Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Analisi e implementazione di un parser neurale per la lingua italiana

Relatore:
Chiar.mo Prof.
FABIO TAMBURINI

Presentata da:
FILIPPO BARTOLINI

Sessione III
Anno Accademico 2019/2020

Ai miei nonni,
Ai miei genitori,
A chiunque mi sia stato vicino,
aiutandomi a raggiungere questo traguardo.

Introduzione

L'analisi sintattica automatica di testi scritti in linguaggio naturale, comunemente nota come parsing, è diventata da diversi anni uno dei principali interessi nel campo dell'elaborazione del linguaggio naturale. Ciò è in parte dovuto al fatto che questo tipo di analisi è presente in molti sistemi diversi, in quanto fornisce un'importante fase preliminare per svariate applicazioni linguistiche come la traduzione automatica, la sentiment analysis, l'information retrieval e via dicendo. Infatti, riuscire a determinare correttamente la struttura sintattica di una frase permette di migliorare, in generale, le prestazioni dei suddetti task.

A causa delle ambiguità e delle diverse astrazioni delle lingue storico-naturali, questa tipologia di analisi risulta essere abbastanza complessa, richiedendo un certo impegno sia per la modellazione di parser performanti sia per la reperibilità delle risorse. Infatti, buona parte degli studi disponibili in letteratura trattano il problema dell'analisi sintattica per la lingua inglese o cinese, data la moltitudine di risorse disponibili rispetto alle altre lingue.

Successivamente allo sviluppo del word embedding contestuale BERT [1], basato su architetture a Transformers e inizialmente disponibile solo per la lingua inglese, la maggior parte delle applicazioni legate all'elaborazione del linguaggio naturale ha ottenuto un incremento prestazionale, compresa l'analisi sintattica effettuata tramite modelli neurali. I miglioramenti dovuti a BERT hanno fatto sì che all'interno delle varie comunità scientifiche si sviluppassero modelli da esso derivati, ma specificatamente addestrati su corpora di altre lingue.

In questo lavoro di tesi, viene affrontato il task di analisi sintattica per la lingua italiana, utilizzando un corpus di dominio generico presente nelle Universal Dependencies [47]. Inizialmente, si sperimenta e si analizzano i risultati ottenibili sfruttando unicamente le rappresentazioni linguistiche di UmBERTo, il più recente modello linguistico pre-addestrato su un corpus italiano, la cui architettura deriva da RoBERTa.

Successivamente, si testano e si esaminano i risultati ottenibili utilizzando il recente parser neurale PaT [2], congiuntamente alle rappresentazioni linguistiche di UmBERTo. Infine, si tenta di migliorare le prestazioni precedentemente ottenute modificando l'architettura del parser PaT con UmBERTo, prendendo ispirazione dal parser neurale di Dozat e Manning [3] che rappresenta lo stato dell'arte.

Struttura della tesi

Nel Capitolo 1 è presente un'introduzione alle reti e ai modelli neurali. In particolare, vengono messi in evidenza i diversi tipi di apprendimento con le loro peculiarità e le principali reti neurali artificiali deep, sulle quali si basano le architetture dei parser utilizzati negli esperimenti oggetto di questa tesi.

Nel Capitolo 2 viene introdotta in dettaglio l'architettura dei Transformers [4], prestando particolare attenzione a BERT [1] e ad alcuni modelli da esso derivati, con specifico riguardo verso UmBERTo.

Nel Capitolo 3 è definito il problema del dependency parsing, affrontato sia nell'approccio transition-based che graph-based. Inoltre, sono introdotte le Universal Dependencies e le metriche con cui è possibile valutare le prestazioni dei parser. Infine, sono riportati alcuni esempi di parser neurali sia transition-based che graph-based.

Nel Capitolo 4 sono descritti il corpus e tutti gli esperimenti oggetto di questa tesi. In particolare, i primi due esperimenti definiscono una baseline per il task di dependency parsing per la lingua italiana, tramite l'utilizzo delle sole rappresentazioni linguistiche di UmBERTo. Successivamente, vengono descritte l'architettura del parser PaT [2] e tutte le modifiche ad essa

applicate per lo sviluppo di un parser neurale a dipendenze per la lingua italiana, tramite l'utilizzo di UmBERTo. Infine, vengono analizzati i risultati ottenuti.

Nell'Appendice A vengono riportate alcune statistiche sul corpus utilizzato in tutti gli esperimenti.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Reti e modelli neurali

Nel corso degli ultimi anni le materie di studio legate all'intelligenza artificiale, con particolare riguardo per la branca del *machine learning*, stanno assumendo un ruolo sempre più rilevante in ambito tecnologico, ma non solo. Infatti, diversi sono i settori economici che hanno riconosciuto nelle tecniche di machine learning dei validi alleati, grazie soprattutto alla loro capacità di riuscire a prendere decisioni basate unicamente sui dati di cui, al giorno d'oggi, le aziende dispongono in grande quantità. I casi d'uso del machine learning sono veramente numerosi. Questi includono la sanità, le scienze, i viaggi, i servizi finanziari, la previsione di fenomeni ecc. Anche gli oggetti che oggi vengono considerati di uso comune, come ad esempio gli smartphone, sono dotati di apparati di intelligenza artificiale per portare a compimento diversi compiti.

Lo scopo di questo primo capitolo è di fornire una breve panoramica su quello che è il significato dell'apprendimento contestualizzato nell'ambito del machine learning. Inoltre, verranno descritti alcuni modelli di reti, prestando particolare attenzione a quelli utilizzati maggiormente dai moderni parser neurali.

1.1 Machine Learning

Il *machine learning* (o *apprendimento automatico*), è una branca in fortissima espansione dell'intelligenza artificiale che permette alle macchine di riuscire a svolgere determinati compiti, senza dover programmare esplicitamente tutti i passi necessari per raggiungere l'obiettivo. Il termine machine learning è stato coniato nel 1959 da Arthur Samuel che lo descrisse come "un campo di studio che offre ai computer la possibilità di apprendere senza essere programmati esplicitamente per farlo".

Oggigiorno una delle definizioni più utilizzate per descrivere l'apprendimento automatico, è quella proposta da Mitchell in [5]:

> Si dice che un programma apprende dall'esperienza E con riferimento ad alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E.

Ovvero, ciò che si evince dalla definizione, è di riuscire a far eseguire un particolare *task* ad una macchina che riesce a migliorare in maniera totalmente autonoma la sua esecuzione, utilizzando solamente l'esperienza che acquisisce nel farlo. In questo modo la macchina impara ad eseguire un certo compito in modo sempre più accurato e preciso, senza che sia stata programmata specificatamente per farlo.

Inoltre, Mitchell nella definizione individua tre entità distinte che devono essere identificate ogniqualvolta si vuole definire correttamente un problema di apprendimento:

- T, la classe dei task che il programma dovrà svolgere
- P, le metriche di valutazione
- E, l'esperienza

Da quanto visto finora si deduce che l'apprendimento, per una macchina, è un processo iterativo e induttivo, cioè che continua a ripetersi migliorando

le prestazioni del modello, tramite la deduzione di un insieme di regole a partire dall'analisi di un insieme di dati.

Le prestazioni di questo tipo di algoritmi vengono particolarmente influenzate dalla rappresentazione dei dati su cui l'algoritmo andrà a operare. Qualsiasi informazione relativa a un dato che ne descrive alcune delle sue proprietà rilevanti, è detta *feature*. Inizialmente l'estrazione delle feature veniva fatta manualmente e questa richiedeva un'ottima conoscenza del dominio di applicazione dell'algoritmo per poter identificare, descrivere e implementare un metodo per l'estrazione delle feature più rilevanti. Successivamente, tramite l'utilizzo di modelli di *deep learning*, si è passati all'estrazione automatica delle feature. In questo caso è il modello che in modo totalmente automatico riesce progressivamente a riconoscere feature sempre più dettagliate a partire dai dati di input.

Una caratteristica comune a tutti gli algoritmi di apprendimento automatico è la presenza di un insieme di dati di input, detto *training set*. Al contrario, il tipo di output ottenuto dal modello e la modalità con cui vengono presentati gli esempi di apprendimento, caratterizzano i vari algoritmi che possono essere suddivisi in tre categorie o paradigmi:

1. Apprendimento supervisionato
2. Apprendimento non supervisionato
3. Apprendimento per rinforzo

1.1.1 Apprendimento supervisionato

L'apprendimento supervisionato si basa sul fornire in input alla rete un insieme di esempi (training set), dove ogni esempio è costituito da una coppia (x, y). La prima componente di ogni coppia è un vettore che contiene uno o più campi descrittivi che definiscono le caratteristiche di ogni elemento. Nella seconda componente è specificato il valore dell'etichetta associato ad ogni coppia.

L'obiettivo di questa tipologia di algoritmi è quello di apprendere una funzione f che assegni alla prima componente di un nuovo input x un'etichetta $\hat{y} = f(x)$ tale che $\hat{y} = y$.

Un ulteriore definizione di apprendimento supervisionato è quella fornita da Goodfellow et al. in [6]:

> Gli algoritmi di apprendimento supervisionato si basano su un set di dati contenente delle feature, ma ogni esempio è anche associato a un'etichetta o un target.

La struttura dell'apprendimento supervisionato basata sulle tre entità definite da Mitchell in [5] può essere espressa come segue:

- Esperienza E
  - Una serie di esempi che sono stati elaborati da un esperto, il supervisore;
  - L'esperto o supervisore classifica gli esempi individuando un particolare fenomeno interessante;
- Problema/Task T
  - Estrarre dagli esempi una descrizione compatta del fenomeno descritto;
  - La descrizione può essere successivamente sfruttata per fare delle previsioni sul fenomeno;
- Performance/Metriche di valutazione P
  - Dipende da quanto accurata è la previsione su esempi non considerati dal supervisore.

I problemi di apprendimento supervisionato si possono dividere in problemi di:

1. Classificazione: il modello cerca di riconoscere e categorizzare i dati di input assegnandogli un'etichetta $\hat{y}$ discreta, cioè che può assumere uno tra un insieme di valori predeterminati. Se i possibili valori sono due si parla di classificazione binaria, altrimenti classificazione multi-etichetta;

2. Regressione: il modello studia la relazione tra due o più variabili predittive e indipendenti tra loro, con la variabile target (etichetta) $\hat{y}$ che assume valori appartenenti al continuo. Questo significa che il valore dell'etichetta non rientra in un insieme di possibilità, ma può essere determinato in modo più flessibile.

Per valutare correttamente le prestazioni di un modello si utilizzano degli esempi che non fanno parte del training set. In particolare, a partire dall'insieme dei dati iniziali, viene costituito un altro insieme, disgiunto da quello utilizzato per l'apprendimento, detto *test set*. Quest'ultimo viene utilizzato per valutare il modello e capire quanto l'algoritmo sia riuscito a generalizzare il problema. La metrica utilizzata per la valutazione di un modello di classificazione sul test set è l'accuratezza, ovvero la percentuale di esempi che sono stati etichettati correttamente.

Questi algoritmi sfruttano i principi della distribuzione matematica e della funzione di verosimiglianza. Infatti, una volta allenato un modello, una possibile misura della sua accuratezza sul test set è data dalla *verosimiglianza a posteriori*, che misura la probabilità che il modello assegni i valori corretti delle etichette, a partire dalle feature di input.

Nei modelli lineari vengono definiti i parametri $\Theta = \{W, b\}$ ed è su questi che viene definita la funzione f_Θ da apprendere:

$$\hat{y} = f_\Theta(x) = Wx + b \tag{1.1}$$

dove $W \in \mathbb{R}^{k \times n}$ e $b \in \mathbb{R}^k$. I parametri $w_{ij} \in W$ sono chiamati *pesi*, mentre il vettore b è detto *bias*.

Per poter stimare i parametri Θ si utilizza una funzione $L(\hat{y}, y)$, detta *loss function* (o *funzione di perdita*), che quantifica il grado di apprendimento del modello, ovvero quanto le sue previsioni $\hat{y}$ si avvicinano al valore reale y. Esistono molteplici tipologie di loss function e la scelta corretta di queste funzioni può aiutare il modello ad apprendere meglio; al contrario, la scelta di una funzione sbagliata potrebbe portare il modello a non apprendere nulla di significativo.

