feature x_i riceve un peso w_i positivo, allora incrementando il valore di quella feature andrà ad incrementarsi il valore della predizione $\hat{y}$, mentre con un valore del peso w_i negativo verrà decrementato il valore della predizione $\hat{y}$.
Abbiamo quindi una defizioni del task: predire y da x tramite $\hat{y} = w^T x$. Abbiamo adesso bisogno di una misura di performance P.
Supponiamo di avere a disposizione una matrice di m esempi di input che non vengono utilizzati per l'addestramento. Supponiamo inoltre di avere disposizione un vettore di regressione che fornisce il valore esatto di y per ognuno di questi esempi. Possiamo chiamare questo insieme 'Insieme di Test', e indichiamo la matrice come X^{test} e il vettore di regressione come y^{test}.

Un modo possibile di misurare le performance del nostro algoritmo di regressione lineare è calcolando l'errore quadratico medio (Mean Squared Error) del modello sull'insieme di test, dato da:

$$MSE_{test} = \frac{1}{m}\sum_i (\hat{y}^{test} - y^{test})_i^2 \tag{2.2}$$

In modo intuitivo possiamo osservare che la misura di errore descresce a 0 quando $\hat{y}^{test} = y^{test}$, ovvero quando la distaza euclidea tra predizione e target diminuisce.
Per permettere all'algoritmo di Machine Learning di migliorare i suoi risultati in termini di predizione, abbiamo bisogno di un meccanismo che migliori i pesi w in modo tale da ridurre il valore dell'errore quadratico medio MSE_{test} dopo aver osservato l'insieme di addestramento (X^{train}, y^{train}). Un modo molto intuitivo di ottenere questo risultato è quello di minimizzare l'errore quadratico medio dell'insieme di addestramento MSE_{train}: calcolando cioè il suo gradiente e ponendolo uguale a 0:

$$\nabla_w MSE_{train} = 0 \tag{2.3}$$

$$\nabla_w \frac{1}{m} \left\| \hat{y}^{train} - y^{train} \right\|_2^2 = 0 \tag{2.4}$$

$$\frac{1}{m}\nabla_w \left\| X^{train} w - y^{train} \right\|_2^2 = 0 \tag{2.5}$$

$$\nabla_w (X^{train} w - y^{train})^T (X^{train} w - y^{train}) = 0 \tag{2.6}$$

$$\nabla_w (w^T X^{(train)T} X^{train} w - 2 w^T X^{(train)T} y^{train} + y^{(train)T} y^{train}) = 0 \tag{2.7}$$

$$2X^{(train)T} X^{train} w - 2X^{(train)T} y^{train} = 0 \tag{2.8}$$

$$\Rightarrow w = (X^{(train)T} X^{train})^{-1} X^{(train)T} y^{train} \tag{2.9}$$

Normalmente nella regressione lineare è presente anche un termine noto b. Il modello diventa quindi:

$$\hat{y} = w^T x + b \tag{2.10}$$

Il termine b viene spesso chiamato parametro bias, nel senso che in assenza di input, l'output del modello sarà proprio il termine b.

2.1.3 Ottimizzazione basata sul gradiente

La maggior parte degli algoritmi di Machine Learning necessitano un qualche tipo di ottimizzazione. Con il termine ottimizzazione si intende il processo di minimizzare (o massimizzare) una certa funzione $f(x)$ modificando x. Solitamente i problemi di ottimizzazione vengono espressi in termini di minimizzazione di $f(x)$, in modo tale che l'equivalente problema di massimizzazione si ottiene minimizzando $-f(x)$.

La funzione che si cerca di minimizzare o massimizzare è detta funzione obbiettivo, e in particolare quando si tratta di problemi di minimizzazione viene indicata con funzione di costo, funzione di perdita o funzione di errore. Il valore della variabile x che minimizza o massimizza la funzione di costo viene indicato con $x^* = \arg\min f(x)$.
Supponiamo quindi di avere a disposizione una funzione $y = f(x)$, dove sia x che y sono valori reali. La derivata di tale funzione viene indicata con $f'(x)$ o con la notazione $\frac{dy}{dx}$. La derivata $f'(x)$ rappresenta la pendenza di $f(x)$ nel punto x. La derivata è dunque utile per minimizzare una funzione in quanto ci dice come variare x per ottenere un miglioramento in y. Possiamo quindi 'scorrere' la funzione y seguendo valori della derivata tali per cui la pendenza ci permetta di raggiungere i punti di minimo: questa tecnica viene indicata con il termine 'discesa del gradiente' (gradient descent). I punti nei quali la derivata vale zero, cioè $f'(x) = 0$ vengono chiamati punti stazionari. Un punto di minimo(massimo) locale è un punto dove $f(x)$ è minore(maggiore) rispetto a punti nelle vicinanze. Un punto di minimo(massimo) globale è il punto dove la funzione assume il suo valore minimo(massimo). Possono quindi esserci punti di minimo locale che però non sono ottimi a livello globale.
Nel contesto del Machine Learning e del Deep Learning ci troviamo spesso a dover modellare fenomeni tramite funzioni di più variabili del tipo $f : \mathbb{R}^n \to \mathbb{R}$. Per tali funzioni è stato sviluppato il concetto di derivate parziali: la derivata parziale $\frac{\partial}{\partial x_i} f(x)$ misura la variazione di f nel punto x al solo variare di x_i. Con il termine gradiente viene generalizzata la nozione di derivata al caso dove la derivata viene eseguita rispetto ad un vettore: il gradiente di f è un vettore le cui componenti sono le derivate parziali rispetto alle variabili e viene indicato con $\nabla_x f(x)$, dove l'i-esimo elemento del gradiente è la derivata parziale di f rispetto alla variabile x_i. In questo caso i punti stazionari sono quei punti dove ogni componente del gradiente è uguale a zero. La derivata direzionale nella direzione u (un vettore unità) è la pendenza della funzione f nella direzione di u.
Quindi, per minimizzare la funzione f, abbiamo bisogno di trovare la direzione nella quale f decresce, ovvero la direzione nella quale il gradiente risulta essere negativo. In questo modo possiamo far decrescere f procedendo per step e trovando nuovi punti dove calcolare il gradiente per il passo successivo:

$$x' = x - \epsilon \nabla_x f(x) \tag{2.11}$$

Dove ϵ è il passo di apprendimento, uno scalare positivo che determina la dimensione del passo.
Applicando questo approccio a modelli del linguaggio il cui obbiettivo è imparare buone rappresentazioni delle parole sotto forma di Word Embeddings, ci si trova nella necessità di effettuare derivate parziali di una funzione i cui input e output sono entrambi vettori. La matrice contenente tali derivate parziali è conosciuta come matrice Jacobiana: se abbiamo una funzione $f : \mathbb{R}^m \to \mathbb{R}^n$, allora la matrice Jacobiana $J \in \mathbb{R}^{n \times m}$ di f è definita come $J_{i,j} = \frac{\partial}{\partial x_j} f(x)_i$.
Ci sono inoltre determinate situazioni in cui possiamo essere interessati alla derivata della derivata, cioè alla derivata seconda, che rappresenta l'informazione

di come varia l'andamento della derivata prima al variare dell'input, misurando quindi la curvature della funzione che si sta studiando.
Dato che la funzione oggetto di studio può avere più dimensioni in input, ci sono una grande quantità di derivate seconde, che possono essere raccolte all'interno di una matrice detta matrice Hessiana. La matrice Hessiana $H(f)(x)$ è definita nel modo seguente:

$$H(f)(x)_{i,j} = \frac{\partial^2}{\partial x_i \partial x_j} f(x) \tag{2.12}$$

La matrice Hessiana è dunque la matrice Jacobiana del gradiente. Nei punti nei quali le derivate seconde sono continue, l'operatore derivata è commutativo e l'ordine di applicazione delle derivate parziali può essere invertito:

$$\frac{\partial^2}{\partial x_i \partial x_j} f(x) = \frac{\partial^2}{\partial x_j \partial x_i} f(x) \tag{2.13}$$

Questo fatto implica che nella matrice Hessiana $H_{i,j} = H_{j,i}$ e quindi è simmetrica in tali punti. La derivata seconda in una specifica direzione rappresentata da un vettore unità d è data da $d^T H d$. Quando d è un autovalore di H, la derivata seconda in quella direzione è data dall'autovalore corrispondente. Per le altre direzioni di d, la derivata seconda direzionale è una media pesata di tutti gli autovalori, con pesi compresi tra 0 e 1 e con gli autovettori che formano un angolo minore con d che ricevono un peso maggiore. L'autovalore massimo determina la derivata seconda massima, e l'autovalore minimo determina la derivata seconda minima. La derivata seconda direzionale ci dice quindi quanto bene può andare il passo di discesa del gradiente che stiamo per effettuare.
Uno schema generale per l'ottimizzazione di una funzione f(x) può quindi essere il seguente pseudocodice:

Algorithm 1 Pseudocodice del metodo di discesa del gradiente

$k = 0$
while $\nabla f(x_k) \neq 0$ **do**
  calcolare la direzione di discesa $p_k := -\nabla f(x_k)$
  calcolare il passo di apprendimento α_k
  $x_{k+1} = x_k + \alpha_k p_k$
  $k = k + 1$
end while

2.1.4 Stochastic Gradient Descent

Uno degli algoritmi fondamentali per i modelli di Machine Learning è lo 'Stochastic Gradient Descent' (SGD), che altro non è che un'estensione dell'algoritmo di discesa del gradiente visto nel paragrafo precedente.
Un problema ricorrente nell'ambito del Machine Learning consiste nel fatto che

per ottenere un buon grado di generalizzazione sono necessari degli insiemi di addestramento molto grandi, ma questo ha come diretta conseguenza un importante aumento dei costi computazionali per eseguire gli algoritmi di addestramento.
La funzione di costo utilizzata da un algoritmo di apprendimento si compone spesso di una somma effettuata sugli esempi di addestramento di una qualche funzione di loss. Per esempio, utilizzando la cosiddetta funzione 'Negative Log Likelihood' la funzione di costo diventa:

$$J(\theta) = \mathbb{E}_{x_{\hat{p}_{data}}}(-\log p(x;\theta)) = \frac{1}{m}\sum_{i=1}^{m} -\log p(x;\theta) \tag{2.14}$$

Per calcolare il gradiente di una funzione di questo tipo è richiesto il calcolo di:

$$\nabla_\theta J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \nabla_\theta - \log p(x;\theta) \tag{2.15}$$

Il costo computazionale di tale operazione è $O(m)$. Con il crescere della dimensione dell'insieme di addestramento fino a miliardi di esempi, il tempo per effettuare un singolo passo nell'algoritmo di discesa del gradiente può diventare molto grande, rendendo di fatto l'approccio impraticabile.
Questa problematica viene superata nel seguente modo: ad ogni passo dell'algoritmo, viene campionato un minibatch di esempi $\mathbb{B} = \{x_1, ..., x_m\}$ ottenuto in modo uniforme dall'insieme di addestramento. La dimensione del minibatch viene solitamente impostata per contenere un numero relativamente piccolo di esempi, che può variare da uno a poche centinaia. In questo modo il calcolo del gradiente non viene più effettuato sull'intero dataset di addestramento, ma su un suo piccolo sottoinsieme, riducendo la complessità di ogni singolo passo di aggiornamento a $O(1)$. Lo pseudocodice dell'algoritmo diventa quindi:

Algorithm 2 Pseudocodice dell'algoritmo Stochastic Gradient Descent

$k = 0$
while non si verifica la condizione di stop **do**
  campionare un minibatch di m campioni dal dataset $\{x_1, ..., x_m\}$
  calcolare la stima del gradiente $\hat{g}$ sugli m campioni
  Applicare l'aggiornamento: $\theta \leftarrow \theta - \alpha_k \hat{g}$
  $k = k + 1$
end while

2.1.5 Maximum Likelihood Estimation

Uno dei concetti fondamentali alla base delle funzioni utilizzate per le stime dei valori in output dei modelli di Machine Learning è il cosiddetto principio del 'Maximum Likelihood'.
Consideriamo un insieme di m esempi $\mathbb{X} = \{x_1, ..., x_m\}$ ottenuti in modo indipendente da una distribuzione generatrice $p_{data}(x)$. Sia quindi $p_{model}(x;\theta)$ una

famiglia parametrica di distribuzioni di probabilità definite su uno spazio indicizzato da θ. In altre parole, $p_{model}(x;\theta)$ mappa ogni configurazione x in un numero reale stimando la probabilità $p_{data}(x)$.
La stima Maximum Likelihood per θ è definita come:

$$\theta_{ML} = \arg\max_{\theta} p_{model}(\mathbb{X};\theta) \tag{2.16}$$

$$= \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x_i,\theta) \tag{2.17}$$

La produttoria effettuata su così tanti valori di probabilità può risultate impraticabile per vari ragioni matematiche. In modo tale da ottenere un problema di ottimizzazione equivalente ma più conveniente da trattare, viene preso il logaritmo della Likelihood, che non cambia l'andamento della funzione e soprattutto permette di riscrivere la produttoria come una sommatoria grazie alla proprietà dei logaritmi:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(x_i;\theta) \tag{2.18}$$

Dato che arg max non cambia se scaliamo la funzione di costo, possiamo dividere per un fattore m e ottenere una versione che esprima l'aspettazione rispetto alla distribuzione empirica $\hat{p}_{data}$ definita dai dati di addestramento:

$$\theta_{ML} = \arg\max_{\theta} \mathbb{E}_{x\sim\ \hat{p}_{data}} \log p_{model}(x;\theta) \tag{2.19}$$

Possiamo interpretare la Maximum Likelihood come il tentativo di ottenere nel modello una distribuzione uguale a quella che si può osservare nei dati di addestramento per il modello. Applicando questo principio alle necessità dei problemi di Machine Learning, possiamo utilizzare la Maximum Likelihood come funzione obbiettivo da massimizzare per ottenere i valori dei parametri θ ottimi. In modo totalmente equivalente per ottenere un problema di minimizzazione viene spesso utilizzata la 'Negative Log-Likelihood'.

2.2 Reti neurali

L'area di ricerca delle reti neurali è stata inizialmente ispirata dall'obbiettivo di modellare sistemi neurali biologici. In seguito l'attenzione si è spostata su aspetti più ingegneristici, fino ad arrivare ad essere un ottimo strumento per tecniche di Machine Learning. Per introdurre l'argomento cercheremo di fare un percorso simile.
L'unità computazionale di base di un cervello è il neurone. Nel cervello umano sono presenti in modo approssimativo 86 miliardi di neuroni, collegati tra loro da $10^{14} - 10^{15}$ sinapsi. Ogni neurone riceve segnali elettrici in ingresso dai suoi dendriti e produce segnali elettrici in output lungo il suo unico assone, che può

espandersi e connettersi attraverso le sinapsi ai dendriti di altri neuroni. Possiamo osservare questo tipo di struttura nella figura 2.1.

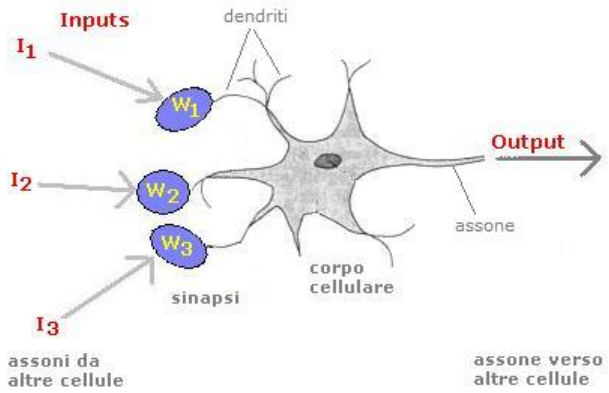

Figura 2.1: La struttura del neurone

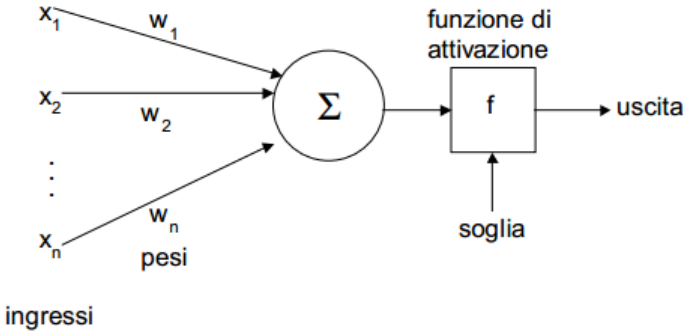

Figura 2.2: La struttura del neurone artificiale

Partendo da questa struttura è stato sviluppato il modello computazionale del neurone, rappresentato in Figura 2.2[2]: i segnali x_i che viaggiano lungo gli assoni vengono combinati con i dendriti di altri neuroni in base al peso w_i. L'idea alla base è che i pesi W dei dendriti possano essere addestrati e quindi utilizzati per controllare il comportamento dei neuroni associati. I contributi dei vari dendriti in ingresso vengono quindi sommati: se il valore di tale somma supera una certa soglia, il neurone può attivarsi e generare quindi un segnale in uscita attraverso il suo assone. Questa attivazione del neurone viene modellata tramite una funzione di attivazione f, che rappresenta la frequenza con cui viene generato un segnale in uscita: sono disponibili in letteratura una grande varietà di funzioni di attivazione che permettono di modellare i vari comportamenti che si vogliono associare al neurone. In altre parole, il neurone effettua un prodotto scalare tra input e pesi,

[2]http://dadf.altervista.org/blog/rete-neurale/

aggiunge un valore di bias e applica la funzione di attivazione. Le funzioni di attivazione più importanti e utilizzate sono:

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$, è una funzione a valori reali che restituisce un valore compreso tra 0 e 1. Per questo motivo può essere interpretata come una funzione che restituisce la probabilità con la quale il neurone si attiverà, dato un certo ingresso. L'andamento della funzione è mostrato in Figura 2.3.

- **tanh**: $\tanh(x) = 2\sigma(2x) - 1$. Si tratta di una versione scalata della funzione sigmoid, a valori reali nell'intervallo $[-1, 1]$ e centrata nello zero. Un grafico della funzione è mostrato in Figura 2.4.

- **ReLU**: $f(x) = \max(0, x)$. Viene quindi di fatto aggiunta una soglia a zero. La funzione è mostrata in Figura 2.5.

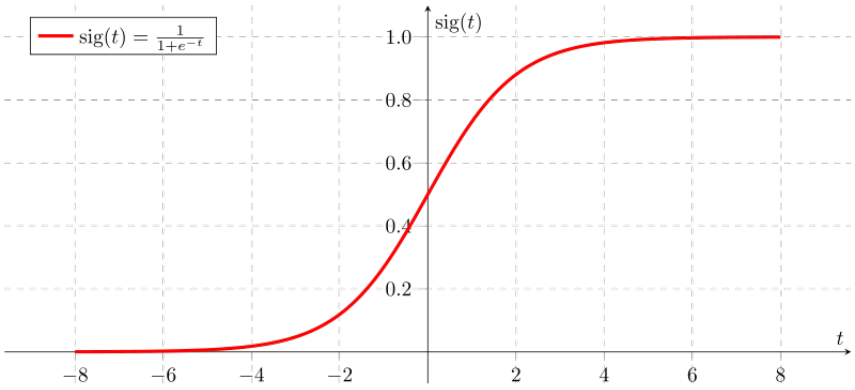

Figura 2.3: La funzione sigmoid

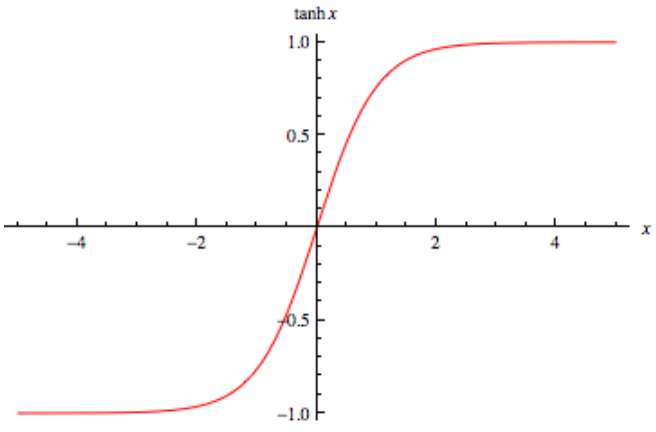

Figura 2.4: La funzione tanh

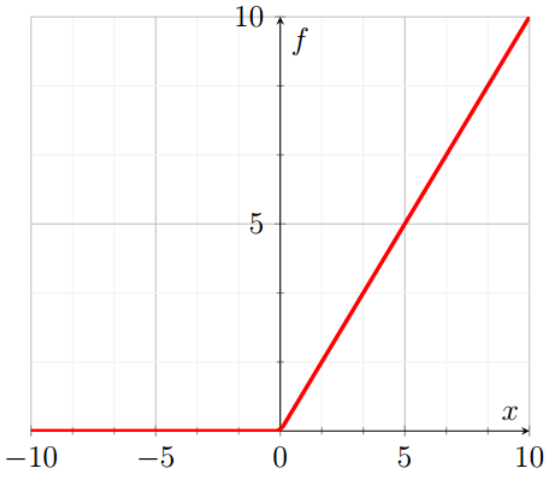

Figura 2.5: La funzione ReLU

Una rete neurale è quindi una combinazione di neuroni, organizzati in un certo modo: è possibile rappresentarla attraverso un grafo, dove gli output di certi neuroni diventano gli input di altri neuroni. Non sono ammessi cicli all'interno del grafo, che altrimenti porterebbero a loop infiniti. Solitamente i neuroni vengono organizzati in vari strati (layer): nel caso più tipico delle reti neurali regolari, i neuroni sono completamente connessi ai neuroni degli strati adiacenti.

Per descrivere una rete neurale si usa solitamente:

- Il numero di strati, escluso il primo strato di input: una rete a N-strati indica quindi una rete neurale con N-1 strati nascosti (hidden layer), uno strato di input e uno strato di output. Lo strato di output solitamente non possiede una funzione di attivazione, ma viene utilizzato per rappresentare gli output del sistema.
- Il numero di neuroni, o più comunemente il numero di parametri da addestrare: i pesi e i biases.

Le attuali architetture possono contenere circa 100 milioni di parametri e sono organizzate in 10-20 strati (da qui il termine 'Deep Learning').

2.3 Deep Learning

Con il termine Deep Learning si va ad indicare un campo del Machine Learning con il quale si cerca di risolvere problemi come quelli visti nella sezione precedente andando ad addestrare gli algoritmi processando un certo dataset di training. L'idea alla base è quella di provare a capire e rappresentare il mondo reale come una gerarchia di concetti, dove ogni concetto è definito tramite le sue relazioni con concetti più semplici. Questo tipo di approccio non richiede quindi che un operatore umano definisca in modo formale la conoscenza che l'algoritmo necessita. L'organizzazione gerarchica dei concetti che definiscono il problema permette

all'algoritmo di imparare concetti complessi partendo da concetti più semplici. Rappresentando in un grafo questa gerarchia di concetti otteniamo un grafo 'profondo', con molti strati: da ciò deriva il nome 'Deep Learning'.
I modelli di Deep Learning sono sostanzialmente rappresentati sotto forma di reti neurali feed-forward. L'obbiettivo di una rete di tipo feed-forward è approssimare una certa funzione f^*. Per esempio, in un classificatore $y = f^*(x)$ mappa l'input x in una categoria y. Una rete di questo genere definisce una funzione $y = f(x;\theta)$ e addestra i valori dei parametri θ. Questo tipo di modello è detto feed-forward perchè l'informazione fluisce attraverso la funzione venendo valutata da x, attaverso i calcoli intermedi utilizzati per definire f e infine nell'output y. Componendo tra loro le varie funzioni che descrivono il modello si ottiene dunque una rete, associata con un grafo diretto aciclico che descrive l'ordine con il quale vengono composte. Per esempio, supponiamo di avere tre funzioni $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ che vanno a formare $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. In questo particolare esempio, $f^{(1)}$ rappresenta il primo strato, $f^{(2)}$ rappresenta il secondo, e così via. La lunghezza di questa catena definisce la *profondità* del modello (e come sottolineato in precedenza, il nome Deep Learning deriva proprio da questo tipo di terminologia). Lo strato finale di una rete feed-forward è detto strato di output (output layer). Nel processo di addestramento del modello, l'obbiettivo è quello di avvicinare il più possibile i valori di $f(x)$ e di $f^*(x)$. I dati di addestramento mostrano il comportamento che dovrebbe avere lo strato di output, ma non specificano in modo diretto quello che deve accadere negli strati intermedi: l'algoritmo di apprendimento deve decidere come utilizzare questi strati per ottenere l'output desiderato senza l'informazione intermedia. Dato che i dati di addestramento non mostrano l'output desiderato per questi strati intermedi, essi vengono indicati con il termine strati nascosti (hidden layer). Infine, queste reti sono reti neurali perchè utilizzano il modello mostrato nella precedente sezione sulle reti neurali.

2.3.1 Algoritmo di Back-Propagation

Quando utilizziamo una rete neurale feed-forward con input x e output $\hat{y}$, l'informazione fluisce attraverso la rete. L'input x fornisce l'informazione iniziale che viene poi propagata attraverso gli strati nascosti fino allo strato di output che produce $\hat{y}$. Questo processo è quello che viene chiamato *forward propagation*.

Durante l'addestramento della rete, la forward propagation può continuare fino a che non viene prodotto un costo scalare $J(\theta)$, funzione del'insieme dei parametri θ. L'algoritmo di Back-Propagation [6] permette all'informazione sul costo di viaggiare a ritroso nella rete in modo tale da poter calcolare il gradiente: attraverso questo algoritmo è quindi possibile semplificare i calcoli per ottenere il gradiente. In questa sezione in particolare descriveremo il processo per il calcolo del gradiente $\nabla_x f(x;y)$ per una funzione arbitraria f, dove x è l'insieme delle variabili per le quali sono richieste le derivate, e y è un insieme addizionale di variabili in input ma per le quali non sono richieste le derivate. Negli algoritmi di apprendimento siamo in particolare interessati al gradiente della funzione

di costo $J(\theta)$ rispetto ai valori dei parametri θ. Nel nostro caso particolare restringiamo il campo a quelle funzioni che hanno come output un singolo valore scalare. Introduciamo ora alcuni concetti importanti per descrivere l'algoritmo di Back-Propagation.

Grafi computazionali

Nel contesto di questa trattazione sull'algoritmo di Back-Propagation, ad ogni nodo del grafo computazionale è associata una variabile. Le variabili possono essere scalari, vettori, matrici, tensori, o variabili di altro tipo. Introduciamo inoltre il concetto di operazione: un'operazione è una semplice funzione di una o più variabili che restituisce un singolo valore in output. Il grafo computazionale [6] è quindi composto da nodi che rappresentano gli input e da nodi che rappresentano le operazioni. L'output di una certa operazione viene etichettato con il valore di tale calcolo e viene disegnato un arco diretto dal nodo operazione che produce l'output al nodo operazione che lo riceve come input per effettuare l'operazione successiva.

Regola della catena

La regola della catena è utilizzata per il calcolo di derivate di funzioni composte. L'algoritmo di Back-Propagation di fatto applica in un modo molto efficiente questa regola molto semplice.