La *cross-entropy loss function* è una fra le funzioni più comunemente utilizzate per i problemi di classificazione:

$$L(\hat{y}, y) = -\sum_{i \in C} y_i log(\hat{y}_i) \tag{1.2}$$

Questa misura la dissomiglianza tra il valore target reale y e la predizione $\hat{y}$, dove C è l'insieme delle k possibili classi tra cui scegliere.

Mentre per quanto riguarda i problemi di regressione, la funzione generalmente più utilizzata è l'*MSE* (*Mean Square Error* o *errore quadratico medio*), che è ottenuto dalla somma delle distanze al quadrato tra il valore reale y e quello predetto $\hat{y}$:

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n} \tag{1.3}$$

Questa funzione è particolarmente utile se si ritiene che i valori target $\hat{y}$, condizionati dall'input, siano normalmente distribuiti intorno a un valore medio, penalizzando i valori anomali.

Quando si utilizza la cross-entropy loss function è necessario che l'output sia normalizzato tramite la funzione *softmax* (o *normalizzazione sigmoidale*):

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_i}} \tag{1.4}$$

dove $z = (z_1, ..., z_k) \in \mathbb{R}^k$ è un vettore di reali di dimensione k.

Questa funzione viene applicata all'equazione 1.1 per far sì che il vettore $\hat{y}$ restituito abbia valori compresi nell'intervallo $[0, 1]$ e la cui somma totale sia pari a 1. In questo modo è possibile interpretare il vettore di output $\hat{y}$

normalizzato come una distribuzione di probabilità, in cui più il valore di un elemento $\hat{y}_c$ del vettore $\hat{y}$ è vicino a 1 e più è probabile che l'esempio di input x appartenga alla classe c.

Nella categoria di apprendimento supervisionato, avendo a disposizione un training set di dati etichettati, una loss function e il risultato del modello parametrico, si definisce la *train loss* L rispetto ai parametri Θ, come la media del valore della loss function su ciascun esempio di training:

$$L(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}^i, y^i) \tag{1.5}$$

L'obiettivo da raggiungere per l'algoritmo di apprendimento è quello di trovare il valore dei parametri $\hat{\Theta}$ che minimizzino la train loss $L(\Theta)$.

L'algoritmo di *backpropagation* fa parte degli algoritmi di apprendimento supervisionato e si basa sul metodo della discesa del gradiente per riuscire a trovare il minimo della funzione d'errore, rispetto ai pesi della rete.

Il metodo della discesa del gradiente si fonda sul fatto che il gradiente indica la direzione di massima crescita (o decrescita se viene considerato nel verso opposto) di una funzione di più variabili. Nella sua implementazione base, si inizia scegliendo un punto casuale nello spazio multidimensionale, valutando il gradiente in tale punto. Successivamente si sceglie un secondo punto in base alla direzione di decrescita indicata dall'antigradiente. Se la funzione valutata nel secondo punto ha un valore inferiore a quello calcolato dalla funzione nel punto precedente, allora si può continuare la discesa seguendo la direzione indicata dall'antigradiente, altrimenti il passo di spostamento viene ridotto e si ricomincia. Per l'applicazione di questo metodo è necessario che la funzione di loss sia differenziabile, per poterne calcolare la derivata. L'algoritmo si ferma quando raggiunge un punto di minimo, sia esso locale o globale.

Diversi sono gli algoritmi basati sulla discesa del gradiente, che possono essere usati per risolvere il problema di ottimizzazione dei parametri Θ. Tra questi, alcuni di quelli più largamente utilizzati, descritti in [7], sono:

- *AdaGrad* (Adaptive Gradient) [8]: è un algoritmo in grado adattare il *learning rate* (o *tasso di apprendimento*) ai parametri, così da ottenere grandi aggiornamenti per i parametri associati a feature rare e piccoli aggiornamenti per i parametri associati a feature più comuni. Nella formula di aggiornamento dei pesi, il tasso di apprendimento η viene modificato ad ogni passo temporale t per ogni parametro Θ_i, in base al valore dei gradienti passati:

$$\Theta_{t+1} \leftarrow \Theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \circ g_t \tag{1.6}$$

  dove G_t è una matrice in cui ciascun elemento rappresenta la somma dei quadrati del gradiente calcolata rispetto a Θ_i fino al passo t-esimo. L'operatore $\circ$ simboleggia il prodotto fra la matrice G_t e il vettore g_t. Il principale vantaggio di AdaGrad è l'eliminazione della regolazione manuale del tasso di apprendimento η. Lo svantaggio maggiore deriva dall'accumulo dei gradienti al quadrato nel denominatore. Siccome ogni termine che si aggiunge è positivo, la somma totale continuerà a crescere durante la fase di training, facendo sì che il tasso di apprendimento diventi monotonicamente decrescente.

- *RMSProp* (Root Mean Square Propagation): è un algoritmo non pubblicato, proposto da Geoff Hinton, il cui sviluppo è stato guidato dalla necessità di risolvere il problema del tasso di apprendimento radicalmente decrescente di AdaGrad. Anche in questo metodo il tasso di apprendimento è adattabile e la formula per l'aggiornamento dei pesi è la seguente:

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \Theta_{t+1} &\leftarrow \Theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \circ g_t \end{aligned} \tag{1.7}$$

  dove la media $E[g^2]_t$ relativa al t-esimo istante, dipende esclusivamente dalla media precedente e dall'attuale gradiente.

- *Adam* (Adaptive Moment Estimation) [9]: è un altro metodo che calcola il tasso di apprendimento adattabile per ciascun parametro. In particolare, oltre a memorizzare la media pesata delle precedenti radici dei gradienti v_t, come RMSProp, conserva anche una media pesata dei precedenti gradienti m_t:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{1.8}$$

  dove m_t e v_t sono stime, rispettivamente, della media e della varianza non centrata del gradiente. Quando m_t e v_t vengono inizializzati come vettori di zeri, questi tendono ad avere inclinazioni verso lo zero, specialmente all'inizio quando i tassi di decadimento sono piccoli. Per contrastare questi bias si calcolano le stime corrette di m_t e v_t:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{1.9}$$

  Infine, la formula di aggiornamento dei parametri è calcolata come:

$$\Theta_{t+1} \leftarrow \Theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \circ \hat{m}_t \tag{1.10}$$

  I valori predefiniti che vengono proposti sono 0.9 per β_1, 0.999 per β_2 e 10^{-8} per ϵ.

Il numero di *epoche* definisce quante iterazioni l'algoritmo effettua sul training set, prima di restituire il valore aggiornato per i parametri. Un'alternativa allo specificare il numero di epoche può essere indicare il numero di iterazioni che rappresentano la quantità di *mini-batch* sui quali iterare. Ovvero, ad ogni epoca l'algoritmo estrae un insieme di esempi dal training set, chiamato mini-batch, e lo utilizza per stimare il valore della train loss e aggiornare i parametri. La dimensione del mini-batch può variare e può

essere definita dall'utente come un iperparametro. È bene notare che la scelta della grandezza dei mini-batch può influenzare anche altri iperparametri, primo fra tutti il learning rate. Infatti, in alcuni esperimenti questa interazione può rendere difficile comprendere realmente qual è l'effetto causato dalla sola modifica della dimensione dei mini-batch.

Una volta appresi i parametri Θ, per verificare la bontà del modello ottenuto successivamente alla fase di training si utilizza come metrica l'errore. Quest'ultimo viene calcolato sia sul training set che sul test set, in quanto un buon modello deve riuscire a ottenere un basso valore d'errore in entrambe le valutazioni. Se questo non avviene, significa che il modello non ha acquisito la giusta capacità di generalizzazione e pertanto ci si trova davanti ad uno dei seguenti fenomeni:

- *overfitting*: il modello si è adattato eccessivamente al training set ottenendo un train error basso, ma la differenza tra quest'ultimo e il test error è alta;
- *underfitting*: il modello non ha ottenuto buone prestazioni sul training set, raggiungendo un alto valore nel train error, mentre la differenza con l'errore calcolato sul test set è bassa.

Esistono diverse tecniche per arginare il problema dell'overfitting nei modelli neurali, come, ad esempio, l'*early stopping* e il *weight decay*.

Il metodo dell'early stopping si basa sull'idea di interrompere anticipatamente il processo di apprendimento sapendo che da una certa iterazione dell'algoritmo di aggiornamento dei pesi in poi, sebbene il training error tenda a diminuire, il test error inizierà ad aumentare. Pertanto, arrestando nel punto giusto il processo di apprendimento, si cercherà di ottenere il miglior risultato possibile per un dato modello.

Il weight decay (o *regolarizzazione* l_2 *o* L_2) è un termine di regolarizzazione che cerca di portare la rete verso delle configurazioni caratterizzate da pesi minori. Per far questo viene introdotto un termine, detto *regolarizzatore*, che moltiplicato per un opportuno coefficiente di decadimento, per-

mette al modello di acquisire una maggior generalizzazione rispetto al task considerato.

È bene notare che per verificare l'andamento dell'errore durante la fase di training del modello, è necessario utilizzare un insieme di dati differente rispetto al test set per non alterarne la valutazione finale. A questo scopo viene creato un terzo insieme di dati disgiunto sia dal test set che dal training set, chiamato *validation set*.

1.1.2 Apprendimento non supervisionato

Nell'apprendimento non supervisionato, al contrario di quello supervisionato, si hanno in input dei dati senza etichetta. In questo caso l'idea è di osservare la struttura dei dati per riuscire a estrapolare delle informazioni, relazioni o schemi interessanti (*knowledge discovery*).

Diverse sono le tecniche che vengono utilizzate per affrontare questo tipo di problemi. Due esempi canonici sono il clustering e la riduzione della dimensionalità dei dati.

La clusterizzazione non supervisionata consiste nella suddivisione dei dati in un certo numero di gruppi (o *cluster*). Gli elementi del dataset che risulteranno essere più simili fra loro, faranno parte dello stesso cluster, mentre quelli meno simili verranno inseriti in cluster diversi. La bontà del risultato dipende dal tipo di metrica scelta, in quanto tutte le tecniche di clustering si basano sul concetto di similarità e quindi di distanza tra due elementi. Una delle funzioni di distanza più utilizzate è la *distanza di Minkowki*, che può essere considerata una generalizzazione sia della *distanza euclidea* che della *distanza di Manhattan*. È bene notare, però, che non tutte le distanze sono delle metriche valide che possono essere utilizzate.

In generale, le tecniche di clustering possono seguire due diverse strade:

- *Bottom-up*: si parte supponendo che tutti i dati di input siano considerati un cluster a sé. Successivamente, sarà poi compito dell'algoritmo quello di raggruppare i cluster più vicini. L'algoritmo procede iterativamente ad unire elementi al cluster fino ad ottenerne un numero

prefissato oppure fino a quando la distanza fra i cluster non supera una certa soglia;

- *Top-down*: inizialmente tutti gli elementi sono raggruppati in un unico cluster. Sarà poi l'obiettivo dell'algoritmo dividere il cluster in tanti altri di dimensioni inferiori, cercando di raggruppare elementi fra loro omogenei. Anche in questo caso l'algoritmo procede iterativamente fino ad ottenere un numero prefissato di cluster.

La riduzione della dimensionalità dei dati viene fondamentalmente utilizzata in due casi: per rappresentare i dati ad elevata dimensionalità e durante la fase di elaborazione delle feature. Nel primo caso, l'idea è quella di ridurre la dimensione dei dati proiettandoli su un sottospazio di dimensione inferiore, così da riuscire a catturare le caratteristiche chiave dei dati stessi. Il metodo più comunemente utilizzato per questo scopo è la *PCA* (*Principal Component Analysis*). Nel secondo caso, invece, l'obiettivo è quello di eliminare il rumore dai dati.