Sia $x \in \mathbb{R}$ un numero reale, e siano f e g due funzioni tali che $f, g : \mathbb{R} \to \mathbb{R}$. Supponiamo che $y = g(x)$ e che $z = f(g(x)) = f(y)$. Allora la regola della catena afferma che:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.20}$$

Passando al caso in cui le variabili in input siano dei vettori: supponiamo che $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$. Se $y = g(x)$ e $z = f(y)$, allora:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{2.21}$$

In notazione vettoriale:

$$\nabla_x z = (\frac{\partial y}{\partial x})^T \nabla_y z \tag{2.22}$$

Dove $\frac{\partial y}{\partial x}$ è la matrice Jacobiana di dimensione $n \times m$ di g.

Applicazione ricorsiva della regola della catena per ottenere la Back-Propagation

L'algoritmo di Back-Propagation si basa esattamente su questa semplice osservazione: una volta che abbiamo ottenuto l'output della rete tramite il passo di

forward propagation in cui dai nodi in input siamo arrivati allo strato di output, possiamo ricorsivamente applicare la regola della catena vista nel paragrafo precedente per calcolare tutti i gradienti necessari ad ogni singolo strato della rete, in modo tale da generare i segnali di errore e propagarli all'indietro fino ai parametri del modello e poter quindi applicare gli aggiornamenti indicati dal gradiente e continuare con il successivo passo di ottimizzazione.
Supponiamo che un certo nodo operazione contenga dunque la funzione $h = f(z)$, possiamo quindi memorizzare al suo interno le seguenti informazioni:

- un gradiente 'locale': si tratta del gradiente $\frac{\partial h}{\partial z}$, ovvero del gradiente dell'output rispetto all'input. Se il nodo operazione possiede più nodi input viene calcolato un gradiente locale rispetto a ciascuna variabile in input.
- un gradiente 'upstream': si tratta del gradiente $\frac{\partial s}{\partial h}$, ovvero del gradiente del valore propagato all'indietro dal nodo successivo a quello corrente, calcolato rispetto all'output h.
- un gradiente 'downstream': si tratta del gradiente $\frac{\partial s}{\partial z}$ ovvero del gradiente del valore propagato all'indietro dal nodo successivo a quello corrente calcolato rispetto all'input z. Per la regola della catena $\frac{\partial s}{\partial z} = \frac{\partial s}{\partial h}\frac{\partial h}{\partial z}$, che altro non sono che valori già calcolati in precedenza e disponibili. Questo semplifica molto i calcoli per ottenere i gradienti per effettuare l'aggiornamento dei parametri del modello.

La situazione appena descritta è mostrata in Figura 2.6.

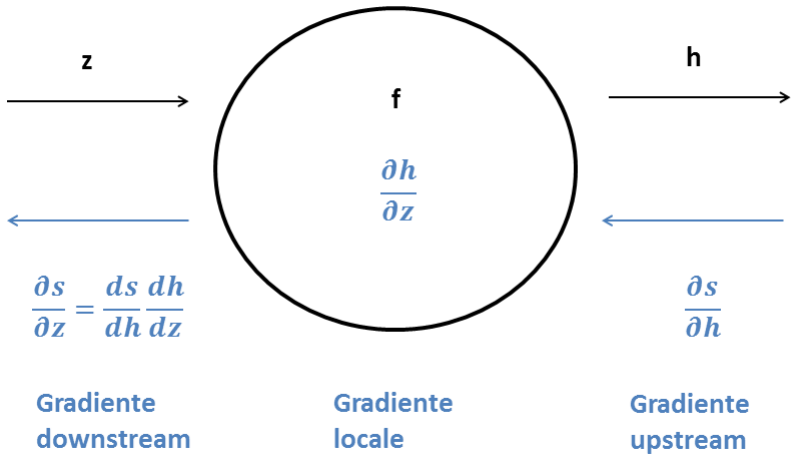

Figura 2.6: Esempio del passo di Back-Propagation in un nodo operazione

In conclusione, la metodologia appena presentata si compone di due passi fondamentali:

- passo forward: in questa fase iniziale vengono effettuati i calcoli per propagare l'informazione dagli input allo strato di output. In ogni nodo vengono effettuate le operazioni relative e salvati i valori intermedi.

- passo backward: in modo ricorsivo viene applicata la regola della catena per il calcolo dei gradienti intermedi per ottenere il valore del gradiente della funzione di costo relativo ai parametri del modello per effettuare l'aggiornamento e proseguire con un nuovo passo di addestramento.

2.4 Modello del linguaggio basato su reti neurali

Nel capitolo precedente abbiamo visto una introduzione al campo dell'elaborazione del linguaggio natuale, dei Word Embeddings e delle loro applicazioni. Una applicazione molto importante è il loro utilizzo all'interno di modelli del linguaggio basati su reti neurali: i Word Embeddings mettono a disposizione la loro capacità di rappresentare le parole sotto vari aspetti, e le reti neurali vengono utilizzate per addestrare questi vettori utilizzando corpus di grandi dimensioni.
L'idea alla base è la seguente: le parole che all'interno di un testo si trovano nello stesso contesto è molto probabile che condividano un qualche tipo di significato (Harris 1954). La probabilità che una sequenza di parole W si verifichi può quindi essere formulata tramite la regola di Bayes:

$$\mathrm{P}[W] = \prod_{t=1}^{N} \mathrm{P}[w_t \mid w_1, ..., w_{t-1}] = \prod_{t=1}^{N} \mathrm{P}[w_t \mid h_t] \tag{2.23}$$

Dove $\mathrm{P}[W]$ è la distribuzione congiunta della sequenza W, e h_t rappresenta le parole contesto attorno alla parola w_t. L'obbiettivo è di valutare la probabilità che la parola w_t si verifichi data l'informazione sul suo contesto.
La maggior parte dei modelli di questo tipo appartengono alla classe dei modelli non-supervisionati.
Tipicamente l'obbiettivo del modello del linguaggio basato su rete neurale è quello di massimizzare o minimizzare una certa funzione di costo, che molto spesso si tratta di una funzione di Log-Likelihood.
Supponiamo di avere una rete neurale feed-forward come quella presentata in [4]: data una sequenza $w_1, w_2, ..., w_n$ di parole nel corpus, siamo interessati a massimizzare la log-likelihood di $\mathrm{P}\left[w_t \mid \tilde{w}_{t-n+1}^{t-1}\right]$ nella rete neurale rappresentata in Figura 2.7. Sia x il vettore delle parole, tale che:

$$x = [e(w_{t-n-1}, ..., e(w_{t-2}), e(w_{t-1})] \tag{2.24}$$

Dove $e(w)$ rappresenta l'embedding della parola w. Abbiamo quindi un layer nascosto, e la funzione di output è:

$$y = b + U(\tanh(d + Wx)) \tag{2.25}$$

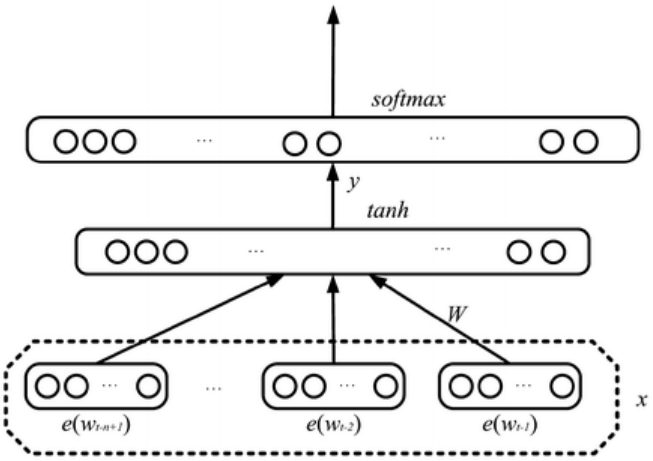

Figura 2.7: La struttura della rete neurale feed-forward

Dove U è la matrice di trasformazione, W la matrice dei pesi, b e d sono vettori bias. Infine y viene convogliata in un layer softmax per ottenere la probabilità della parola target. I parametri θ in questo tipo di modello sono (b, U, d, W). Tuttavia, dovendo utilizzare la funzione di softmax per effettuare la normalizzazione sull'intero vocabolario, se questo cresce di dimensione il costo computazionale per l'addestramento e il testing di tale modello potrebbe diventare molto oneroso.
Un modello alternativo a quello appena presentato è il modello Word2Vec proposto da Mikolov et al. in [12] e poi ulteriormente analizzato in [13]. Nei due articoli vengono presentati due framework: Skip-gram e CBOW(Continuous Bag-of-Words). La semplicità dei modelli e l'utilizzo di vari stratagemmi per mantenere i costi computazionali bassi lo hanno reso il modello di riferimento nel campo dei Word Embeddigs. La presentazione di tale modello sarà oggetto del prossimo capitolo.

Capitolo 3

Word2Vec: il modello Skip-gram

In questo capitolo viene presentato il modello Word2Vec, proposto da Mikolov et al. in [12] e [13]. In particolare, nel primo articolo gli autori propongono due architetture per l'addestramento di Word Embeddings di buona qualità da collezioni di documenti molto grandi composte da miliardi di parole con dizionari composti da milioni di parole, con una particolare attenzione alla complessità computazionale. Tali modelli sono il Continuous Bag-of-Words (CBOW) e lo Skip-Gram (SG). Nel secondo articolo sono invece presentati diversi accorgimenti per il miglioramento nelle prestazioni del modello Skip-Gram, attravero l'applicazione di Hierarchical Softmax, Negative Sampling e di subsampling di parole frequenti.

3.1 Introduzione ai modelli CBOW e SG

L'obbiettivo dei modelli Continuous Bag-of-Words e Skip-Gram è quello di addestrare i Word Embeddings tramite l'utilizzo di reti neurali. Tale addestramento viene effettuato cercando di ottimizzare una funzione obbiettivo che altro non è che il calcolo della probabilità di predizione di una certa parola centrale o di un insieme di parole contesto all'interno di una finestra di dimensione $2D$. Se dunque l'input del sistema è costituito da un documento di lunghezza L, sia w_t la parola centrale corrente, $0 \leq t \leq L-1$, mentre le parole contesto relative sono w_{t-j}, dove $1 \leq j \leq D$ e $-D \leq j \leq -1$. Possiamo osservare tale schema in Figura 3.1:

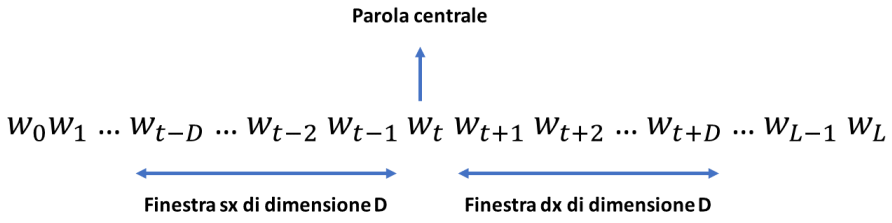

Figura 3.1: Parola centrale w_t e relativa finestra contesto

Abbiamo quindi due possibili approcci, che sono alla base dei due modelli che verranno presentati:

- **Modello CBOW**: viene effettuata la predizione della parola centrale w_t data la Bag-of-Words delle parole contesto w_{t-j}. Con il termine Bag-of-Words in letteratura [8] si viene a indicare il multi-insieme composto dalle parole w_{t-j}, dove viene ignorato l'ordine con cui tali parole compaiono nel testo ma viene mantenuta l'informazione sulla loro molteplicità.

- **Modello Skip-Gram**: viene effettuata la predizione delle parole contesto w_{t-j} data la parola centrale w_t.

La predizione viene effettuata applicando la funzione Softmax $\mathbb{R}^n \to \mathbb{R}^n$:

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \tag{3.1}$$

Tale funzione Softmax mappa dei valori arbitrari x_i in una distribuzione di probabilità p_i, dato che i valori vengono normalizzati tra 0 e 1. In particolare il calcolo della funzione Softmax viene effettuato sul valore del prodotto scalare tra i vettori della parola centrale e delle parole contesto. Ad ogni parola w corrispondono quindi due vettori:

- il vettore v_w quando w è parola centrale.

- il vettore u_w quando w è parola contesto.

I parametri dei modelli sono dunque composti dai vettori centrali e contesto delle parole del dizionario. Se d è la dimensione dei vettori, V la dimensione del dizionario, allora i parametri del modello costituiscono una matrice $\theta \in \mathbb{R}^{2dV}$. L'addestramento avviene in questo modo: iterativamente viene fatto scorrere il documento di lunghezza L e ad ogni passo viene costruita la finestra relativa alla parola centrale corrente, vengono calcolati i valori di probabilità che vengono utilizzati per il calcolo dei gradienti tramite l'algoritmo di BackPropagation che abbiamo visto nel capitolo precedente. I valori dei gradienti vengono quindi utilizzati per l'aggiornamento dei parametri del modello utilizzando lo Stochastic Gradient Descent visto nel capitolo precedente. Nelle sezioni seguenti vedremo più nel dettaglio i modelli sopra elencati.

3.2 Modello CBOW

Il modello Continuous Bag-of-Words si occupa dell'addestramento di Word Embeddings attraverso l'ottimizzazione di una funzione obbiettivo che calcola la probabilità della parola centrale corrente data la Bag-of-Words delle parole contesto. Lo schema di tale modello è rappresentato in Figura 3.2:

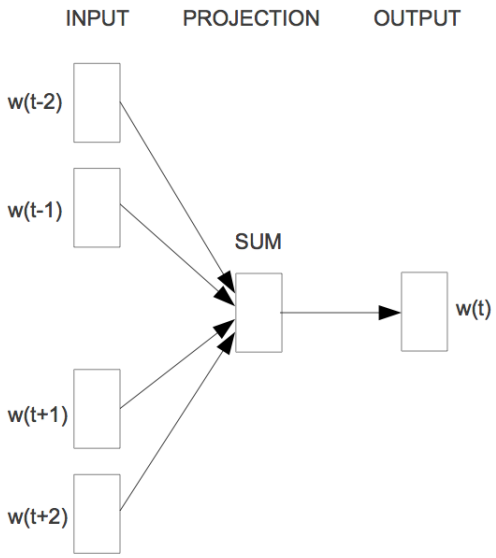

Figura 3.2: Schema del modello CBOW

Lo strato di input dello schema è composto dalla Bag-of-Words dei vettori one-hot $\{x_{t-D}, ..., x_{t-1}, x_{t+1}, ..., x_{t+D}\}$ relativi alle parole contesto $\{w_{t-D}, ..., w_{t-1}, w_{t+1}, ..., w_{t+D}\}$ per una finestra di dimensione $2D$. Lo strato nascosto è composto da un vettore h di dimensione d. Lo strato di output è composto dal vettore one-hot y della parola centrale oggetto della predizione w_t. I vettori one-hot in input sono connessi allo strato nascosto tramite una matrice di pesi $W \in \mathbb{R}^{V \times d}$ che è quindi la matrice composta dai vettori relativi alle parole contesto, mentre lo strato nascosto è collegato allo strato di output tramite una matrice di pesi $W' \in \mathbb{R}^{d \times V}$, ovvero la matrice composta dai vettori relativi alle parole centrali.

3.2.1 Forward Propagation in CBOW

L'output viene generato dall'input nel modo seguente: il valore del layer nascosto h viene calcolato come:

$$h = \frac{1}{2D} W \cdot \sum_{-D \leq j \leq D, j \neq 0} x_{t-j} \tag{3.2}$$

Viene dunque effettuata una media dei vettori in ingresso pesati dalla matrice W. Gli input per il calcolo del layer di output sono quindi:

$$u_i = (v'_{w_i})^T \cdot h \tag{3.3}$$

Dove v'_{w_i} è la i-esima colonna della matrice W'. L'output y_i viene quindi calcolato tramite la funzione di Softmax:

$$y_i = P(w_{y_i} \mid w_{t-D}, ..., w_{t-1}, w_{t+1}, ..., w_{t+D}) = \frac{\exp(u_j)}{\sum_{i=1}^{V} \exp(u'_j)} \tag{3.4}$$

3.2.2 Addestramento del modello CBOW

La fase di addestramento dei pesi delle matrici W e W' avviene nel modo seguente: si inizializzano i valori in modo casuale, normalmente con valori presi in modo uniforme nell'intervallo $[-1, 1]$. Dopodiché in modo iterativo vengono forniti al modello esempi di addestramento composti dalle parole contesto relative alla parola centrale di cui si vuole effettuare la predizione. Viene osservato l'errore tra il valore della predizione e il valore corretto, si calcola il gradiente di tale errore rispetto agli elementi di entrambe le matrici e vengono quindi applicati gli aggiornamenti alle matrici stesse.

La funzione obbiettivo da massimizzare è costituita dalla probabilità condizionata della parola di output date le parole contesto in input:

$$E = -\log P(w_O \mid w_{I,t-D}, ..., w_{I,t-1}, w_{I,t+1}, ..., w_{I,t+D}) \tag{3.5}$$

$$= -u_{j^*} - \log \sum_{j'=1}^{V} \exp(u_{j'}) \tag{3.6}$$

$$= -v_{w_O}^T \cdot h - \log \sum_{j'=1}^{V} \exp(v_{w_{j'}}^T \cdot h) \tag{3.7}$$

Dove j^* è l'indice della parola di output corrente. Adesso abbiamo bisogno delle equazioni per l'aggiornamento dei pesi delle matrici W e W'.

Il primo passo è il calcolo della derivata della funzione E rispetto all'input del layer di output u_j:

$$\frac{\partial E}{\partial u_j} = y_j - t_j = e_j \tag{3.8}$$

Dove $t_j = 1$ se $j = j^*$, altrimenti $t_j = 0$. Questo non è altro che l'errore di predizione del nodo j nel layer di output. Il secondo passo è il calcolo della derivata di E rispetto ai pesi w'_{ij} utilizzando la regola della catena vista nel capitolo precedente:

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w'_{ij}} \tag{3.9}$$

$$= (y_j - t_j) \cdot h_i \tag{3.10}$$

L'equazione di aggiornamento per i pesi della matrice W' che collega il layer nascosto con il layer di output sono quindi:

$$w_{ij}^{'(new)} = w_{ij}^{'(old)} - \eta \cdot (y_j - t_j) \cdot h_i \tag{3.11}$$

o,

$$(v'_{w_j})^{(new)} = (v'_{w_j})^{(old)} - \eta \cdot (y_j - t_j) \cdot h \tag{3.12}$$

Dove $\eta > 0$ è il passo di apprendimento, v'_{w_j} è il vettore di output di w_j.

Per quanto riguarda l'aggiornamento della matrice W, dobbiamo trovare un'equazione simile per i pesi w_{ij}. Il primo passo consiste nel calcolo della derivata di E rispetto al nodo nascosto h_i, utilizzando la regola della catena:

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \tag{3.13}$$

$$= \sum_{j=1}^{V} (y_j - t_j) \cdot w'_{ij} = \sum_{j=1}^{V} e_j \cdot w'_{ij} = EH_i \tag{3.14}$$

Dove e_j è l'errore di predizione della j-esima parola nel layer di output, e EH è un vettore di dimensione d composta dalla somma dei vettori di output di tutte le parole del dizionario, pesate dall'errore di predizione. Il passo successivo è composto dal calcolo della derivata di E rispetto a un peso arbitrario in input w_{ki}:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} \tag{3.15}$$

$$= \sum_{j=1}^{V} (y_j - t_j) \cdot w'_{ij} \cdot \frac{1}{2D} \cdot x_k = \frac{1}{2D}(x \cdot EH) \tag{3.16}$$

L'equazione di aggiornamento per i pesi della matrice W è dunque:

$$(v'_{w_{I,c}})^{(new)} = (v'_{w_{I,c}})^{(old)} - \eta \cdot \frac{1}{2D} \cdot EH \tag{3.17}$$

Dove $w_{I,c}$ è la c-esima parola nell'insieme contesto di input e $\eta > 0$ è il passo di apprendimento.

3.3 Modello Skip-Gram

Il modello Skip-Gram si occupa dell'addestramento di Word Embeddings attraverso l'ottimizzazione di una funzione obbiettivo che calcola la probabilità delle parole contesto $\{w_{t-D}, ..., w_{t-1}, w_{t+1}, ..., w_{t+D}\}$ in una certa finestra di dimensione $2D$ data la parola centrale corrente w_t. Lo schema di tale modello è rappresentato in Figura 3.3:

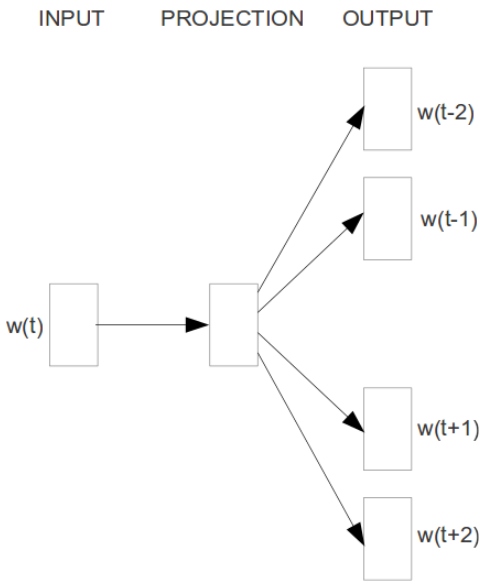

Figura 3.3: Schema del modello Skip-Gram

3.3.1 Forward Propagation in Skip-Gram

L'input del modello è il vettore one-hot x relativo alla parola centrale corrente w_t, mentre in output l'insieme dei vettori $\{y_{t-D}, ..., y_{t-1}, y_{t+1}, ..., y_{t+D}\}$ relativi alle parole contesto $\{w_{t-D}, ..., w_{t-1}, w_{t+1}, ..., w_{t+D}\}$. La matrice W di dimensione $V \times d$ è la matrice dei pesi che collega lo strato di input allo strato nascosto la cui i-esima riga rappresenta i pesi relativi alla i-esima parola del vocabolario (è cioè la matrice contenente i vettori relativi alle parole centrali). Inoltre è presente una matrice W' di dimensione $d \times V$ contenete i pesi relativi alle parole contesto. Nel layer nascosto viene effettuato un prodotto tra il vettore one-hot relativo alla parola centrale attuale e la matrice dei pesi W. Essendo il vettore x one-hot, il prodotto vettore-matrice restituisce come risultato il vettore dei pesi relativo alla parola centrale, cioè la k-esima riga della matrice W:

$$h = x^T W = W_{(k,.)} = v_{w_I} \tag{3.18}$$

Nel layer di input, a differenza del modello visto nella sezione precedente, viene effettuato il calcolo di $2D$ distribuzioni, una per ogni parola contesto nella finestra, applicando nuovamente la funzione Softmax. L'input del j-esimo nodo per la c-esima parola in output è:

$$u_{c,j} = (v'_{w_j})^T h \tag{3.19}$$

L'output del modello è dunque:

$$P(w_{c,j} = w_{O,c} \mid w_I) = y_{c,j} = \frac{\exp(u_{c,j})}{\sum_{j'=1}^{V} \exp(u_{j'})} \tag{3.20}$$

Viene effettuato il calcolo della probabilità che il vettore prodotto dal modello sia effettivamente il vettore relativo alla parola contesto c-esima all'interno della finestra in esame, data la parola centrale in input.

3.3.2 Addestramento del modello Skip-Gram

Nella sezione precedente abbiamo visto come l'input del modello viene propagato attraverso la rete per generare l'output. In questa sezione vediamo come ricavare le equazioni necessarie per gli aggiornamenti delle matrici dei pesi W e W'. La funzione obbiettivo del modello è:

$$E = -\log P(w_{O,t-D}, ..., w_{O,t-1}, w_{O,t+1}, ..., w_{O,t+D} \mid w_I) \tag{3.21}$$

$$= -\log \prod_{-D \leq c \leq D, c \neq 0} \frac{\exp(u_{cj_c^*})}{\sum_{j'=1}^{V} \exp(u'_{j'})} \tag{3.22}$$

$$= -\sum_{-D \leq c \leq D, c \neq 0} (u_{c_c^*}) + 2D \cdot \log \sum_{j'=1}^{V} \exp(u_{j'} \tag{3.23}$$

Viene quindi effettuato il calcolo della derivata di E rispetto agli input della rete $u_{c,j}$:

$$\frac{\partial E}{\partial u_{c,j}} = y_{c,j} - tc, j = e_{c,j} \tag{3.24}$$

che è l'errore di predizione. Indichiamo inoltre con EI il vettore V-dimensionale che raccoglie la somma delle predizioni su tutte le parole contesto:

$$EI_j = \sum_{-D \leq c \leq D, c \neq 0} e_{c,j} \tag{3.25}$$

A questo punto si può effettuare il calcolo della derivata della funzione obbiettivo E rispetto ai valore della matrice W':

$$\frac{\partial E}{\partial w'_{i,j}} = \sum_{-D \leq c \leq D, c \neq 0} \frac{\partial E}{\partial u_{c,j}} \cdot \frac{\partial u_{c,j}}{\partial w'_{i,j}} = EI_j \cdot h_i \tag{3.26}$$

Possiamo a questo punto ricavare l'equazione di aggiornamento per la matrice dei pesi W':

$$(w'_{i,j})^{(new)} = (w'_{i,j})^{(old)} - \eta \cdot EI_j \cdot h_i \tag{3.27}$$

o,

$$(v'_{w_j})^{(new)} = (v'_{w_j})^{(old)} - \eta \cdot EI_j \cdot h \tag{3.28}$$

Dopo aver trovato un'equazione per aggiornare i pesi di W', cerchiamo di trovare un'espressione analoga per l'aggiornamento di W. Iniziamo con il calcolo della derivata rispetto al layer nascosto h:

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \tag{3.29}$$

$$= \sum_{j=1}^{V} \sum_{-D \leq c \leq D, c \neq 0} (y_{c,j} - t_{c,j}) \cdot w'_{i,j} \tag{3.30}$$

Ora è possibile passare al calcolo della derivata rispetto a W:

$$\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} \tag{3.31}$$

$$= \sum_{j=1}^{V} \sum_{-D \leq c \leq D, c \neq 0} (y_{c,j} - t_{c,j}) \cdot w'_{i,j} \cdot x_k \tag{3.32}$$

E quindi l'equazione di aggiornamento per i pesi di W è:

$$(w_{i,j})^{(new)} = (w_{i,j})^{(old)} - \eta \cdot \sum_{j=1}^{V} \sum_{-D \leq c \leq D, c \neq 0} (y_{c,j} - t_{c,j}) \cdot w'_{ij} \cdot x_j \tag{3.33}$$

3.3.3 Ottimizzazioni del modello

Nelle sezioni precedenti è stato presentato il modello Skip-gram e la sua funzione obbiettivo, la quale effettua un calcolo di probabilità sulle parole contesto in una certa finestra attorno ad una parola centrale che è l'input del modello. In modo più formale, data una sequenza di parole di addestramento $\{w_1, w_2, ..., w_T\}$, la funzione obbiettivo di Skip-Gram è massimizzare la probabilità logaritmica media:

$$\frac{1}{T} \sum_{t=1}^{T} \sum_{-D \leq j \leq D, j \neq 0} \log P(w_{t+j} \mid w_t) \tag{3.34}$$

Dove D è la dimensione della finestra contesto. Aumentando la dimensione della finestra è possibile ottenere risultati migliori in termini di accuratezza, al costo di un maggiore tempo di addestramento. La formulazione più semplice per $P(w_{t+j} \mid w_t)$ è quella che utilizza la funzione Softmax:

$$P(w_O \mid w_I) = \frac{\exp(u_{wO}^T v_{wI})}{\sum_{j'=1}^{V} \exp(u_{j'}^T v_{wI)}} \tag{3.35}$$

Dove v_w e u_w sono le rappresentazioni della parola w quando è centrale e quando è contesto, e V è il numero di parole all'interno del dizionario. Come è possibile notare, tale formulazione è inutilizzabile nella pratica, dato che la sommatoria al denominatore viene calcolata su tutti i valori del dizionario, che nelle applicazioni pratiche può essere di grandi dimensioni, nell'ordine di 10^5 - 10^7 termini diversi. Gli autori del modello hanno quindi proposto in [13] alcune possibili soluzioni e accorgimenti per ottenere dei tempi di esecuzione applicabili alle applicazioni pratiche di tale modello.