Un'altra definizione di apprendimento non supervisionato è quella fornita da Goodfellow et al. in [6]:

> Gli algoritmi di apprendimento non supervisionato si basano su un insieme di dati contenente molte feature, quindi apprendono proprietà utili direttamente della struttura di questo insieme di dati. Nel contesto del *deep learning*, di solito desideriamo conoscere l'intera distribuzione di probabilità che ha generato un insieme di dati.

Uno dei vantaggi dell'apprendimento non supervisionato rispetto a quello supervisionato, è la facilità di reperire i dati. Spesso è più facile ottenere grandi quantità di dati senza etichetta rispetto a quelli etichettati, in quanto per quest'ultimi può anche essere richiesto l'intervento umano. Nonostante questo, costruire dei modelli prestanti per l'apprendimento non supervisionato è, generalmente, più complesso. Inoltre, anche la valutazione delle prestazioni di un algoritmo di apprendimento non supervisionato può

rivelarsi più complicata rispetto a quella di un algoritmo di apprendimento supervisionato.

Apprendimento semi-supervisionato

In questo approccio di apprendimento automatico fra tutti i dati presenti nei training set, solo pochi di essi sono stati etichettati. L'idea alla base di questa metodologia è che i dati senza etichetta utilizzati insieme ad una piccola quantità di dati etichettati, potrebbero migliorare l'accuratezza dell'apprendimento.

1.1.3 Apprendimento per rinforzo

L'apprendimento per rinforzo si pone l'obiettivo di costruire un sistema (o *agente*) che tramite le interazioni con l'ambiente circostante, migliori le proprie prestazioni. In particolare, per ogni decisione che viene presa dall'agente, l'ambiente gli fornisce un feedback numerico detto *reward*. Questo può essere considerato come una misura della bontà della scelta appena presa. L'obiettivo dell'agente è di massimizzare il feedback ricevuto nel tempo.

Più precisamente, l'agente e l'ambiente interagiscono fra di loro ad intervalli di tempo discreti, $t = 0, 1, 2, 3, ..., n$. Ad ogni istante t, l'agente riceve una precisa rappresentazione dello stato dell'ambiente $s_t \in S$, dove S è l'insieme degli stati possibili. In base a queste informazioni l'agente sceglie un'azione $a_t \in A(s_t)$, dove $A(s_t)$ è l'insieme di tutte le azioni possibili nello stato $_t$. Nell'iterazione successiva, l'agente riceve il premio numerico $r_{t+1} \in \mathbb{R}$ e si sposta nel nuovo stato s_{t+1}.

L'insieme delle relazioni che l'agente implementa tra stati e probabilità di scegliere una delle azioni disponibili per quello stato, viene detta *politica* e denotata con π, dove $\pi_t(s, a)$ rappresenta la probabilità che l'azione a venga intrapresa nello stato s al tempo t.

Infine, nel caso più semplice, il reward totale è dato dalla somma dei reward ottenuti ad ogni istante temporale: $\sum_{i=0}^{i=T} r_{t+i}$.

Questo metodo di apprendimento fonda le sue radici nello studio del comportamento animale e su una precisa teoria psicologica, definita in [10]:

> Applicare una ricompensa immediatamente dopo il verificarsi di una risposta aumenta la sua probabilità che si ripresenti, mentre fornire una punizione dopo la risposta diminuirà la probabilità (di ripresentarsi).

Una definizione di apprendimento per rinforzo è quella fornita da Sutton e Barto in [11]:

> L'apprendimento per rinforzo è imparare cosa fare - come mappare le situazioni alle azioni - in modo da massimizzare una ricompensa numerica. Al discente non viene detto quali azioni intraprendere, ma deve invece scoprire quali azioni producono la maggiore ricompensa provandole.

1.2 Deep learning

Come visto in precedenza, diversi sono i problemi nel campo dell'intelligenza artificiale che possono essere risolti progettando precisamente un insieme di feature da estrarre per quel determinato compito, fornendole direttamente ad un algoritmo di apprendimento automatico. Tuttavia, per alcune attività è difficile riuscire a stabilire quali feature, tra tutte quelle estraibili, siano quelle veramente rilevanti.

Una possibile soluzione a questo problema è attraverso l'utilizzo del machine learning per riconoscere direttamente qual è la rappresentazione corretta dei dati in input. La branca del machine learning che si occupa di questo compito è chiamata *representation learning* (o *feature learning*). Gli algoritmi di representation learning a loro volta possono essere suddivisi in supervisionati e non supervisionati, in base alla presenza di etichette nei dati di input. Il representation learning, al contrario del machine learning, permette di creare modelli che sono in grado di gestire una moltitudine di tipi di dato, sia strutturati che no, comprese le immagini e i testi.

Purtroppo, il representation learning non è più sufficiente quando le feature da estrarre sono di alto livello, ovvero particolarmente soggette a variazioni. Una tecnica per la risoluzione di questo problema è tramite l'utilizzo del *deep learning*, una branca del representation learning. I modelli di deep learning sono formati da diversi *strati* o *livelli*, detti *layer*, dove ogni layer prende in input l'output del precedente. Grazie ai molteplici strati si riesce a rappresentare i dati con un livello progressivo di dettaglio sempre maggiore: nei livelli più bassi si riescono a riconoscere i pattern più semplici, mentre nei livelli più avanzati si gestiscono gli aspetti di più alto livello.

Il deep learning trova fondamento biologico nella struttura presente nei neuroni all'interno del cervello umano, dove i dati non vengono elaborati in un unico punto, ma è presente una fitta rete di nodi che collaborano tra di loro per raggiungere un obiettivo comune. Cercare di rappresentare queste complesse connessioni era l'obiettivo iniziale che ci si era posti con lo sviluppo delle prime *reti neurali*. Gli archi di una rete corrispondono alle *sinapsi*, ovvero gli impulsi nervosi che vengono trasmessi da un neurone all'altro, mentre i nodi corrispondono ai singoli *neuroni*, di cui un esempio è riportato in figura 1.2.

1.2.1 Reti neurali artificiali

Una *rete neurale artificiale* (o *Artificial Neural Network*, *ANN*) è composta da nodi collegati fra loro da archi diretti. Ogni neurone ha diversi collegamenti, ognuno con un valore di *input* a_i e un certo *peso* $w_{i,j}$, dove i indica l'indice del neurone da cui provengono questi valori e j è l'indice del neurone corrente. Fra i vari archi entranti ne esiste uno fittizio, denotato nella figura 1.2 con a_0, il cui valore è 1 e con un peso associato $w_{0,j}$. Questo viene detto *bias* e ha l'obiettivo di rendere il modello più flessibile dopo la fase di training.

Ogni singolo neurone calcola una somma pesata fra i suoi input:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i \tag{1.11}$$

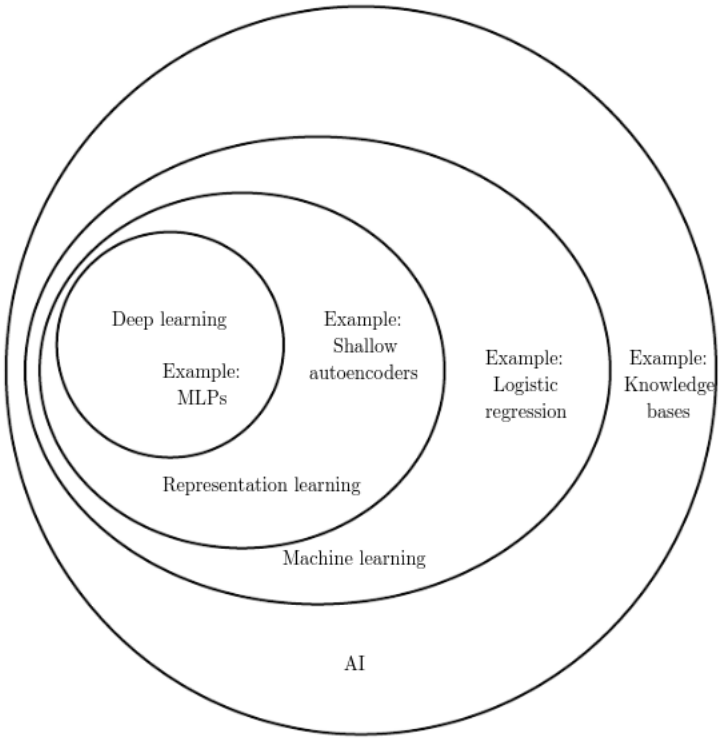


Figura 1.1: Diagramma di Eulero-Venn raffigurante le relazioni fra le varie metodologie di apprendimento, insieme ad un modello di esempio per ognuna di esse, come riportato in [6].

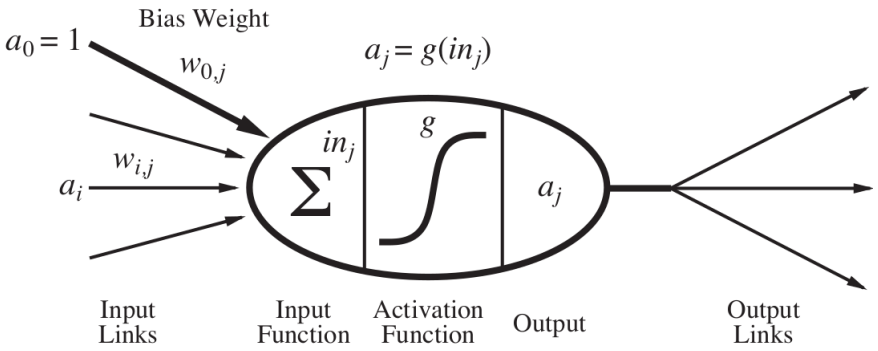


Figura 1.2: L'output del neurone j è pari a $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, dove a_i è il valore di output del neurone i e $w_{i,j}$ è il valore del peso del collegamento fra il neurone i e questo neurone, come illustrato in [12].

Se il valore di questa somma eccede una certa soglia, il neurone applica una *funzione di attivazione* non lineare g alla precedente sommatoria, per ricavare il valore di output:

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{1.12}$$

In base al tipo di funzione di attivazione utilizzata, si otterrà un certo tipo di neurone. Ad esempio, nel caso di una *funzione threshold* (o *step function*), il neurone prende il nome di *percettrone*; mentre nel caso di una *funzione logistica* il neurone viene chiamato *percettrone sigmoide*.

Per ottenere una rete neurale i vari neuroni vanno collegati fra di loro ed esistono fondamentalmente due modi per farlo:

- Reti neurali feed-forward
- Reti neurali ricorrenti

Reti neurali feed-forward

Le *reti neurali feed-forward* sono le reti con la topologia più semplice in quanto le connessioni vanno solamente in una direzione e, quindi, la rete assume la forma di un grafo diretto aciclico.

La rete è organizzata a livelli: ogni neurone di un certo livello riceve input solo dai neuroni del livello precedente e propaga i suoi output verso i neuroni dei livelli successivi. In base al numero di livelli si possono distinguere le reti a strato singolo, dove ogni neurone si collega direttamente dagli input della rete ai suoi output; e le reti multistrato, in cui si hanno uno o più strati di neuroni nascosti (o *hidden units*), ovvero non direttamente collegati all'output della rete.

Il percettrone può essere considerato come il più semplice modello di rete neurale feed-forward a strato singolo. Il *teorema di convergenza del percettrone* definito da Rosenblatt in [13], assicura che il percettrone riuscirà sempre a delimitare due classi in un sistema *linearmente separabile*. Al contrario, per quanto concerne i casi *non linearmente separabili*, è necessario utilizzare più percettroni disposti su più livelli, ovvero il *percettrone multistrato* (o *Multilayer Perceptron*, *MLP*).

In particolare, prendendo come esempio la figura 1.3, gli output della rete neurale avranno la forma:

$$y_k(x) = \tilde{g}(\sum_{j=0}^{M} w_{k,j}^{(2)}\ g(\sum_{i=0}^{d} w_{j,i}^{(1)} x_i)) \tag{1.13}$$

dove:

- $\tilde{g}()$ e $g()$ sono le funzioni di attivazione della rete, dell'ultimo livello e del livello nascosto, rispettivamente;
- $w_{j,i}^{(1)}$ indica il peso del primo livello: dall'input i-esimo alla j-esima unità nascosta. Fra questi pesi $w_{j,0}^{(1)}$ denota il bias per la j-esima unità nascosta;
- $w_{k,j}^{(2)}$ indica il peso del secondo livello: tra la j-esima unità nascosta e il k-esimo output. Fra questi pesi $w_{k,0}^{(2)}$ denota il bias.

Le reti multistrato sono in grado di trattare un numero maggiore di funzioni rispetto alle reti a singolo strato grazie alla loro maggior espressività. Purtroppo, il teorema di convergenza non si estende a questo tipo di reti.

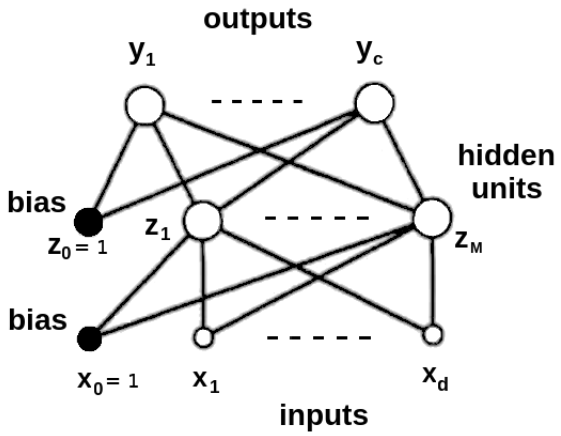


Figura 1.3: Un esempio di rete feed-forward con due livelli di pesi adattabili [14]. Il bias nel primo livello è mostrato come un peso proveniente da un input aggiuntivo avente un valore fisso $x_0 = 1$. Allo stesso modo, il bias presente nel secondo livello è mostrato come un peso proveniente da un'unità nascosta extra, con valore $z_0 = 1$.

Tuttavia, secondo il *teorema di Kolmogorov* del 1957, una rete di percettroni con un unico stato nascosto dovrebbe essere in grado di rappresentare qualsiasi funzione continua, approssimandola con qualsiasi grado di precisione semplicemente aumentando il numero dei neuroni che compongono lo strato. Nonostante ciò, questo teorema applicato alle reti neurali è ancora oggetto di dibattito e, in ogni caso, esso non fornisce alcun tipo di indicazione su come debba essere la struttura della rete neurale per risolvere un certo task.

Per l'addestramento dei percettroni multistrato si utilizza l'algoritmo di *backpropagation* (o *retropropagazione dell'errore*), chiamato in questo modo in quanto non vengono effettuati dei passi iterativi, ma è un procedimento a ritroso: dai nodi di output della rete fino a quelli di input.

Nel caso in cui ci si trovi in uno scenario di apprendimento supervisionato, attraverso il confronto tra i valori di output della rete e quelli dell'etichetta presente nel training set, è possibile stabilire l'errore dei percettroni dello strato di uscita. Al contrario, non c'è alcun tipo di indicazione su quali debbano essere i valori di output dei nodi interni alla rete. Pertanto, per

determinare l'errore di un percettrone nascosto è necessario retropropagare l'errore a partire dallo strato di output.

Uno dei problemi legato all'addestramento delle reti neurali deep è il problema della *scomparsa/esplosione del gradiente* (o *vanishing/exploding gradient problem*). Considerata la grande quantità di pesi di una rete neurale deep, può succedere in alcuni casi che il gradiente assuma un valore prossimo allo zero o, viceversa, diventi eccessivamente alto. Per evitare questo tipo di problema è bene utilizzare particolari funzioni di attivazione rettificate, come la *ReLU* (*Rectified Linear Unit*) e usare strategie di *batch-normalization*, dove, per ogni mini-batch, vengono normalizzati gli input di ogni livello della rete in modo da ottenere media zero e varianza uno, come descritto in [15].

Nella fattispecie, la ReLU mappa i valori negativi a zero, mantenendo invariati quelli positivi:

$$f(x) = \begin{cases} x, & \text{se } x \geq 0 \\ 0, & \text{se } x < 0 \end{cases} \tag{1.14}$$

Reti neurali ricorrenti

Le *reti neurali ricorrenti* (*Recurrent Neural Networks, RNN*) sono un particolare tipo di rete neurale in cui sono presenti delle connessioni di feedback, ovvero dei cicli all'interno della struttura della rete. La peculiarità di queste reti è la capacità di poter avere l'ordine dei dati e la memoria dei precedenti input. Teoricamente le RNN possono utilizzare informazioni in sequenze arbitrariamente lunghe, ma in pratica sono limitate a "guardare indietro" solo di qualche step.

Queste caratteristiche risultano fondamentali nella stragrande maggioranza dei task di *elaborazione del linguaggio naturale* (o *Natural Language Processing*, *NLP*), in cui si affrontano problematiche relative all'elaborazione e comprensione del linguaggio naturale.

Dal punto di vista del grafo computazionale, è possibile eseguire il cosiddetto *unfolding* della struttura della rete ricorrente, come mostrato in

figura 1.4, in modo da ottenere una sorta di rete feed-forward di lunghezza arbitraria.

L'idea alla base delle reti ricorrenti è quella di condividere i parametri Θ, ovvero pesi e bias, attraverso i vari stati in cui la rete si trova dopo aver analizzato una sequenza di input.

Il vettore di output $\hat{y}^t$ di una RNN può essere calcolato come:

$$\hat{y}^t = RNN_\Theta(s^{t-1}, x^t) \tag{1.15}$$

Come si evince dalla formula, il calcolo del vettore di output dipende da molteplici fattori, in quanto prende in considerazione il t-esimo esempio di training x^t, lo stato precedente s^{t-1} della rete e i parametri condivisi Θ.

Data la natura ricorsiva delle RNN, per conoscere il t-esimo output della sequenza di input si deve analizzare l'intera sequenza. Per esempio, come descritto in [16], se si considera una sequenza di lunghezza $t = 4$ e si espandesse la ricorsione si otterrebbe:

$$\begin{aligned} s_4 &= RNN(s_3, x_4) \\ &= RNN(\overbrace{RNN(s_2, x_3)}^{s_3}), x_4) \\ &= RNN(RNN(\overbrace{RNN(s_1, x_2)}^{s_2}, x_3), x_4) \\ &= RNN(RNN(RNN(\overbrace{RNN(s_0, x_1)}^{s_1}, x_2), x_3), x_4) \end{aligned} \tag{1.16}$$

Pertanto, s_n, così come $\hat{y}_n$, potrebbe essere visto come codifica dell'intera sequenza di input.

L'esempio di figura 1.4 mostra una corrispondenza biunivoca fra input e output, ma quello che si può fare in realtà è mascherare parte degli input o degli output in modo da ottenere diverse combinazioni. Ad esempio, è possibile utilizzare una rete *many-to-one* nel caso in cui si debba classificare una sequenza di dati con un singolo output (classificazione binaria), oppure usare una *one-to-many* per generare una canzone a partire da una singola nota musicale.

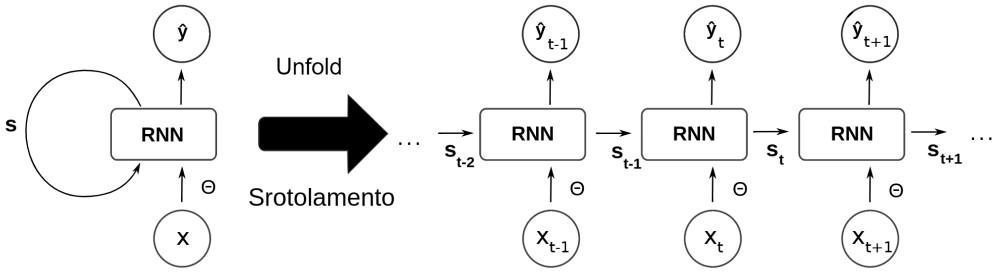


Figura 1.4: Un esempio di rete neurale ricorrente in cui sono presenti sia la sua rappresentazione ricorsiva che quella unfold.

Un caso particolare di RNN sono le *reti neurali ricorrenti bidirezionali* che vengono utilizzate per catturare un maggior numero di informazioni dalla sequenza di input. In particolare, le RNN bidirezionali sono modelli che utilizzano sia le informazioni passate, catturate dalla RNN_f che legge la sequenza in avanti, sia le informazioni future, catturate dalla RNN_b che legge l'input in direzione opposta. L'idea è di combinare l'output di entrambe le RNN per eseguire, ad esempio, la classificazione o previsione degli elementi della sequenza, come mostrato in [16]:

$$\begin{aligned} \hat{y}_f^t &= RNN_f(s_f^{t-1}, x^t) \\ \hat{y}_b^t &= RNN_b(s_b^{t-1}, x^{n-t+1}) \\ \hat{y}^t &= BiRNN(x^t) = [\hat{y}_f^t; \hat{y}_b^t] \end{aligned} \tag{1.17}$$

In [17] viene messo in evidenza come una RNN possa facilmente mappare sequenze di input in sequenze di output, ogni volta che la lunghezza delle sue sequenze coincide. Tuttavia, non è chiaro come applicare una RNN a problemi le cui sequenze di input e output hanno lunghezze diverse, con relazioni complicate e non monotone. Questa limitazione è particolarmente significativa in quanto molti problemi importanti, come ad esempio il riconoscimento vocale o la traduzione automatica, vengono espressi al meglio attraverso sequenze le cui lunghezze differiscono tra loro.

La soluzione a questo tipo di problema è tramite l'utilizzo di una particolare architettura detta *encoder-decoder* o *sequence-to-sequence*, sviluppata

e introdotta da Google nel 2014 [17].

La struttura dell'architettura encoder-decoder, come si evince dal nome, è composta da due elementi: l'encoder e il decoder.

L'encoder è formato da, generalmente, uno stack di diverse unità ricorrenti in cui ciascuna accetta un singolo elemento della sequenza di input, ne raccoglie le informazioni e le propaga in avanti nella rete. Si noti che in questo caso l'output delle unità ricorrenti dell'encoder viene scartato e ciò che si preserva è solo il suo stato.

Lo stato nascosto finale ottenuto dall'encoder, detto *contesto* c, è un vettore di lunghezza fissa che incapsula, in modo compresso, tutte le caratteristiche più significative derivanti dagli elementi di input. Inoltre, funge da stato nascosto iniziale per il decoder.

Il decoder, costituito anch'esso, generalmente, da uno stack di unità ricorrenti, interpreta il contesto e lo utilizza per generare la sequenza di output. Ogni unità ricorrente genera un elemento di output, durante una certa fase temporale. La forza e l'utilità di questo modello risiede nella capacità di riuscire a mappare sequenze di lunghezze diverse tra loro. Questo ha dato la possibilità a tanti problemi, in particolare nel campo dell'NLP, di essere risolti utilizzando questa architettura.

Una delle maggiori limitazioni degli encoder-decoder si verifica nel momento in cui la sequenza di input è troppo lunga per poter essere adeguatamente riassunta all'interno del vettore di lunghezza fissa c. Una metodologia efficiente per gestire questo problema è attraverso l'uso dell'*attention* [18]. L'idea è che l'encoder durante la scansione delle parole nella sequenza di input, invece di scartare tutte le informazioni contenute negli stati nascosti, ad eccezione dell'ultimo (contesto c), crei una sorta di memoria con le informazioni ottenute, salvando tutti gli stati nascosti prodotti. Successivamente, al momento della decodifica, invece di fare affidamento solo sullo stato nascosto finale dell'encoder, il decoder produce una parola alla volta, prestando attenzione, a ogni step, a una specifica parte della sequenza di input. Ovvero, come illustrato in figura 1.5, ogni output prodotto dal decoder dipende da

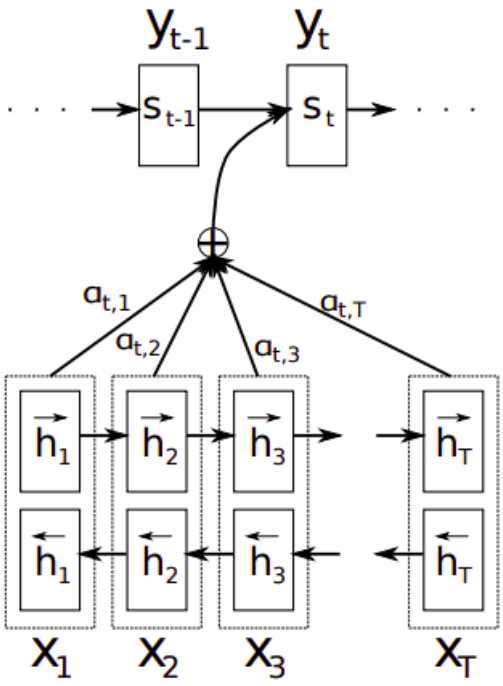


Figura 1.5: Illustrazione di una architettura encoder-decoder con attention [18], dove le y_i corrispondono agli output del decoder, le x_i sono le parole della frase di input e le a_i sono i pesi dell'attention.

una combinazione pesata di tutti gli stati nascosti dell'encoder e non solo dall'ultimo.