Hierarchical Softmax

Una possibile soluzione per abbassare la complessità computazionale del calcolo della funzione Softmax su dizionari di grandi dimensioni è quella di effettuare una approssimazione attraverso la tecnica del Hierarchical Softmax, proposta per la prima volta nell'ambio dei modelli del linguaggio basati su reti neurali da Morin e Bengio in [15]. Tale approccio consiste nel valutare la funzione Softmax su un numero di parole logaritmico $\log_2(V)$, invece che sull'intero dizionario V, attraverso l'uso di una rappresentazione ad albero binario dello strato di output con le V parole come foglie e, come nodi interni, le probabilità relative dei nodi figli. Questo tipo di rappresentazione definisce dei cammini casuali che assegnano delle probabilità alle parole.

In modo più formale possiamo dire che ciascuna parola w può essere raggiunta attraverso un percorso che parte dalla radice dell'albero. Sia dunque $n(w, j)$ il j-esimo nodo sul percorso dalla radice alla foglia w, e sia $L(w)$ la lunghezza di tale percorso, in modo tale che $n(w, 1) = root$ e $n(w, L(w)) = w$. Inoltre, per ogni nodo interno n, sia $ch(n)$ un nodo figlio arbitrariamente fissato e sia $[x]$ la funzione:

$$[x] = \begin{cases} 1 & \text{se x è vero} \\ -1 & \text{altrimenti} \end{cases} \tag{3.36}$$

Allora attraverso l'approccio Hierarchical Softmax possiamo definire $P(w_O \mid w_I)$ come:

$$P(w_O \mid w_I) = \prod_{j=1}^{L(w)-1} \sigma([n(w, j+1) = ch(n(w, j))] \cdot u_{n(w,j)}^T v_{w_I}) \tag{3.37}$$

Dove $\sigma(x)$ è la funzione è la funzione sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{3.38}$$

Si può verificare che:

$$\sum_{w=1}^{V} P(w \mid w_I) = 1 \tag{3.39}$$

Quindi il costo per il calcolo di $\log P(w_O \mid w_I)$ e $\nabla \log P(w_O \mid w_I)$ è proporzionale a $L(w_O)$, che in media non è maggiore di $\log V$. Inoltre, al contrario della formulazione standard del Softmax nel modello Skip-Gram, che assegna ad ogni parola due rappresentazioni v_w e u_w, la formulazione del Hierarchical Softmax assegna una rappresentazione v_w ad ogni parola e una rappresentazione u_w per ogni nodo interno dell'albero binario.

La struttura dell'albero utilizzata nel Softmax gerarchico ha una certa importanza sulle performance: gli autori dell'articolo utilizzano nei loro esperimenti un albero binario di Huffman, il quale assegna codici più corti alle parole più frequenti [13].

Negative Sampling

Un approccio alternativo al Hierarchical Softmax è la Noise Contrastive Estimation (NCE), introdotta da Gutmann e Hyvarinen in [7]. NCE ipotizza che un buon modello debba essere in grado di distinguere i dati dal rumore utilizzando la regressione logistica. Nella sua forma più completa NCE si occupa di approssimare la massimizzazione della Log Probability del Softmax, mentre nel modello Skip-Gram si è interessati all'addestramento di Word Embeddings di buona qualità. Dunque è possibile effettuare alcune semplificazioni al NCE. La funzione obbiettivo del Negative Sampling è quindi definita come:

$$\log \sigma(u_{w_O}^T v_{w_I}) + \sum_{i=1}^{k} \mathbb{E}_{w_i \sim P_n(w)} \left[\log \sigma(-u_{w_i}^T v_{w_I}) \right] \tag{3.40}$$

Che viene utilizzato per sostituire ogni occorrenza di $\log P(w_O \mid w_I)$ nella funzione obbiettivo del modello Skip-Gram. Il task è diventato quindi quello di distinguere la parola target w_O dalla distribuzione di rumore $P_n(w)$ utilizzando la regressione logistica, con k campioni negativi per ogni passo di addestramento. La distribuzione $P_n(w)$ è un parametro libero del modello: gli autori dell'articolo, dopo un'analisi delle possibili alternative, pongono come scelta migliore quella della distribuzione unigram $U(w)$ elevata alla potenza $\frac{3}{4}$, cioè:

$$\frac{U(w)^{\frac{3}{4}}}{Z} \tag{3.41}$$

Utilizzando questo accorgimento è possibile abbassare di molto il costo computazionale della fase di aggiornamento dei parametri del modello:

- Nella versione più semplice di Skip-Gram, ad ogni passo di addestramento vengono effettuati i calcoli sulla predizione delle parole contesto a partire dalla parola centrale corrente, calcolati i gradienti e aggiornati i pesi di tutte le parole del dizionario. Per esempio, se si sta utilizzando un dizionario con 1 milione di parole, ad ogni passo verranno aggiornati i vettori relativi a tutte le parole. Questo può essere impraticabile nella realtà a livello di tempi di computazione

- Utilizzando il Negative Sampling proposto dagli autori, ad ogni passo di addestramento vengono aggiornati i pesi relativi alle parole in output, cioè le parole oggetto di previsione data la parola centrale corrente, e k campioni negativi. Il processo di aggiornamento della matrice dei pesi a questo punto viene effettuato con tempi decisamente minori.

Subsampling delle parole frequenti

In collezioni testuali di dimensione molto grande, le parole più frequenti possono facilmente occorrere centinaia di milioni di volte. Un esempio di parole di questo tipo possono essere le preposizioni semplici, gli articoli, le coniugazioni del verbo essere. Questo gruppo di parole ad alta frequenza fornisce chiaramente meno

informazione rispetto a parole più rare. Per esempio, nel modello Skip-Gram la co-occorrenza di parole come "Francia" e "Parigi" fornisce più informazioni rispetto alla co-occorrenza di "Francia" e "la", dato che praticamente tutte le parole co-occorrono con un articolo.

Al fine di ottenere un certo grado di bilanciamento tra le parole frequenti e le parole rare, viene introdotto dagli autori [13] un semplice approccio di subsampling: ogni parola w_i nell'insieme di addestramento viene scartata con probabilità:

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{3.42}$$

Dove $f(w_i)$ è la frequenza della parola w_i e t è un valore di soglia scelto in modo arbitrario, solitamente nell'ordine di 10^{-5}. Gli autori spiegano di aver scelto una formula di questo tipo in quanto effettua un subsampling molto aggressivo contro quelle parole con frequenza maggiore di t. Questa formula è stata scelta in modo completamente euristico, ma si può osservare come nella pratica funzioni molto bene, accelerando il processo di apprendimento e migliorando l'accuratezza dei vettori relativi alle parole rare.

Capitolo 4

TensorFlow

In questo capitolo viene presentato TensorFlow[1], un framework open source per l'apprendimento automatico sviluppato originariamente dal team di Google Brain[2] che mette a disposizione degli sviluppatori tutta una serie di moduli testati ed ottimizzati per la realizzazione di algoritmi e modelli di Machine Learning e Deep Learning. In particolare si tratta di un ottimo strumento per la ricerca e l'applicazione di Machine Learning e Deep Learning.

TensorFlow, come è possibile notare dal nome stesso, si basa sull'utilizzo di flow graph, o grafi di flusso, e tensori, array multidimensionali. TensorFlow inoltre supporta i principali sistemi operativi (Windows[3], Linux[4] e MacOs[5]) e Android[6], e fornisce API native in linguaggio Python[7]/, C/C++, Java[8].

4.1 Modello di programmazione

I calcoli all'interno di un programma TensorFlow sono descritti tramite un grafo aciclico, composto da un insieme di nodi. Il grafo rappresenta il flusso con il quale vengono calcolati i dati, dove ogni nodo ha zero o più input e zero o più output. I valori che scorrono lungo gli archi di tale grafo rappresentano i tensori, array di dimensioni arbitrarie. I calcoli che vengono effettuati localmente in un nodo vengono chiamati operazioni. Tipicamente l'utente costruisce il grafo computazionale utilizzando uno tra i linguaggi supportanti (Python o C++). Una applicazione TensorFlow solitamente è costituita da due fasi distinte:

- Prima fase: il modello viene rappresentato e costruito come un grafo simbolico con placeholder per i dati in input e variabili per lo stato del modello.

[1]https://www.tensorflow.org/
[2]https://ai.google/research/teams/brain/
[3]https://www.microsoft.com/it-it/windows
[4]https://www.linuxfoundation.org/
[5]https://www.apple.com/it/macos/high-sierra/
[6]https://www.android.com/
[7]https://www.python.org
[8]https://java.com/it/

Questo significa che in questa fase il sistema non è a conoscenza del valore dei dati che verranno forniti in input.

- Seconda fase: Il programma viene effettivamente messo in esecuzione in modo ottimizzato utilizzando come dati di input quelli forniti dall'utente sotto forma di dizionari.

Questa divisione in due fasi permette il riutilizzo di uno stesso modello con collezioni di dati di addestramento diverse, oppure la sua condivisione con altri utenti.

Un esempio di grafo computazione viene mostrato in Figura 4.1, con il codice sorgente di esempio relativo [1]:

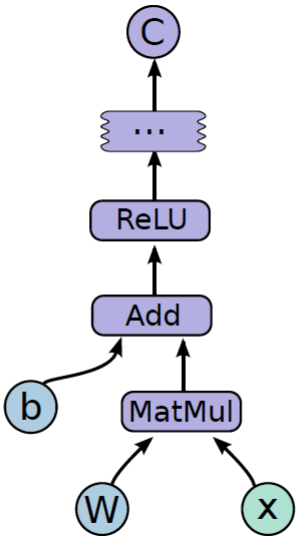

Figura 4.1: Esempio grafo computazionale

Codice 4.1: Esempio creazione ed esecuzione di un grafo

```
import tensorflow as tf

b=tf.Variable(tf.zeros([100]))                    # vettore 100-D a componenti nulle
W=tf.Variable(tf.random_uniform([784,100],-1,1)) # matrice W 784x100 random
x=tf.placeholder(name='x')                        # Placeholder per gli input
relu=tf.nn.relu(tf.matmul(W,x) + b)               # Funzione ReLu(Wx+b)
C = [...]                                         # Funzione di costo


s=tf.Session()
for step in range(0,10):
        input= ...costruzione array input 100-D...      # Creazione vettore di input
        result= s.run(C, feed_dict = {x: input}         # Calcolo del costo
        print step, result
```

Nelle prossime sezioni vengono mostrati i componenti principali di un grafo computazionali in TensorFlow.

Tensori

In TensorFlow i dati sono modellati attraverso tensori, cioè array n-dimensionali con elementi appartenenti ai classici tipi primitivi, come *int32*, *float32* o *string*. I tensori rappresentano, come abbiamo detto, gli input e gli output dei calcoli che vengono solitamente effettuati negli algoritmi di Machine Learning: per esempio, una moltiplicazione tra matrici prende in input due tensori 2-D e produce un tensore 2-D in uscita. Le dimensioni di un tensore possono essere modificate durante la computazione.

Operazioni e Kernel

Un'operazione ha un nome e rappresenta un calcolo astratto, per esempio una moltiplicazione tra matrici, oppure una somma. Riceve in input $m \geq 0$ tensori e produce in output $n \geq 0$ tensori. Un'operazione ha un tipo (come per esempio *Const*, *MatMul*, *Assign*) e può avere zero o più attributi che ne determinano il comportamento in esecuzione. Un uso comune degli attributi delle operazioni è quello di renderle polimorfiche in relazione a diversi tipi di tensori in ingresso.
Le operazioni messe a disposizione da TensorFlow sono:

- operazioni matematiche tra scalari.
- operazioni su array e matrici.
- operazioni di assegnamento.
- blocchi costruttivi per reti neurali.
- operazioni per la gestione dei checkpoint e del controllo di flusso di esecuzione.

Un kernel è una particolare implementazione di un'operazione che può essere eseguita su un particolare tipo di dispositivo: può essere quindi pensato per una specifica CPU o una specifica GPU.

Sessioni

I programmi client interagiscono con il sistema di TensorFlow creando delle sessioni. L'interfaccia Sessione supporta:

- un metodo Extend, che permette di aumentare la dimensione del grafo gestito dalla sessione corrente con nodi e archi addizionali.
- un metodo Run, che riceve in input un insieme di nomi di output che devono essere calcolati, e un insieme di tensori da dare in input al grafo.