Come precedentemente evidenziato nelle reti neurali deep, anche con le RNN ci si ritrova di fronte al problema del vanishing/exploding gradient, che compromette l'efficacia delle RNN. Per contrastare questo problema, vengono utilizzate delle unità RNN con una struttura particolare chiamate *Long Short-Term Memory* (LSTM) [19] e *Gated Recurrent Unit* (GRU).

La LSTM presenta tre *gate*, i quali sono i responsabili delle modifiche alla memoria o all'output. Il *forget gate* è l'unità deputata a decidere quali parti di informazioni contenute nello stato vanno dimenticate, mentre l'*input gate* si occupa delle parti dello stato che l'unità deve ricordare. Infine, l'*output gate* decide quali parti del nuovo stato considerare per restituire l'output della cella.

Tutte le sopracitate tipologie di gate sono formate da una combinazione lineare tra l'input x_t e lo stato precedente s_{t-1} a cui è applicata una funzione di attivazione sigmoide σ.

La GRU è una versione semplificata della LSTM, che cerca di mantenerne i principali vantaggi, ma riducendone il numero di parametri e la complessità. La principale semplificazione deriva dall'utilizzo di un unico *update gate*, che combina il *forget gate* e l'*input gate* della LSTM.

Reti neurali convoluzionali

Le *reti neurali convoluzionali* (o *Convolutional Neural Networks*, *CNN*) sono un ulteriore famiglia di reti neurali e rappresentano il principale modello utilizzato nel campo della *computer vision*. In particolare, queste reti sono specializzate nell'elaborazione di dati 2D o 3D con una topologia a griglia [6]. Il loro nome deriva dalla presenza di almeno uno strato della rete che compie l'operazione matematica di convoluzione, al posto di una più semplice moltiplicazione fra matrici [6]. La convoluzione consiste nell'applicare un filtro, chiamato *kernel* (o *nucleo*), ad un array multidimensionale di input. L'altezza e la larghezza del kernel possono variare, mentre la profondità deve essere la medesima di quella dell'array di input. Quello che succede è che il nucleo viene traslato sull'array di input e in ogni posizione viene calcolato l'output come prodotto scalare tra il kernel e la posizione dell'input coperta. Ciò che scaturisce al termine di questo processo viene detta *mappa di attivazione* (o *feature map*).

Un esempio visivo di come avviene questa operazione è mostrato in figura 1.6.

I parametri dei filtri, ovvero i pesi e il bias di ciascun kernel, costituiscono i parametri addestrabili nello strato di convoluzione, durante la fase di addestramento della rete. Al contrario, ci sono alcuni iperparametri che devono essere definiti dall'utente prima della fase di training, come la dimensione dei filtri (altezza e larghezza del kernel) e il loro numero, il passo (o *stride*), ovvero il passo con cui il filtro si muove scansionando l'input in altezza e

larghezza e il padding, che permette di regolare l'altezza e la larghezza della mappa di attivazione.

All'interno dell'architettura di una generica rete neurale convoluzionale si possono trovare diverse tipologie di layer, che vengono concatenati dal programmatore con l'obiettivo di realizzare una rete il più performante possibile per un determinato compito. Alcuni dei layer utilizzati più frequentemente, oltre a quello convoluzionale, sono:

- Funzione di attivazione: lo strato di convoluzione è tipicamente seguito da una funzione di attivazione non lineare, in quanto solo le *feature* attivate vengono trasmesse al layer successivo. Una delle funzioni più utilizzate, in questo caso, è la ReLU;

- Strato di pooling: in questo strato, comunemente inserito fra due o più layer di convoluzione, si ha un'operazione di sottocampionamento con l'obiettivo di ridurre progressivamente la dimensione spaziale degli input. Questo consente di diminuire il numero di parametri e il carico computazionale. Lo strato di pooling opera sulle mappe di attivazione, applicando un filtro che esegue una certa operazione deterministica, solitamente il massimo (*max-pooling*) o la media (*avg-pooling*);

- Strato fully-connected (o *strato denso*): layer in cui tutti gli elementi sono connessi tra loro, come i livelli di un MLP. Può essere utilizzato come strato finale di una rete per un task di classificazione;

- Dropout [20]: una forma di normalizzazione molto efficiente, che può anche essere utilizzata insieme ad altre forme di normalizzazione, come la regolarizzazione l_2. Ad ogni iterazione dell'algoritmo di addestramento della rete, alcuni nodi vengono temporaneamente rimossi dal modello. Ciascun nodo viene conservato nel modello con una certa probabilità assunta da un parametro di regolarizzazione p, definito dall'utente. In questo modo la parte di nodi rimossi e quella di nodi conservati è casuale e diversa ad ogni iterazione.

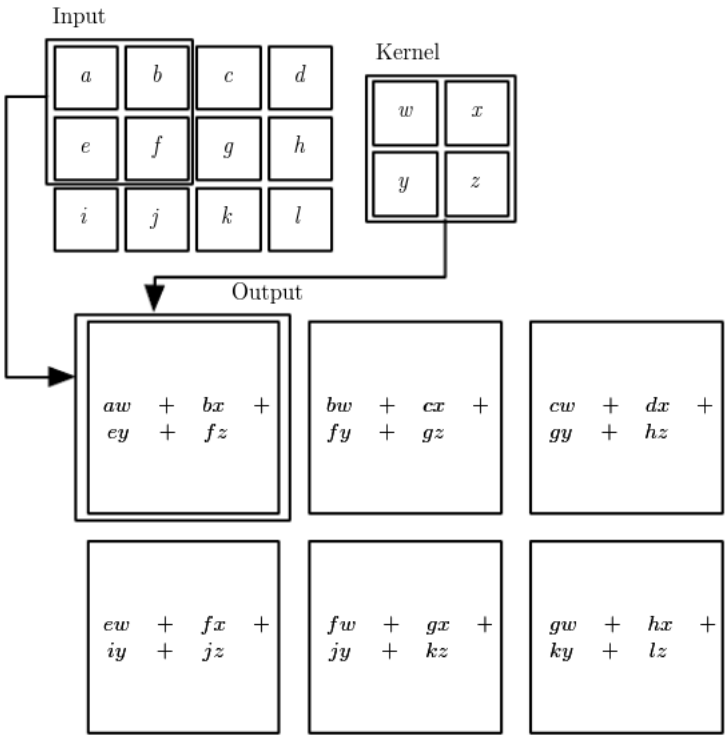


Figura 1.6: La convoluzione avviene tra la matrice 3 × 4 di input e il kernel 2 × 2. L'output risultante è una matrice 2 × 3. Le frecce mettono in evidenza il calcolo per formare il primo elemento della matrice di output. Calcolato il primo elemento, il kernel scorre sulla matrice e applicandolo alla regione che ricopre si calcola l'elemento corrispondente [6].

Sommariamente, l'architettura di una CNN può essere suddivisa in due parti: una prima parte di *feature detection*, ovvero dell'estrazione di feature attraverso operazioni come la convoluzione, la ReLU e il pooling e una parte di classificazione che genera l'output predetto dal layer fully-connected.

Embedding layer

Finora si sono sempre considerati i parametri Θ di una rete come valori numerici reali, pertanto non necessitavano di alcun tipo di modifica.

Uno dei problemi più importanti per le applicazioni nel campo dell'NLP è capire come poter rappresentare gli elementi simbolici, come ad esempio

le parole di un vocabolario, per poterli elaborare in modo automatico e apprenderli congiuntamente ai parametri Θ. Non è immediato convertire un testo in una serie di numeri reali a causa di tutti i significati intrinseci che quest'ultimo veicola: le diverse relazioni tra le parole, la sua struttura e così via.

La *semantica vettoriale* (o *vector semantics*) è il modo standard per codificare le parole rappresentandole come un punto in uno spazio semantico multidimensionale, mantenendone tutte le loro caratteristiche.

Questa trasformazione da parola a vettore d-dimensionale è ottenuta in un particolare livello della rete, chiamato *embedding layer*. L'output di questo layer è una *matrice di embedding* $E \in R^{|V| \times d}$, dove ad ogni riga della matrice corrisponde la codifica d-dimensionale di ciascuna parola presente nel vocabolario V.

L'approccio più semplice per la rappresentazione delle parole è tramite il *one-hot encoding*. In questo caso la dimensione dello spazio vettoriale generato dipende dalla dimensione del vocabolario, ovvero $d = |V|$, e ogni elemento del vettore corrisponde ad una feature unica. Questo tipo di rappresentazione viene detta *sparsa* in quanto ogni vettore è formato da tutti zero, tranne la feature che corrisponde alla parola da rappresentare che assume valore uno. Si deve inoltre considerare che questa rappresentazione non fornisce alcun tipo di informazione sulle relazioni semantiche che ci possono essere fra le diverse parole. Questo deriva dal fatto che i vettori sono ortogonali fra loro e pertanto non permettono di estrarre un valore di similarità.

La codifica che si contrappone a quella sparsa è detta *codifica densa* (o *dense encoding*). In questo caso la dimensione dei vettori è molto inferiore rispetto a quella del vocabolario V. Per ogni feature $w_i \in V$ con $1 \leq i \leq |V|$ si calcola il rispettivo vettore $\vec{v}(w_i) \in \mathbb{R}^d$. I principali vantaggi di questa codifica sono la minor occupazione di memoria da parte dei vettori e il fatto che feature simili condividono vettori simili che possono essere combinati tra loro. Inoltre, la codifica densa può essere ottenuta anche durante la fase di apprendimento della rete, insieme agli altri parametri.

Uno degli approcci più noti e utilizzati per ottenere una codifica densa è il WORD2VEC, descritto in [21], che comprende due modelli (Fig. 1.7): Continuous bag-of-words (CBOW) e Skip-gram. Entrambi sono basati sull'ipotesi di *semantica distribuzionale*, secondo la quale

> il grado di similarità semantica tra due espressioni linguistiche è funzione della similarità linguistica dei contesti linguistici in cui le due espressioni appaiono [22].

- CBOW è un'architettura che effettua la predizione della parola centrale date le parole del contesto. Questo modello è basato sul *bag-of-words* in quanto l'ordine delle parole non influenza il risultato;
- Skip-gram è un'architettura che effettua il task contrario rispetto al CBOW, ovvero prende in input una parola e predice le due parole antecedenti e successive, ovvero il contesto.

Nonostante esistano altri algoritmi, oltre a quelli sopracitati, per creare word embedding, come *GloVe* (Global Vectors for Word Representation) [23] o *fastText* [24], il funzionamento è grosso modo simile:

- *GloVe* è un algoritmo di unsupervised learning che permette di ottenere una rappresentazione matriciale di un insieme di parole. L'addestramento è eseguito su statistiche aggregate globali di co-occorrenza delle parole estratte da un corpus di testo tokenizzato [23];
- *fastText* è un'estensione di WORD2VEC in cui l'input per la rete neurale non sono le singole parole, ma una loro suddivisione in diversi *n-gram*. Successivamente il vettore di embedding per una data parola sarà rappresentato dalla somma dei suoi n-gram.