Variabili

In una applicazione di TensorFlow può capitare di dover eseguire il grafo svariate volte. La maggior parte dei tensori non è progettata per sopravvivere più di una singola esecuzione. Abbiamo dunque a disposizione le *Variabili*, un tipo particolare di operazione che ritorna un gestore per tensori che sopravvivono attraverso varie esecuzioni del grafo. Nelle applicazioni di Machine Learning, i parametri del modello che devono essere addestrati sono tipicamente salvati all'interno di Tensori gestiti da variabili, e vengono aggiornati durante le esecuzioni del grafo di addestramento.

4.2 Calcolo del Gradiente

Molti algoritmi di ottimizzazione, tra cui i comuni algoritmi di apprendimento automatico come lo stochastic gradient descent che abbiamo visto nel Capitolo 2, effettuano il calcolo del gradiente di una funzione di costo rispetto a un certo insieme di input per poter generare degli aggiornamenti dei parametri per migliorare le predizioni del modello che implementano. Dato che quindi si tratta di una funzione necessaria in tante applicazioni, TensorFlow offre un supporto [1] per il calcolo automatico del gradiente. Se un certo tensore C in un grafo computazionale di TensorFlow dipende, magari attraverso un complesso sottografo di operazioni, da un certo insieme di tensori $\{X_k\}$, è presente una funzione interna che restituisce l'insieme di tensori $\{\frac{\partial C}{\partial x_k}\}$. Il calcolo dei tensori gradiente avviene secondo la seguente procedura.

Nel momento in cui TensorFlow ha bisogno di calcolare il gradiente di un certo tensore C rispetto a un altro tensore I da cui C dipende, per prima cosa cerca il percorso nel grafo computazionale che parte da I e arriva a C. In seguito risale da C a I, e per ogni operazione nel percorso all'indietro aggiunge un nodo al grafo computazionale, componendo i gradienti parziali utilizzando la regola della catena. Questi nuovi nodi calcolano la funzione gradiente per la corrispondente operazione nel percorso di forward. Tornando all'esempio mostrato in Figura 4.1, possiamo osservare tale procedura in Figura 4.2:

4.3 TensorBoard: visualizzazione dei grafi computazionali

Dato che la struttura dei grafi computazionali che si possono definire all'interno di TensorFlow può diventare molto complessa, viene messo a disposizione dell'utente lo strumento TensorBoard, un tool di visualizzazione.

Con l'obbiettivo di aiutare l'utente nel visualizzare l'organizzazione del grafo, che in modelli di Deep Learning può raggiungere diverse decine di migliaia di nodi, TensorBoard è in grado di rappresentare i nodi del grafo tramite blocchi ad alto livello, che ne evidenzino i gruppi con strutture identiche. Il processo di

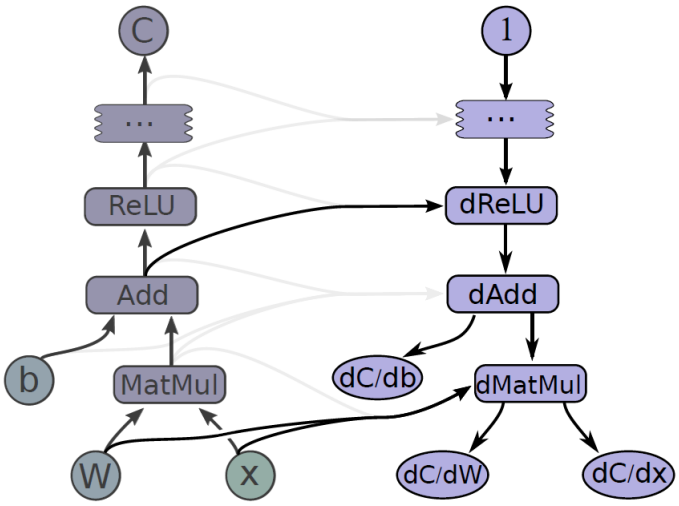

Figura 4.2: Esempio calcolo del gradiente automatico

visualizzazione è interattivo: l'utente può ingrandire e espandere gruppi di nodi per ottenere maggiori dettagli.

Sono inoltre disponibili per l'utente tutta una serie di funzioni per la visualizzazione e l'esame dello stato di vari aspetti del modello rappresentato dal grafo computazionale. TensorFlow supporta quindi diverse operazioni di riepilogo:

- esaminare il valore della funzione di costo mediata su una certa collezione di esempi di addestramento.
- esaminare il tempo necessario all'addestramento del modello.
- visualizzare riepiloghi sotto forma di istogrammi
- visualizzare riepiloghi sotto forma di immagini

Solitamente in fase di progettazione e implementazione del modello, le informazioni necessarie a costruire tali riepiloghi sono specificate all'interno di funzioni apposite che permettono di salvarle in un file di log associato all'addestramento del modello. In questo modo è quindi possibile ricaricare il modello in un secondo momento per poter fare ulteriori iterazioni di addestramento, magari su un nuovo insieme di dati di addestramento, oppure per visualizzare le informazioni sui riepiloghi grazie a TensorBoard.

Capitolo 5

Esperimenti

In questo capitolo vengono presentati gli esperimenti effettuati sul modello Skip-Gram visto nel Capitolo 3, implementato in linguaggio Python utilizzando il framework TensorFlow. In particolare, il modello è stato addestrato utilizzando la collezione di PubMed[1], database di articoli scientifici di argomento biomedicale. Da tale collezione sono state estratte le informazioni riguardanti Titolo e Abstract (se presente), pre-processate e infine utilizzate per l'addestramento del modello. La valutazione è stata effettuata attraverso la valutazione intrinseca, testando semantic similarity e semantic relatedness degli Embeddings ottenuti utilizzando due dataset: il dataset WordSim353, che fornisce degli score di similarity assegnati da operatori umani tra coppie di termini molto frequenti nella lingua inglese, e il dataset UMNSRS[2], che fornisce degli score di similarity e relatedness tra coppie di termini assegnati da medici della Scuola di Medicina dell'Università del Minnesota. Bisogna porre particolare attenzione sul fatto che semantic similarity e semantic relatedness sono due nozioni diverse: la semantic relatedness si riferisce al giudizio umano di quanto una coppia di concetti siano in relazione, ed è la più generale tra le due nozioni, mentre la semantic similarity è un caso speciale di relatedness legata alla somiglianza (in termini di forma o stato) di un concetto. Una misura di semantic similarity prende in input due concetti e restituisce uno score numerico che quantifica in un certo modo quanto i due concetti siano simili. La misura di semantic relatedness prende in considerazione anche informazioni sulle relazioni esistenti tra i concetti e sulla loro co-occorrenza all'interno del corpus in esame [17].

Il codice di tale implementazione, costituito dai moduli per il pre-processing e per la definizione, l'addestramento e la valutazione del modello sono disponibili in un repository GitHub[3].

Gli esperimenti sono stati condotti utilizzando un Notebook Asus[4] modello N53SV equipaggiato con:

[1] https://www.ncbi.nlm.nih.gov/pubmed/
[2] http://rxinformatics.umn.edu/SemanticRelatednessResources.html
[3] https://github.com/andreabeschi/pubmed_word2vec
[4] https://www.asus.com/it/

- Processore Intel[5] Core i7 2630QM.
- 6GB di RAM DDR3 1333 Mhz.
- SSD Samsung[6] 840 EVO da 250GB.
- Scheda Grafica NVIDIA[7] GeForce GT 540M.
- HDD da 1 TB.
- Sistema Operativo Ubuntu[8] 16.04 LTS (Xenial Xerus).

In fase di addestramento del modello tramite TensorFlow non è stata sfruttata la GPU ma solamente la CPU, in quanto la scheda grafica montata sul notebook ha una CUDA Compute Capability[9] di 2.1, mentre il minimo richiesto da TensorFlow per tale parametro è 3.0[10].

5.1 PubMed

La collezione PubMed è prodotta dal National Center for Biotechnology Information[11](NCBI) ed è composta da oltre 28 milioni citazioni a pubblicazioni di ambito biomedicale da:

- MEDLINE[12], database prodotto dalla National Library of Medicine[13](NLM) degli Stati Uniti che copre la letteratura dal 1966 ad oggi nei campi della medicina, dell'infermieristica, della farmacologia, dell'odontoiatria, della veterinaria e dell'assistenza sanitaria in generale. Le citazioni di MEDLINE provengono da oltre 5200 giornali scientifici in circa 40 lingue.
- giornali di scienze biologiche.
- libri online.

Le citazioni possono contenere inoltre i link alla versione completa disponibile su PubMed Central[14] e sui siti web degli editori.

I dati di PubMed sono interamente disponibili al download[15] tramite FTP (File Transfer Protocol) o API. NML produce un insieme di record da MEDLINE/PubMed in formato XML[16] su base annuale, rilasciato nel mese di dicembre.

[5]https://www.intel.it/content/www/it/it/homepage.html
[6]https://www.samsung.com/it/
[7]http://www.nvidia.it/page/home.html
[8]https://www.ubuntu.com/
[9]https://developer.nvidia.com/cuda-gpus
[10]https://www.tensorflow.org/install/install_linux
[11]https://www.ncbi.nlm.nih.gov/
[12]https://www.nlm.nih.gov/bsd/medline.html
[13]https://www.nlm.nih.gov/
[14]https://www.ncbi.nlm.nih.gov/pmc/
[15]https://www.nlm.nih.gov/databases/download/pubmed_medline.html
[16]https://www.w3.org/XML/

Ogni giorno, NML produce dei file di aggiornamento per includere nuove citazioni o versioni revisionate, oppure per eliminare quei record obsoleti e non più necessari.

In particolare, per l'addestramento del modello Skip-Gram, è stata utilizzata la baseline annuale pubblicata il 28 Novembre 2017, composta da 928 file in formato XML (dal file pubmed18n0001.xml al file pubmed18n0928.xml). Le caratteristiche della baseline 2018 sono mostrate in Tabella 5.1:

Formato	**Dimensione** (GB)
Archivio .gz	23.5
File .xml	199

Tabella 5.1: PubMed 2018 baseline

Da tale collezione sono state estratte le informazioni riguardo titoli e abstract delle citazioni generando dei file di testo, che hanno subito una fase di preprocessing per poter essere utilizzati nel modello Skip-Gram per l'addestramento di Word Embeddings.

5.2 Pre-Processing dei dati di addestramento

Come accennato nella sezione precedente, i dati PubMed hanno subito una fase di pre-processing per essere utilizzati per l'addestramento del modello. Le operazioni eseguite in tale fase sono state:

1. apertura dei file XML attraverso il pacchetto Python *xml.etree.ElementTree*. Tale pacchetto permette, tramite la funzione *iterparse* di effettuare il parsing iterativo del file XML, senza doverlo caricare interamente in RAM. Questo requisito è dovuto al fatto che i file XML di PubMed possono raggiungere dimensioni di oltre 300MB e, se caricati interamente in RAM con i riferimenti a tutti i nodi, occupano in breve tempo l'intera memoria disponibile, obbligando il sistema operativo a terminare il programma Python che si occupa dell'estrazione delle informazioni. Scorrendo iteramente il file XML sono quindi state estratte le informazioni relative al campo *ArticleTitle* e al campo *AbstractText*, se presente. I caratteri di tali stringhe sono stati convertiti tutti in lowercase, in modo tale da considerare le versioni delle parole con maiuscole come la stessa parole, i simboli numerici sono stati convertiti nelle rispettive parole, ed è stata rimossa la punteggiatura. In questo modo sono stati ottenuti i testi relativi a titoli e abstract delle citazioni come liste di parole.

2. Le stringhe ottenute al punto precedente sono state dunque salvate in file TXT con la seguente organizzazione: per ogni file XML è stata creata una cartella con lo stesso nome (per esempio, la prima cartella creata è 'pubmed18n0001'), al cui interno è stato salvato un file TXT per ogni articolo.

Per nominare questi file TXT è stato utilizzato l'identificativo univoco della citazione relativa, il PMID.

3. In questa fase sono stati creati i dizionari relativi alla collezione testuale ottenuta nelle fasi precedenti. Sono inoltre state eliminate quelle parole che compaiono nell'intera collezione un numero di volte inferiore al parametro *min_freq*. Si tratta cioè di quelle parole così rare da non rappresentare contenuto informativo, ma anzi da considerare rumore all'interno del modello. Alla fine di questa fase sono stati ottenuti i seguenti dizionari:

   - un dizionario *words_to_int*, che mappa le parole in interi. Ad ogni parola del dizionario è quindi assegnato un identificativo numerico univoco, che permetterà in seguito, in fase di addestramento del modello, di ottenere il vettore relativo alla parola utilizzando una funzione di look-up sulla matrice dei vettori utilizzando proprio questo identificativo.
   - un dizionario *int_to_words*, che effettua l'operazione inversa alla precedente, ovvero permette di recuperare la parola corrispondente ad un certo identificativo numerico.
   - un dizionario *words_count*, che contiene l'informazione sul numero di occorrenze di ciascuna parola all'interno dell'intera collezione. Questo dizionario viene utilizzato nella fase di sub-sampling delle parole frequenti in fasi di addestramento del modello.

Le informazioni riguardanti i dati di addestramento sono mostrate nella Tabella 5.2.

Tempo Pre-Processing (ore)	37
Articoli totali	27.837.540
Dimensione (GB)	95
Parole totali	3.908.723.620
Parole distinte	3.142.434
Parole distinte (freq $\geq$ min_freq)	611.186

Tabella 5.2: Statistiche corpus di addestramento

Analizzando il tempo necessario ad effettuare il pre-processing della collezione, è stato notato che in gran parte dipendeva dall'organizzazione dei file testuali all'interno delle cartelle generate: di fatto, l'intero processo viene rallentato dal disco fisso meccanico, che impiega molto tempo a creare, aprire, scrivere e chiudere i piccoli file testuali, che a volte sono composti unicamente dal titolo della pubblicazione scientifica a cui si riferiscono. Dunque è stata sviluppata una seconda versione, più "compatta", nella quale viene creato un unico file di testo per ognuno dei 928 file XML di PubMed, con all'interno i record relativi, uno per ogni riga. In questo modo la dimensione occupata su disco passa da 95GB

a 23,5GB, e il tempo di elaborazione scende a 4 ore e 45 minuti. Lo stesso tipo di comportamento sarà presente anche in fase di addestramento, per la quale verrà sviluppata anche la versione "compatta". Un confronto fra i risultati tra la versione originale e la versione compatta sono mostrati nella Tabella 5.3.