Questi embedding multidimensionali si sono dimostrati molto utili nella maggior parte dei task di NLP, grazie alla loro capacità di catturare diverse informazioni sintattico/semantiche delle parole.

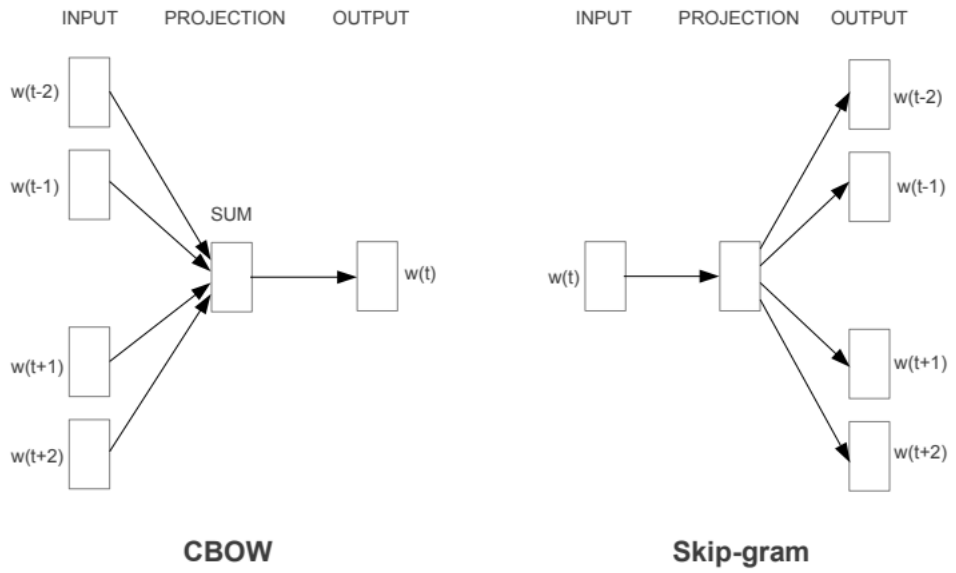


Figura 1.7: L'architettura CBOW prevede la parola corrente in base al contesto, mentre lo Skip-gram prevede le parole del contesto data la parola corrente.

Capitolo 2

Transformers

Come visto nel capitolo precedente, una delle architetture più frequentemente utilizzate come soluzione ai problemi di elaborazione del linguaggio naturale è quella *sequence-to-sequence* (o *seq2seq*). Questo tipo di architettura si basa su complesse reti neurali ricorrenti che includono un encoder e un decoder e sono soggette a diverse limitazioni, come descritto in 1.2.1. Alcuni dei problemi che possono essere risolti tramite questa architettura comprendono quelli di traduzione automatica o sintesi di lunghi testi scritti in linguaggio naturale.

Nel 2017 una nuova architettura chiamata *Transformers*, alternativa a quella *seq2seq*, è stata introdotta nell'articolo "Attention Is All You Need" [4].

I Transformers si basano esclusivamente su meccanismi di *attention* per rappresentare le dipendenze globali tra input e output, eliminando completamente l'utilizzo di reti neurali ricorrenti e convoluzionali. Inoltre, sono completamente parallelizzabili grazie al loro design.

2.1 Architettura

La maggior parte delle architetture seq2seq più prestanti sono formate da una struttura encoder-decoder. Anche i Transformers seguono questo

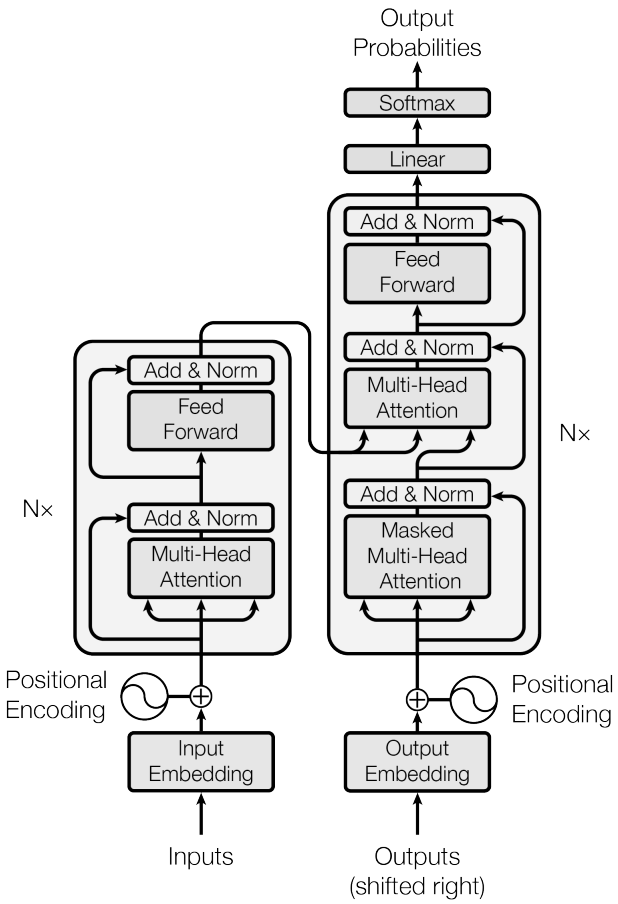


Figura 2.1: Rappresentazione della struttura di un Transformers con lo stack dell'encoder a sinistra e quello del decoder a destra [4].

schema generale, come mostrato in figura 2.1, includendo sia nell'encoder che nel decoder un meccanismo di attention.

I dati di input sono formati da sequenze di testo. Queste vengono fornite al modello attraverso la creazione di un embedding per ogni parola presente nella sequenza. La dimensione dell'embedding è un iperparametro, stabilito dall'utente.

Ogni frase verrà, quindi, elaborata come un insieme di token, ma per non perdere l'ordinamento originale delle parole viene aggiunto un *positional embedding* a quello originale. In questo modo, il modello durante l'elaborazione dei dati sarà in grado di ricostruire l'ordine originale delle parole.

Gli elementi di input precedentemente descritti vengono inviati all'encoder che li elabora all'interno del suo stack e successivamente li passa al decoder che ne restituisce in output la loro rappresentazione. Infine, sopra a questa architettura sono presenti uno strato fully-connected insieme a una softmax, in modo che l'output prodotto dal decoder possa essere convertito in un vettore in cui ogni valore fa riferimento ad una delle parole del vocabolario. La funzione softmax si occupa di trasformare questo vettore in una distribuzione di probabilità dalla quale è possibile prevedere il token successivo.

Sia lo stack dell'encoder che quello del decoder hanno lo stesso numero di layer, ovvero $N = 6$.

Nonostante l'apparente sequenzialità dell'architettura, in seguito verrà mostrato che molti dei calcoli eseguiti sono fatti in parallelo. Questo è dovuto principalmente al fatto che ogni token può attraversare l'architettura in modo indipendente rispetto agli altri. Inoltre, nessuno stack dell'encoder condivide i pesi appresi con lo stack del decoder.

2.1.1 Encoder

Più dettagliatamente, l'encoder è formato da tre diverse tipologie di layer: *self-attention*, *feed-forward network* e *add & normalization*. La self-attention è il meccanismo più importante dei Transformers e verrà trattato in dettaglio nella sezione 2.1.3.

La rete feed-forward è formata da uno stack di due livelli fully-connected con una funzione di attivazione ReLU fra i due. La trasformazione che viene applicata da questa rete è la seguente [4]:

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2 \tag{2.1}$$

dove $\Theta_i = (W_i, b_i)$ con $i \in \{1, 2\}$ rappresenta i pesi e il bias dei due layer considerati, mentre x è il punteggio normalizzato dell'attention.

Assumendo che la dimensione dell'input d_{input} sia pari a 512, il primo layer la quadruplicherà portandola a $d_{inner} = 2048$; mentre il secondo layer riporterà la dimensione al suo valore originale.

Infine, viene utilizzata una, cosiddetta, *connessione residua* attorno sia alla rete feed-forward che al livello di self-attention, seguita da un layer di normalizzazione. Lo scopo del livello *Add & Norm* è di sommare l'output elaborato dal layer precedente, ovvero $Layer(x)$, con l'input originale x, ottenuto grazie alla connessione residua, per poi applicare il layer di normalizzazione sui dati risultanti, quindi $LayerNorm(x + Layer(x))$, con l'obiettivo di normalizzare le feature.

In generale, le connessioni residue e in particolare i *residual block* rappresentano la peculiarità delle *reti neurali residuali* (o *Residual Neural Network*, *ResNet*) sviluppate nel 2015 da Microsoft [25]. Il concetto su cui si basa il residual block è quello di sottoporre un input x ad una certa sequenza di operazioni di convoluzione e ReLU, ottenendo una $F(x)$; infine, sommare al risultato ottenuto l'input originale, ovvero $H(x) = F(x) + x$.

2.1.2 Decoder

Il decoder ha una struttura sostanzialmente molto simile a quella dell'encoder. Il suo input è formato da un particolare embedding, unitamente all'output dello stack dell'encoder. Nella fattispecie, l'embedding di input è stato precedentemente shiftato a destra di una posizione e gli è stata applicata una maschera di soli zeri. Questo permette al modello di concentrarsi solo sulle parole precedenti rispetto a quella attuale, in quanto, in questa fase, non è di alcun aiuto conoscere le parole successive. In seguito, l'informazione derivante dall'embedding viene utilizzata dal meccanismo di attention per poterla incorporare con i dati provenienti dallo stack dell'encoder. Infine, la rete feed-forward e la normalizzazione dei livelli applicheranno le stesse trasformazioni descritte in precedenza.

2.1.3 Self-Attention

La *self-attention* viene definita in [4] come "un meccanismo di attenzione che mette in relazione le diverse posizioni di una singola sequenza al fine di calcolare una rappresentazione della sequenza stessa". Ancora, "senza alcuna informazione aggiuntiva, si possono estrarre gli aspetti rilevanti di una sequenza, permettendole di occuparsi della sua rappresentazione tramite la self-attention" [26].

Pertanto, come si intuisce dalle precedenti definizioni, l'idea dietro al metodo della self-attention è quella di determinare la relazione esistente tra la parola corrente e tutte le altre parole presenti nella sequenza. Per farlo è necessario calcolare tre elementi per ogni parola: Q (*Query*), K (*Key*) e V (*Value*). Questi elementi sono chiamati *embedding* e vengono formati dalla moltiplicazione tra il *word embedding* di una parola e tre diverse matrici, rispettivamente, W_Q, W_K e W_V, che il modello apprende durante la fase di training.

Ad esempio, dato il word embedding della parola w di dimensione $w_{emb} = 512$ e le matrici W_Q, W_K e W_V, apprese durante il training, i suoi embedding Q, K e V sono pari a:

$$\begin{aligned} Q &= w_{emb} * W_Q \\ K &= w_{emb} * W_K \\ V &= w_{emb} * W_V \end{aligned} \tag{2.2}$$

di dimensione, rispettivamente, d_k, d_k, d_v.

Successivamente, una volta codificati gli embedding per una parola in una certa posizione, si calcola il punteggio dell'attention, per determinare quanta attenzione porre sulle altre parole dell'input. Per farlo si valuta il prodotto scalare tra Q e K e si scala di un fattore $\sqrt{d_k}$.

Queste stesse operazioni di codifica degli embedding e del calcolo del punteggio dell'attention vengono eseguite su tutte le parole della frase.

Il passaggio successivo consiste nell'applicare la funzione softmax su tutti i risultati ottenuti così da riuscire ad avere una distribuzione di probabilità

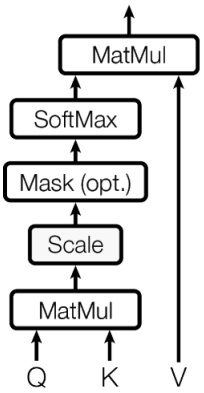


Figura 2.2: Rappresentazione grafica delle operazioni effettuate per il calcolo della Scaled Dot-Product Attention [4].

che permetta di scegliere la parola più rilevante da considerare. Infine, tutto verrà moltiplicato per V e sommato, così da ottenere il nuovo embedding per la parola corrente, come mostrato in figura 2.2.

Tutti i calcoli appena descritti possono essere sia parallelizzati che generalizzati. In particolare, la parallelizzazione è possibile in quanto ogni parola si comporta in modo indipendente dalle altre mentre la generalizzazione scaturisce dal fatto che si può utilizzare come word embedding una matrice invece che un vettore. In quest'ultimo caso è possibile elaborare un'intera frase invece che una singola parola alla volta. Dal punto di vista matematico tutti i vettori definiti in precedenza diventeranno matrici mentre i restanti calcoli possono essere riassunti dalla seguente equazione:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.3}$$

Più precisamente, come mostrato nella figura 2.1, all'interno dell'architettura dei Transformers non è presente la self-attention, bensì la *Multi-Head Attention* (Fig. 2.3). Essenzialmente, quest'ultima è formata da h livelli di self-attention impilati in parallelo, con diverse trasformazioni lineari dello stesso input. In questo modo il vantaggio è che il modello può aumentare la sua capacità di concentrarsi su parole in posizioni differenti, in quanto

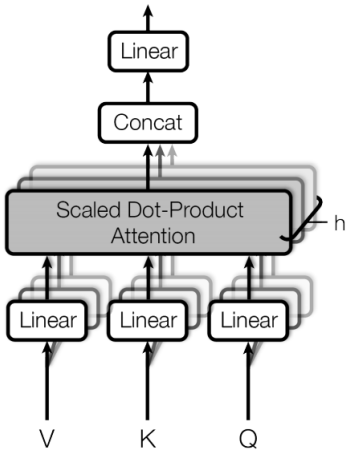


Figura 2.3: Rappresentazione dell'architettura per il calcolo della Scaled Dot-Product Multi-Head Self-Attention [4].

vengono considerate insieme più matrici Q, K e V, dette *teste* (o *heads*). La matrice di attention è ora la concatenazione delle matrici risultanti da tutte le teste disponibili.

A questo punto, tuttavia, è necessaria una nuova matrice $W_h \in \mathbb{R}^{Md_v \times d_{emb}}$ che viene appresa durante l'addestramento della rete, con lo scopo di condensare tutte le informazioni della matrice di training in una nuova che rispetti le dimensioni previste dalla rete feed-forward.

2.2 BERT

BERT (*Bidirectional Encoder Representations from Transformers*), presentato da Google nel 2019 in [1], è una nuova rappresentazione del modello linguistico che ha rivoluzionato lo stato dell'arte nel campo dell'elaborazione del linguaggio naturale. In particolare, BERT e tutti i modelli da esso derivati sono considerati lo stato dell'arte per la quasi totalità dei task di NLP.

BERT è stato rilasciato in due versioni, in cui cambiano il numero di layer (blocchi di Encoder), di self-attention head e di livelli nascosti. In particolare,

Modello		Layer	Livelli nascosti	Att. Head	Numero Parametri (M)	Lingue
$\text{BERT}_{\text{BASE}}$	Uncased	12	768	12	110	1
	Cased	12	768	12	110	1
	Multilingual Cased	12	768	12	110	104
	Multilingual Uncased	12	768	12	110	102
$\text{BERT}_{\text{LARGE}}$	Uncased	24	1024	16	340	1
	Cased	24	1024	16	340	1

Tabella 2.1: Dimensioni dei modelli $\text{BERT}_{\text{BASE}}$ e $\text{BERT}_{\text{LARGE}}$.

$\text{BERT}_{\text{BASE}}$ è formato da 12 layer, 12 self-attention head e 768 livelli nascosti, per un totale di 110 milioni di parametri. $\text{BERT}_{\text{LARGE}}$, invece, è costituito da 24 layer, 16 self-attention head e 1024 livelli nascosti, per un totale di 340 milioni di parametri.

Per entrambe queste versioni esistono delle varianti in cui è presente una distinzione tra maiuscole e minuscole e in cui è disponibile il supporto ad oltre 100 diverse lingue.

Nella tabella 2.1 sono riassunte le caratteristiche principali dei vari modelli attualmente disponibili.

Qualsiasi versione di BERT prende in considerazione entrambe le fasi che formano la sua strategia di training: la fase di *pre-training* e quella di *fine-tuning*.

Come suggerito dal suo nome, una caratteristica chiave di BERT è la bidirezionalità, ottenuta tramite l'utilizzo di più Encoder: una delle parti principali dell'architettura dei Transformers. Questo meccanismo di bidirezionalità viene utilizzato durante la fase di pre-training, consentendo a BERT di apprendere il contesto di una singola parola osservando sia le parole precedenti che quelle successive alla parola corrente. Al contrario, la parte di Decoder dei Transformers non viene mai utilizzata.

Durante la fase di fine-tuning, BERT viene inizializzato con i parametri ottenuti dalla fase di pre-addestramento che verranno poi regolarizzati sulla

base del compito che andrà a svolgere.

Entrambe queste fasi, con i loro rispettivi task, saranno descritte più in dettaglio nelle sezioni successive.

Una caratteristica molto importante di BERT è la sua versatilità. Infatti, il modello è stato progettato in modo tale che la sua architettura fosse quanto più generale possibile, così da permettergli di affrontare una vasta gamma di compiti. Per rendere possibile questa sua duttilità, sono stati introdotti due token che aiutano il modello nei vari task. In particolare:

- il token *[CLS]* segna l'inizio di una frase;
- il token *[SEP]* segna la fine di una frase e funge da separatore quando più frasi vengono unite in un'unica sequenza.

Inoltre, anche la rappresentazione dell'input è stata creata ad-hoc per questo modello e consiste nella somma fra *token embedding*, *positional embedding* e *sequence embedding* (o *segment embedding*) per ogni sottoparola, come mostrato in figura 2.4. Il positional embedding è lo stesso presente nel modello Transformers mentre il sequence embedding viene utilizzato da BERT per capire a quale frase appartiene un certo token.

L'algoritmo di tokenizzazione usato da BERT è chiamato *WordPiece model* [27] ed è un algoritmo simile al *Byte-Pair Encoding* (BPE). Il vocabolario viene inizializzato con i singoli caratteri del linguaggio preso in considerazione e successivamente le combinazioni di caratteri più frequenti vengono aggiunte iterativamente al vocabolario. In particolare, il modello esegue i seguenti passi, descritti in [28]:

1. Inizializza il vocabolario con tutti i diversi caratteri presenti nella lingua considerata;
2. Crea un modello linguistico basato sui dati di training ottenuti nel punto precedente;
3. Genera delle nuove parole combinando due unità fra quelle già presenti nel vocabolario corrente. Successivamente, aggiunge al vocabolario la

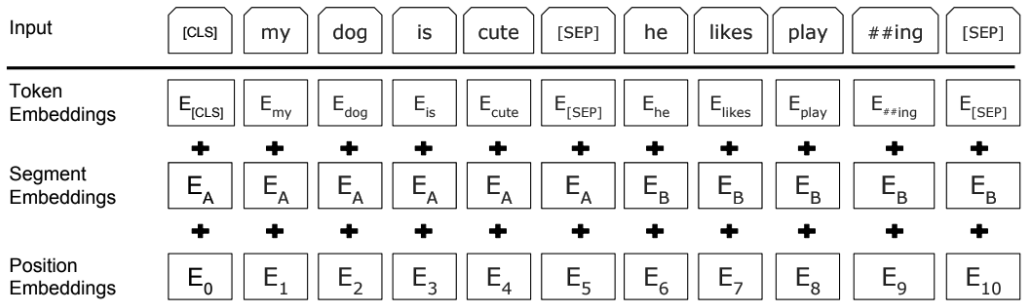


Figura 2.4: L'embedding di input per BERT è formato dalla somma dei token embedding, positional embedding e sequence embedding [1].

parola che massimizza la probabilità dei dati di training. Ciò significa trovare la coppia di token, la cui probabilità divisa per la probabilità del suo primo token seguito dal secondo token è la maggiore tra tutte le coppie di simboli;

4. Ripetere il processo dal punto 2 fino a quando non viene raggiunto un limite predefinito di parole nel vocabolario o l'aumento di probabilità, dovuto all'aggiunta di una nuova parola, scende al di sotto di una certa soglia.

Seguendo questo procedimento, è impossibile che esistano delle parole non presenti nel vocabolario (*Out-Of-Vocabulary*, OOV). Questo è dovuto al fatto che qualsiasi parola non direttamente presente nel vocabolario viene suddivisa in sottoparole più comuni, chiamate *wordpiece*. Dal punto di vista notazionale, ogni sottoparola è facilmente riconoscibile in quanto preceduta dal prefisso ##.

Per quanto riguarda la rappresentazione dell'output, questa è formata dai vettori provenienti dagli strati nascosti finali.

2.2.1 Pre-training

BERT utilizza due task di apprendimento semi-supervisionato per la fase di pre-training: *Masked Language Model* (MLM) e *Next Sentence Prediction*

(NSP).

Il modello è stato addestrato per questi due compiti contemporaneamente in quanto l'obiettivo è quello di ridurre al minimo la funzione di perdita di entrambe le strategie. Il corpus di addestramento è formato da un BookCorpus (800 milioni di parole) e una parte di Wikipedia inglese (2500 milioni di parole).

MLM

La bidirezionalità del modello linguistico di BERT è resa possibile poiché durante il training alcuni token di input vengono mascherati in modo casuale e viene chiesto al modello di prevederli. Ciò che avviene nello specifico è l'applicazione della funzione softmax sulla rappresentazione, derivante dallo strato nascosto finale, del token mascherato, i cui possibili valori di output sono rappresentati dall'intero vocabolario.

La principale conseguenza di questo approccio è la mancata corrispondenza tra i dati di pre-training e quelli di fine-tuning, perché in quest'ultima fase i token che sono stati mascherati non sono disponibili.

Per far fronte a questo problema, le operazioni di mascheramento vengono leggermente modificate e la rappresentazione finale del token mascherato sarà utilizzata per la previsione del token originale tramite la funzione di loss *cross-entropy*.

In particolare, le operazioni di mascheramento seguono queste ipotesi:

- solo il 15% dei token può essere mascherato;
- dopo aver scelto l'i-esimo token, questo può essere:
  - rimpiazzato con il token [MASK], per l'80% delle volte
  - rimpiazzato con un token qualsiasi, per il 10% delle volte
  - lasciato immutato, per il rimanente 10% delle volte

NSP

Questo task è particolarmente importante per tutti quei problemi che si basano sulla comprensione della relazione tra due frasi, come *Question Answering*, che non viene catturata direttamente dal task di MLM. Durante il pre-training il modello riceve una coppia di frasi (A, B) in input e apprende se la seconda frase è quella che segue la prima frase nel testo originale, dove:

- il 50% delle volte la frase B è quella successiva ad A ed è etichettata come *IsNext*;
- il 50% delle volte la frase B è una scelta in modo casuale dal testo originale ed è etichettata come *NotNext*.

La probabilità che la frase sia effettivamente quella successiva o una casuale (*isTheNextSentence*) è prevista attraverso l'uso di un livello di classificazione e una funzione softmax.

Una volta terminata la fase di pre-training, esistono due strategie per applicare le rappresentazioni linguistiche ottenute a degli specifici task: *fine-tuning* e *feature-based* (o *feature extraction*).

2.2.2 Fine-tuning

Utilizzando le potenzialità del *transfer learning*, tramite la fase di fine-tuning, è possibile partire da una qualsiasi delle versioni pre-addestrate di BERT, descritte nella tabella 2.1, e addestrare un nuovo modello per uno specifico compito in un tempo relativamente ridotto e con molti meno dati di training.

Con il termine *transfer learning* ci si riferisce alla tecnica di sfruttare le conoscenze di un modello già addestrato su un certo problema per risolvere un problema differente, ma in qualche modo ad esso correlato.

L'idea generale è di utilizzare un modello di BERT pre-addestrato in cui si aggiunge un ulteriore livello per modificare il suo output e risolvere così uno specifico task.