	Versione originale	**Versione compatta**
Tempo impiegato	37 ore	4 ore e 45 minuti
Spazio occupato	95 GB	23,5 GB

Tabella 5.3: Pre-processing originale vs. Pre-processing compatto

A questo punto i dati sono pronti per essere utilizzati per l'addestramento del modello, la cui implementazione è oggetto della prossima sezione.

5.3 Implementazione del Modello

In questa sezione viene descritto come è stato implementato il modello: vedremo come viene costruito il grafo computazionale relativo al modello Skip-Gram, i suoi componenti principali, e come il modello viene addestrato per ottenere in output i Word Embeddings relativi alle parole contenute nella collezione di PubMed.

5.3.1 Costruzione del Grafo Computazionale

Come già accennato in precedenza, per l'implementazione del modello Skip-Gram è stato utilizzato il linguaggio Python e il framework TensorFlow. Il codice che si occupa della costruzione del grafo computazionale relativo al modello è mostrato nel seguente frammento:

Codice 5.1: Grafo computazionale modello Skip-Gram

```
import tensorflow as tf

graph = tf.Graph()
with graph.as_default():

    #Input
    with tf.name_scope('inputs'):
        train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
        train_labels = tf.placeholder(tf.int32, shape=[batch_size,1])

    #Lookup degli embeddings:
    #inizialmente vengono inizializzati con componenti uniformi tra -1.0 e +1.0
    with tf.name_scope('embeddings'):
        embeddings = tf.Variable(tf.random_uniform(
                                        [vocabulary_size, embedding_size],
                                        -1.0, 1.0))
        embed = tf.nn.embedding_lookup(embeddings, train_inputs)

    #Variabili per la funzione obbiettivo NCE loss:
        #nce_weights matrice dei pesi di dimensioni
        #vocabulary_size * embedding_size
        #nce_biases vettore di zeri di dimensione vocabulary_size
    with tf.name_scope('weights'):
        nce_weights = tf.Variable(tf.truncated_normal(
```

```
                                    [vocabulary_size, embedding_size],
                                    stddev=1.0 / math.sqrt(embedding_size)))
    with tf.name_scope('biases'):
        nce_biases = tf.Variable(tf.zeros([vocabulary_size]))


    #Funzione obbiettivo NCE loss
    with tf.name_scope('loss'):
        loss = tf.reduce_mean(tf.nn.nce_loss(
                                    weights = nce_weights,
                                    biases = nce_biases,
                                    labels = train_labels,
                                    inputs = embed,
                                    num_sampled = num_sampled,
                                    num_classes = vocabulary_size))

    tf.summary.scalar('loss', loss)

    #Funzione di ottimizzazione (SGD con passo di apprendimento di $\alpha$)
    with tf.name_scope('optimizer'):
        optimizer = tf.train.GradientDescentOptimizer($\alpha$).minimize(loss)

    #Op che inizializza le variabili
    init = tf.global_variables_initializer()

    #Saver
    saver = tf.train.Saver()
```

Alla luce di quello che è stato visto nel capitolo precedente, possiamo riconoscere gli elementi principali di un programma TensorFlow:

- Vengono definiti i placeholder per i dati in ingresso che verranno forniti solo in fase di addestramento. Si tratta di un vettore *train_inputs* di dimensione *batch_size* che conterrà le parole centrali e un vettore colonna *train_labels* della stessa dimensione che conterrà le relative parole contesto.

- Viene definita la matrice che conterrà i Word Embeddings da addestrare. Tale matrice ha chiaramente come dimensioni la dimensione del dizionario e la dimensione degli Embeddings. Questi elementi vengono inizializzati a valori casuali presi in modo uniforme tra -1 e 1.

- Viene utilizzata la funzione di look-up *tf.nn.embedding_lookup* che permette di ottenere dall'intera matrice di pesi solo quelli che vengono utilizzati al passo corrente per l'addestramento.

- Vengono definite la matrice dei pesi W di dimensione uguale a quella degli Embeddings, inizializzata con valori casuali e il vettore dei biases b di dimensione *embedding_size*, inizializzata con valori nulli.

- Viene definita la funzione di loss, che implementa l'approccio Noise Contrastive Estimation per il calcolo approssimato della funzione Softmax che abbiamo visto nel capitolo relativo al modello Skip-Gram. Tra i parametri accettati è presente *num_sampled*, ovvero il numero di negative samples da utilizzare.

- Viene definita la funzione di ottimizzazione, in particolare viene utilizzato lo Stochastic Gradient Descent, con passo di addestramento α.

- Vengono infine definite le operazioni che si occupano dell'inizializzazione globale delle variabili e del salvataggio dei riepiloghi nel file di log generato durante la fase di addestramento.

Utilizzando lo strumento TensorBoard è possibile visualizzare il grafo computazionale creato, mostrato in Figura 5.1. All'interno delle frecce, sono riportate le dimensioni delle matrici e dei tensori ottenute durante uno degli esperimenti, ma non sono necessari per comprendere la struttura del grafo.

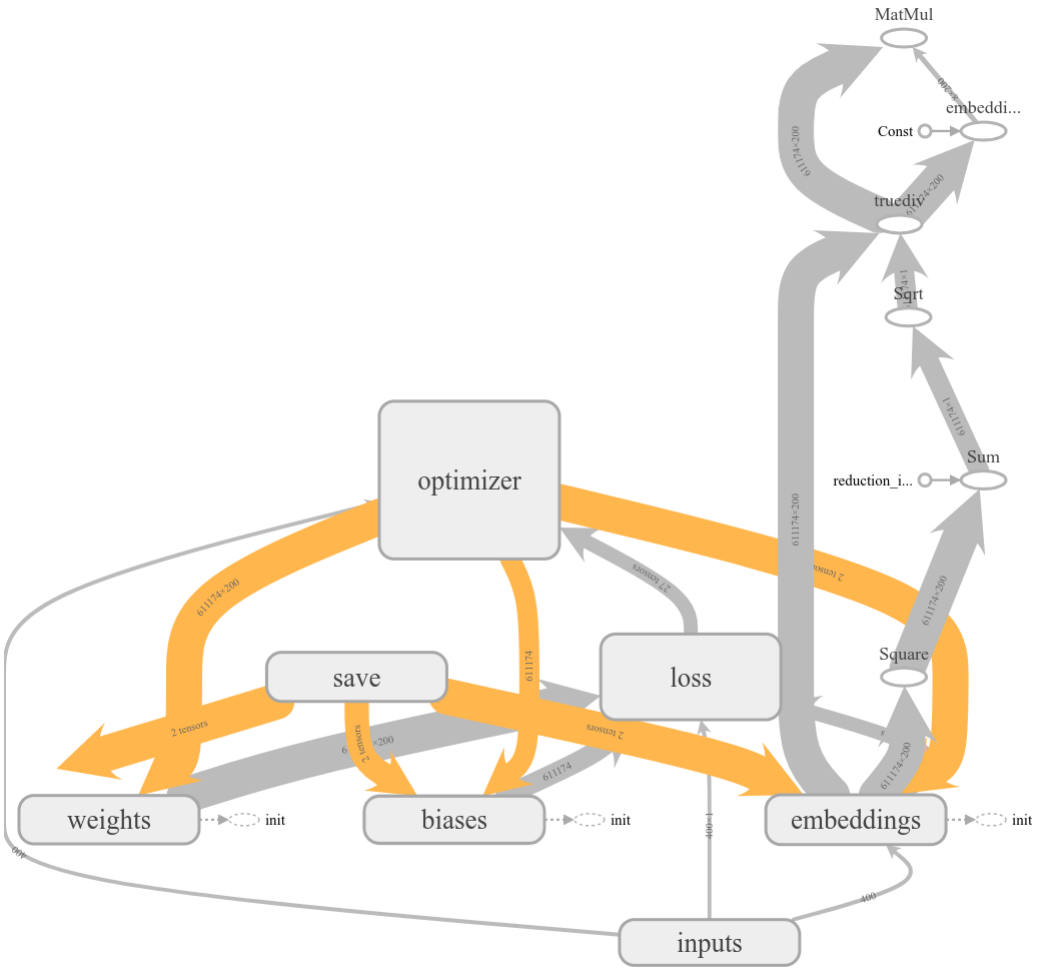

Figura 5.1: Grafo computazionale del modello Skip-Gram

Nella sezione seguente viene presentata la fase di addestramento del modello.

5.3.2 Addestramento del modello

L'addestramento del modello Skip-gram avviene nel modo seguente: viene fatta scorrere l'intera collezione di addestramento che ha subito il processo di pre-processing: in modo iterativo viene aperto ogni file TXT all'interno delle 928

cartelle ottenute in fase di pre-processing in cui avviene lo scorrimento per finestre. Ogni parola all'interno del file testuale viene considerata centrale in una finestra di dimensione *window_size*, e vengono quindi create le relative coppie *(parola centrale, parola contesto)* che vengono fornite al modello in un numero pari al *batch_size* per step di addestramento. Il modello riceve queste coppie in input, effettua il look-up sulla matrice contenente gli Embeddings relativi alle parole contenute in tali coppie, esegue la forward-propagation per ottenere il valore della funzione di costo e ne calcola i gradienti rispetto agli input per poter quindi aggiornare i parametri secondo l'algoritmo Stochastic Gradient Descent. Quando l'intera collezione di addestramento è stata utilizzata, il programma termina, e all'interno della matrice degli Embeddings sono presenti i valori dei Word Embeddings addestrati.

Viene utilizzato l'approccio Negative Sampling presentato in precedenza, con un numero di campioni negativi pari a *num_sampled*, e viene inoltre effettuato il sub-sampling delle parole frequenti utilizzando come valore di soglia *sub_sampling_threshold*.

Per la fase di addestramento sono stati utilizzati impostati i parametri del modello in accordo con quanto proposto in [5], [20] e [14]. Tali parametri sono mostrati nella Tabella 5.4:

Parametro	**Valore**
embedding_size	200
window_size	5
learning_rate	0.5
batch_size	500
num_sampled	20
sub_sampling_threshold	10^{-5}
min_freq	10

Tabella 5.4: Parametri per l'addestramento del modello

Come per la fase di pre-processing, è stata sviluppata una versione compatta del programma di addestramento, la cui unica differenza sta nella generazione delle coppie *(parola centrale, parola contesto)* che avviene da 928 file TXT che contengono tutte le informazioni di un file XML all'interno di un'unica cartella, rispetto alla versione base dove i file testuali sono salvati uno per articolo presente nella collezione PubMed all'interno di cartelle che portano il nome del relativo file XML.

Per quanto riguarda i tempi di elaborazione per l'addestramento del modello sull'intero dataset di addestramento, abbiamo un comportamento simile a quello osservato in fase di pre-processing: la versione base del modello che utilizza l'organizzazione in cartelle dei file testuali risente molto dei tempi di apertura di ogni singolo file di testo. Questo si traduce in un tempo di addestramento di 30 ore. La versione compact, invece, che utilizza un unico file testuale per ognuno dei 928 file XML di PubMed, ha registrato un tempo di addestramento di 18 ore e 32 minuti. Un riepilogo è mostrato nella Tabella 5.5:

Versione	Tempo di Addestramento
Base	30 ore
Compact	18 ore e 32 minuti

Tabella 5.5: Tempi di addestramento del modello

Infine bisogna sottolineare la riutilizzabilità dei programmi sviluppati con altre collezioni di dati di addestramento, anche di argomento diverso da quello medico preso in esame in questa tesi. Infatti è possibile addestrare il modello tramite i programmi sviluppati e disponibili nel repository GitHub[17] utilizzando altre collezioni, con l'unico vincolo che il corpus di documenti sia strutturato in file testuali all'interno di cartelle (per la versione base dell'implementazione) oppure in grandi file testuali dove i contenuti sono organizzati uno per riga.

5.4 Risultati e valutazione

La valutazione dei Word Embeddings ottenuti nella fase di addestramento è stata eseguita seguendo la valutazione intrinseca presentata in [19] e [5]. In particolare, sono stati utilizzati il dataset WordSim353 proposto in [2] e il dataset UMNSRS[18] proposto in [16]. Il primo dataset è composto da 353 coppie di termini molto frequenti nella lingua inglese a cui è stato assegnato un valore di semantic similarity intervistando degli operatori umani. Il secondo dataset si compone di due insiemi di coppie di termini di ambito medico per darne una misura di semantic similarity e di semantic relatedness:

- UMNSRS-Sim, composto da 449 coppie di termini per le quali è assegnato uno score di similarity.
- UMNSRS-Rel, composto da 458 coppie di termini per le quali è assegnato uno score di relatedness.

I punteggi assegnati alle coppie di termini sono stati assegnati da medici appartenenti alla Scuola di Medicina dell'Università del Minnesota. Riportiamo ancora una volta l'attenzione sul fatto che semantic similarity e semantic relatedness sono due nozioni diverse, nonostante siano in un qualche modo collegate tra loro. In particolare, la semantic similarity è un caso particolare di relatedness dove l'attenzione è posta sulla somiglianza in termini di forma di due concetti, mentre la semantic relatedness si riferisce ad un concetto più ampio di relazione tra concetti che non tiene conto della loro forma. Questa osservazione è importante per poter analizzare i risultati ottenuti, in quanto il fatto che la similarity sia un caso particolare di relatedness, come vedremo, si traduce in minori coefficienti di correlazione tra similarità del coseno degli Embeddings e score assegnati da operatori umani.

[17] https://github.com/andreabeschi/pubmed_word2vec
[18] http://rxinformatics.umn.edu/SemanticRelatednessResources.html

Per il calcolo della similarity tra gli Embeddings addestrati con il modello Skip-Gram delle coppie di termini all'interno dei dataset è stata utilizzata la misura della similarità del coseno. Date quindi due parole w_1 e w_2 appartenenti al dizionario, la similarità del coseno è definita come:

$$\text{cosine\_sim}(w_1, w_2) = \frac{u_1 \cdot u_2}{\|u_1\| \, \|u_2\|} \tag{5.1}$$

Dove u_1 e u_2 sono i Word Embeddings relativi a w_1 e w_2.

Sono stati quindi calcolati i valori di similarità del coseno per le coppie contenute in WordSim353, UMNSRS-Sim e UMNSRS-Rel, i cui valori sono stati confrontati con gli score assegnati dagli operatori umani utilizzando il coefficiente ρ di correlazione di Spearman:

$$\rho = 1 - \frac{6 \sum_i D_i^2}{N(N^2 - 1)} \tag{5.2}$$

Dove $D_i = r_i - s_i$ è la differenza dei ranghi r_i della prima variabile e s_i della seconda variabile all'oservazione i-esima, e N è il numero complessivo di osservazioni.

I risultati di tale processo di valutazione intrinseca sono mostrati nella Tabella 5.6.

Dataset	ρ
WordSim353	0.10
UMNSRS-Sim	0.43
UMNSRS-Rel	0.47

Tabella 5.6: Valutazione intrinseca

Per quanto riguarda la valutazione sul dataset UMNSRS i risultati sono in linea con quanto ottenuto in [5] e [20]: il modello è quindi in grado di codificare informazioni riguardanti la semantic similarity e la semantic relatedness all'interno dei vettori che vengono addestrati. In particolare, come è stato accennato in precedenza, si può osservare che il coefficiente di correlazione relativo a UMNSRS-Sim sia minore rispetto al coefficiente relativo a UMNSRS-Rel, in quanto la similarity è un caso speciale di relatedness. I risultati relativi al dataset WordSim353 sono invece meno incoraggianti, in quanto il valore del coefficienti di correlazione tra gli score assegnati dagli operatori umani e i valori di similarità del coseno ottenuti dagli embeddings addestrati dal modello risulta essere molto basso. Questo è però sicuramente dovuto al tipo di collezione con cui è stata effettuato l'addestramento del modello: avendo infatti utilizzato titoli e abstract di articoli scientifici di ambito bio-medicale è un risultato atteso che il modello abbia addestrato meglio i Word Embeddings relativi a termini specifici del campo bio-medicale.

Capitolo 6

Conclusioni

In questo capitolo finale viene riassunto il lavoro di tesi fatto e raccolte le conclusioni sul lavoro svolto e sui risultati ottenuti. Argomento principale di questa tesi è lo studio dei Word Embeddings per documenti di ambito medico. In particolare è stato studiato il modello Skip-Gram [12], implementato ed addestrato tramite la collezione di citazioni da articoli scientifici di PubMed. L'obbiettivo principale è stato quello di sviluppare uno strumento che potesse essere utilizzato in modo efficace per l'addestramento di tali Embeddings sfruttando una piattaforma con poche risorse di calcolo come un normale notebook (senza peraltro l'utilizzo di una GPU). Il codice di tale implementazione è reso disponibile in un repository GitHub e può essere liberamente utilizzato.

Nel primo capitolo è stato introdotto l'ambito di ricerca del Natural Language Processing, o Elaborazione del Linguaggio Naturale: ne sono state mostrate le caratteristiche principali, le motivazioni che ne stanno alla base e le problematiche maggiori che si riscontrano affrontandolo, tra cui spicca l'intrinseca ambiguità tipica dei linguaggi naturali, che qualifica il problema della perfetta comprensione del linguaggio naturale come problema AI-completo [22]. Sono stati mostrati i livelli a cui si applica l'Elaborazione del Linguaggio Naturale, gli approcci utilizzati in letteratura, e le principali applicazioni pratiche in cui trova utilizzo, tra cui troviamo sistemi di reperimento dell'informazione, sistemi di Question-Answering, Sentiment Analysis, Machine Translation e assistenti vocali.
All'interno dell'ampio campo di ricerca del Natural Language Processing ci siamo poi concentrati sui Word Embeddings, argomento principale di questa tesi: ne è stata data una definizione formale, è stato presentato il problema principale per il quale sono stati sviluppati, cioè quello della rappresentazione delle parole all'interno dei calcolatori, e anche in questo caso, i campi di applicazione in cui trovano utilizzo.

Il secondo capitolo è dedicato al Deep Learning e alle reti neurali, strumenti fondamentali per il lavoro che è stato portato avanti con questa tesi. Partendo da un'introduzione al Machine Learning, di cui il Deep Learning è un'estensione, ne è stata data una definizione formale in accordo con [6], ne sono stati mostrati gli elementi fondamentali ed è stato sviluppato l'esempio della regressione lineare. Un argomento molto importante e fondamentale per il lavoro mostrato nei

capitolo successivi è quello dell'ottimizzazione basata sul gradiente, in particolare attraverso il metodo della Stochastic Gradient Descent. Sono state in seguito introdotte le reti neurali, altro tassello fondamentale per la costruzione del modello oggetto dello studio di questa tesi. Dopodiché è stato introdotto il Deep Learning, come abbiamo detto campo del Machine Learning, che permette l'addestramento di reti neurali anche molto complesse tramite l'organizzazione in strati più semplici. Da questo derivano reti neurali anche molto 'profonde'. Fondamentale in questo ambito è l'algoritmo di Back-Propagation [6], che permette il calcolo dei valori del gradiente della funzione obbiettivo del modello rispetto agli input e che quindi fornisce le informazioni al modello per la generazione degli aggiornamenti dei parametri utilizzati per l'addestramento.

Nel terzo capitolo è stato quindi introdotto il modello Word2Vec proposto da Mikolov et al. in [12] per l'addestramento di Word Embeddings di qualità da corpus di documenti anche molto ampi. Si tratta di un modello molto potente ma al tempo stesso semplice e leggero, apprezzato in letteratura e oggetto di svariati esperimenti, come in [5], [20] e [14]. L'idea alla base di tale modello è quella secondo la quale parole che compaiono all'interno del testo in contesti simili condividono un qualche tipo di significato (Harris 1954). Sono stati quindi proposti due approcci: il modello Continuous-Bag-of-Words (CBOW), che effettua predizioni sulla parola centrale data la media delle parole contesto, e il modello Skip-Gram, che in modo speculare effettua predizioni sulle parole contesto data la parola centrale. Di questi modelli sono state presentate le strutture, i parametri, le funzioni obbiettivo e gli aggiornamenti ai parametri che ne permettono l'addestramento. Per quanto riguarda il modello Skip-Gram, utilizzato negli esperimenti condotti, sono stati inoltre mostrati alcuni accorgimenti proposti dagli autori in [13] che permettono di ottenere costi computazionali molto minori in fase di addestramento: si tratta del Negative Sampling, che rende possibile il calcolo approssimato della funzione Softmax, e il subsampling delle parole frequenti, che permette di ridurre le co-occorrenze di termini che restituiscono poca informazione semantica.

Nel quarto capitolo è stato presentato uno strumento fondamentale per l'implementazione del modello Skip-Gram: il framework di Machine Learning e Deep Learning TensorFlow [1]. Di tale strumento sono stati mostrati gli aspetti e le funzioni principali utilizzate: il modello di programmazione, i componenti fondamentali, il calcolo automatizzato del gradiente e lo strumento TensorBoard, per la visualizzazione dei grafi computazionali e delle informazioni sul processo di addestramento del modello e sui risultati ottenuti.

Nel quinto capitolo sono stati presentati gli esperimenti condotti sul modello Skip-Gram utilizzando come corpus di addestramento la collezione di argomento bio-medicale PubMed. Il modello Skip-Gram è stato implementato utilizzando il linguaggio di programmazione Python e il framework TensorFlow. Gli esperimenti sono stati condotti su un notebook con processore Intel Core i7 2630QM, 6GB di RAM e O.S Ubuntu 16.04. I dati di PubMed, liberamente scaricabili da Internet, sono stati estratti e pre-processati per eliminare punteggiatura, parole troppe rare e in generale per formattare il testo per renderlo utilizzabile in fase di addestramento del modello. In particolare sono stati utilizzati i testi

relativi a titoli e abstract degli articoli scientifici di ambito bio-medicale presenti nella collezione di PubMed. Il modello è stato quindi addestrato con tale collezione, e gli Embeddings risultanti sono stati valutati in modo intrinseco in termini di semantic similarity e semantic relatedness attraverso due dataset: il dataset UMNSRS, costituito da coppie di termini medici per i quali operatori umani esperti hanno assegnato degli score, e il dataset WordSim353, costuito da coppie di termini molto frequenti nella lingua inglese per i quali sono assegnati degli score. I risultati ottenuti in termini di correlazione tra gli score assegnati alle coppie di termini in UMNSRS e i valori di similarità del coseno tra gli Embeddings delle relative parole ottenuti tramite l'addestramento sono in linea con analoghe pubblicazioni [5], [20] e mostrano come il modello Skip-Gram, applicato alla collezione di ambito medico, sia in grado di addestrare gli Embeddings in modo tale che codifichino al loro interno informazioni semantiche sui termini specifici delle pubblicazioni scientifiche. Word Embeddings che stabiliscono relazioni di similarity e relatedness correlate a quelle assegnate da operatori umani esperti di tale argomento.

In conclusione, è possibile affermare che il modello, avendo a disposizione una collezione di training molto ampia, si comporta bene nel compito dell'addestramento dei Word Embeddings per collezioni di ambito medico. Nonostante la sua potenza, si tratta di un modello molto semplice che è possibile implementare utilizzando strumenti open-source molto maturi come TensorFlow e che soprattutto è possibile addestrare utilizzando la potenza computazionale di un notebook. Una volta addestrati gli Embeddings, è possibile utilizzarli per tutta una serie di task relativi all'Elaborazione del Linguaggio Naturale, come per esempio il miglioramento delle prestazioni di sistemi di reperimento dell'informazione specifici per l'ambito medico e per la categorizzazione automatica di testi. Un possibile sviluppo futuro può essere quello di effettuare l'addestramento sfruttando le ottimizzazioni di TensorFlow per il calcolo su GPU per abbassare ulteriormente i tempi di addestramento del modello (il notebook utilizzato per gli esperimenti montava una scheda grafica troppo obsoleta per tale utilizzo), e il miglioramento del motore di ricerca interno di PubMed, che per il momento permette unicamente ricerche attraverso query booleane, ma che utilizzando le informazioni sui Word Embeddings allenati tramite la sua collezione, potrebbe rendere disponibile agli utenti query più sofisticate ed efficienti.

Bibliografia

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL '09, pages 19–27, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[3] Leonard E. Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Ann. Math. Statist.*, 37(6):1554–1563, 12 1966.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[5] Billy Chiu, Gamal K. O. Crichton, Anna Korhonen, and Sampo Pyysalo. How to train good word embeddings for biomedical nlp. In *BioNLP@ACL*, 2016.

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

[7] Michael U. Gutmann and Aapo Hyvärinen. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *J. Mach. Learn. Res.*, 13(1):307–361, February 2012.

[8] Zellig S. Harris. Distributional structure. *WORD*, 10(2-3):146–162, 1954.

[9] Yang Li and Tao Yang. *Word Embedding for Understanding Natural Language: A Survey*, pages 83–104. Springer International Publishing, Cham, 2018.

[10] Elizabeth D. Liddy. Natural language processing. In *Encyclopedia of Library and Information Science*. Syracuse University, 2001.

[11] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999.

[12] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[14] José Antonio Miñarro-Giménez, Oscar Marín-Alonso, and Matthias Samwald. Applying deep learning techniques on medical corpora from the world wide web: a prototypical system and evaluation. *CoRR*, abs/1502.03682, 2015.

[15] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In Robert G. Cowell and Zoubin Ghahramani, editors, *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*, pages 246–252. Society for Artificial Intelligence and Statistics, 2005.

[16] Serguei V.S. Pakhomov, Greg Finley, Reed McEwan, Yan Wang, and Genevieve B. Melton. Corpus domain effects on distributional semantic modeling of medical terms. *Bioinformatics*, 32(23):3635–3644, 2016.

[17] Ted Pedersen, Serguei V.S. Pakhomov, Siddharth Patwardhan, and Christopher G. Chute. Measures of semantic similarity and relatedness in the biomedical domain. *Journal of Biomedical Informatics*, 40(3):288 – 299, 2007.

[18] Douglas L. T. Rohde and David C. Plaut. Connectionist models of language processing. *Cognitive Studies*, 10(1):10–28, 2003.

[19] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 298–307. Association for Computational Linguistics, 2015.

[20] Yanshan Wang, Sijia Liu, Naveed Afzal, Majid Rastegar-Mojarad, Liwei Wang, Feichen Shen, Paul Kingsbury, and Hongfang Liu. A comparison of word embeddings for the biomedical natural language processing. *CoRR*, abs/1802.00400, 2018.

[21] A. Donald Booth William N. Locke. *Machine Translation of Languages: Fourteen Essays*. Technology Press, 1955.

[22] Roman Yampolskiy. Ai-complete, ai-hard, or ai-easy - classification of problems in ai. In *CEUR Workshop Proceedings*, volume 841, pages 94–101, 01 2012.