A seconda del problema che si sta affrontando è fondamentale scegliere il tipo corretto dell'ultimo livello che si va ad aggiungere. Alcuni dei compiti che possono essere risolti tramite il fine-tuning di BERT rientrano nella categoria dei task a livello di token (o *token-level task*). In questa tipologia di problemi, l'obiettivo è assegnare ad ogni token l'etichetta corretta fra quelle disponibili. Alcuni esempi sono il PoS tagging o *Named Entity Recognition* (NER).

2.2.3 Feature extraction

Come visto nella sezione precedente, tramite il fine-tuning è possibile creare un nuovo modello specializzato per una determinata attività. Tuttavia, è anche possibile utilizzare BERT per estrarre le feature da un dataset, così da poterle sfruttare all'interno di un modello completamente diverso.

Questo approccio ha diversi vantaggi. In primis, non tutte le tipologie di problemi possono essere facilmente rappresentate e risolte dall'architettura dell'Encoder dei Transformers, ma a volte è necessario utilizzare un modello diverso. Inoltre, ci sono notevoli vantaggi dal punto di vista computazionale, in quanto la rappresentazione delle feature dei dati di training può essere calcolata una sola volta, ma essere utilizzata per più modelli.

Questa tecnica è facilmente attuabile grazie all'architettura di BERT. Infatti, quest'ultima è composta da una pila di blocchi di Encoder, ognuno dei quali prende, per ogni parola, un input della dimensione del suo word embedding e crea un output della medesima dimensione. Così facendo l'output di ogni encoder può essere utilizzato come input per l'encoder successivo e ognuno può fornire una possibile rappresentazione delle feature, chiamata *contextual embedding* (o *contextual feature*), per una determinata parola.

Pertanto, l'output di ogni livello di BERT può essere utilizzato per ottenere una rappresentazione delle feature. Da quanto emerge nei risultati riportati nell'articolo [1] sullo sviluppo di BERT, la concatenazione degli ultimi quattro layer dell'Encoder fornisce una rappresentazione molto accurata dell'embedding nel caso di NER, ma questi risultati possono essere estesi an-

che a molti altri task. Infatti, questo è quello che viene fatto quando si utilizza BERT per risolvere problemi reali.

2.3 RoBERTa

L'importanza ottenuta da BERT e i risultati che ha permesso di raggiungere, in particolare nel campo dell'NLP, negli anni successivi al suo rilascio, ha attirato l'interesse di molti team di ricercatori. Fra questi, si sono distinti quelli di *Facebook AI* e dell'*Università di Washington* che circa a metà del 2019 hanno rilasciato *RoBERTa* [29] (*Robustly optimized BERT approach*). Il loro intento è stato quello di dimostrare che BERT, così com'è stato introdotto in [1], fosse un ottimo modello, ma poco allenato e che fosse particolarmente sensibile alle modifiche fatte agli iperparametri. Per dimostrarlo hanno fatto diversi esperimenti partendo dalla configurazione base di BERT e modificando alcuni iperparametri, la strategia di addestramento e il dataset utilizzato per il pre-training.

Per quanto concerne le strategie di addestramento di RoBERTa, quattro sono stati i principali cambiamenti rispetto alla strategia utilizzata da BERT:

- *Mascheramento statico vs. Mascheramento dinamico*: nella versione originale di BERT, il mascheramento delle sequenze di input viene eseguito in modo statico, ovvero solo una volta durante la fase di preprocessing. Per garantire che BERT incontri le stesse sequenze di input, ma con mascheramenti differenti, le istanze dei dati di training vengono decuplicate prima dell'applicazione delle varie maschere. In RoBERTa, al contrario, viene utilizzata una tecnica di mascheramento dinamica che genera ogni volta uno schema di mascheramento diverso. Pertanto, il modello incontrerà una maggior quantità di schemi di mascheramento differenti tra loro e non sarà più necessaria la decuplicazione dei dati.

- *Formato dell'input e task di NSP*: a differenza di BERT, RoBERTa utilizza esclusivamente frasi complete come input per il modello. Queste

frasi vengono campionate in modo contiguo dai documenti, facendo sì che la lunghezza totale non superi i 512 token. Nel caso in cui vengano superati i limiti di un documento, verrà inserito un token particolare che ne segna la fine. Il task di NSP è stato rimosso in quanto poco influente per le prestazioni del nuovo modello. Pertanto, anche i token speciali [CLS] e [SEP] utilizzati da BERT non sono più necessari.

- *Fase di training con batch più grandi*: BERT è stato addestrato per un milione di step con dei batch di dimensione 256 ciascuno che equivalgono a 125.000 step con dei batch di dimensione pari a 2000 ognuno. Gli autori in [29] hanno messo in evidenza come l'aumento di dimensione dei batch migliori, in generale, le prestazioni del modello e richieda un minor tempo per il training. Nonostante questo, gli stessi autori sostengono anche che probabilmente non si è ancora riusciti a trovare la dimensione ideale per i batch da utilizzare per massimizzare le prestazioni del modello.

- *Codifica del testo*: per BERT è stata utilizzata la codifica BPE a livello di singolo carattere. Mentre RoBERTa utilizza una variazione di questa codifica introdotta in [30], che è basata sui byte. Il vantaggio di questa variazione è che sono necessari meno wordpiece per codificare un vocabolario più ampio e non è più necessario tokenizzare l'input nella fase di pre-processing dei dati.

Una volta applicate le migliorie alla tecnica di training di BERT, gli autori di RoBERTa hanno addestrato il nuovo modello, utilizzando come dimensioni per l'architettura quella di $\text{BERT}_{\text{LARGE}}$, sul dataset originale formato dal BookCorpus unitamente a una parte di Wikipedia inglese, per un totale di 100.000 step. Questo ha portato ad un primo miglioramento delle prestazioni, riconfermando la bontà delle scelte fatte per le modifiche della fase di training.

Successivamente, hanno eseguito altri tre training con, rispettivamente, 100.000, 300.000 e 500.000 step, combinando il dataset originale ad altri

Modello	Dati (GB)	Batch Size (K)	Step (K)	SQuAD (v1.1/2.0)	MNLI-m	SST-2
RoBERTa						
con Books + Wiki	16	8	100	93.6/87.3	89.0	95.3
+ dati aggiuntivi	160	8	100	94.0/87.7	89.3	95.6
+ pretrain con più step	160	8	300	94.4/88.7	90.0	96.1
+ pretrain con ancora più step	160	8	500	94.6/89.4	90.2	96.4

Tabella 2.2: Risultati ottenuti da RoBERTa al variare delle dimensioni dei dati (16GB e 160GB) di pre-training e della quantità di step (da 100.000 a 500.000). L'architettura di RoBERTa rispecchia esattamente quella di $\text{BERT}_{\text{LARGE}}$.

tre corpus in lingua inglese, diversi fra loro per domini di appartenenza e dimensioni, per un totale di 160GB di dati. Nello specifico, i tre ulteriori corpus utilizzati sono:

- CC-News: contenente 63 milioni di articoli di notizie in inglese, raccolti nel periodo da settembre 2016 a febbraio 2019, per un totale di 76GB di dati;
- OpenWebText: una ricreazione open source del corpus WebText. Il testo è formato da contenuti web estratti da URL condivisi su Reddit che avessero almeno tre voti positivi, per un totale di 38GB di dati;
- Stories: un insieme di dati formato da un sottoinsieme filtrato dei dati di *Common Crawl*, in modo che lo stile fosse simile al *Winograd Schema*, per un totale di 31GB di dati.

Al termine di tutti i training si sono osservati ulteriori miglioramenti nella prestazione del modello, come riportato in tabella 2.2, rispetto ai task presi in considerazione.

Inizialmente RoBERTa è stato sviluppato solo per la lingua inglese. In seguito, a metà 2020 in [31], è stato proposto XLM-RoBERTa: un modello multilingua, la cui implementazione è uguale a quella di RoBERTa, addestra-

to su 2.5TB di dati filtrati, provenienti da Common Crawl. Anche in questo caso XLM-RoBERTa è riuscito a superare le prestazioni ottenute dalle versioni multilingua di BERT.

2.4 Modelli derivati da RoBERTa

Visti i miglioramenti apportati da RoBERTa, altri modelli sono stati creati ereditandone l'architettura. Nelle sezioni successive ne verranno descritti due: *CamemBERT* e *UmBERTo*.

2.4.1 CamemBERT

CamemBERT [32], rilasciato a fine 2019, è un modello linguistico basato su RoBERTa e rappresenta lo stato dell'arte come modello linguistico per il francese. CamemBERT si differenzia da RoBERTa principalmente per l'aggiunta del mascheramento di intere parole (*Whole Word Masking*) e per l'uso di un diverso modello di tokenizzatore, chiamato *SentencePiece Model* (SPM) [33].

In particolare, questa tecnica di mascheramento applica la maschera a un'intera parola, se almeno uno fra tutti i token creati dal tokenizzatore per quella parola è stato originariamente scelto come maschera. Il pre-training viene fatto tramite il task MLM seguendo l'approccio di mascheramento dinamico di RoBERTa, mantenendo immutate le varie percentuali.

La tokenizzazione fatta con SentencePiece è concettualmente simile a BPE, di cui è un'estensione, ma con alcune differenze. In particolare, con questa tecnica non è necessario che le sequenze di input siano state pre-tokenizzate, in quanto la sua implementazione è sufficientemente veloce da permettere di addestrare il modello direttamente su frasi "grezze". Questa caratteristica è particolarmente utile per tutte quelle lingue in cui non sono presenti degli spazi bianchi a dividere le parole. Inoltre, questo tokenizzatore tratta il testo di input come una sequenza di caratteri Unicode, compreso anche lo spazio bianco che viene gestito come un normale token. Quest'ulti-

mo aspetto è molto utile in quanto permette di non avere ambiguità durante la fase di detokenizzazione.

CamemBERT usa le configurazioni originali di BERT, sia per quanto riguarda la versione *BASE* che quella *LARGE*. Il corpus utilizzato per il training è quello francese contenuto in OSCAR (*Open Super-large Crawled ALMAnaCH coRpus*) [34]: un insieme di corpora monolingua estratti da Common Crawl. In particolare, è stata utilizzata la versione unshuffled del corpus francese, contenente 138GB di dati.

CamemBERT è stato valutato in quattro diversi task per il francese: PoS tagging, dependency parsing, NER e Natural Language Inference (NLI). I risultati ottenuti apportano un miglioramento, rispetto allo stato dell'arte, ad ognuno di essi. In particolare, i miglioramenti vengono raggiunti utilizzando CamemBERT in entrambi i modi possibili: per il fine-tuning e per estrarre le feature da dare in input ad un modello specifico per un determinato task.

Questi risultati confermano, ancora una volta, l'utilità e la bontà dei modelli basati su Transformers.

Un altro aspetto messo in evidenza in [32] è il miglioramento del modello dovuto dall'utilizzo di sorgenti di dati eterogenee tra loro, per genere e stile, utilizzate durante la fase di training. A questo scopo, sono state addestrate versioni alternative di $\text{CamemBERT}_{\text{BASE}}$ variando i dati di pre-training e il numero di epoche (calcolate sulla base delle dimensioni dei dati utilizzati), ma tenendo fisso il numero di step a 100.000.

Ciò che si evince dai risultati ottenuti è che i modelli allenati con 4GB di dati provenienti da OSCAR o CCNet sono migliori rispetto a quelli ottenuti utilizzando 4GB di dati estrapolati da Wikipedia francese. Questo vale sia per il fine-tuning che per il feature embedding fatto con CamemBERT, per gli stessi quattro task sopracitati.

Oltre a quanto detto finora, un altro risultato interessante è scaturito da questi esperimenti. Ovvero, il modello addestrato solo su 4GB di dati proveniente da OSCAR o CCNet ottiene risultati simili al modello addestrato sull'intero corpora francese di OSCAR che conta 138GB di dati. Ciò significa

che i modelli con un'architettura simile a quella BASE di CamemBERT, e quindi di BERT, quando addestrati su corpus derivati da OSCAR e CCNet, che sono eterogenei in termini di genere e stile, 4GB di testo non compresso sono sufficienti per raggiungere risultati vicini allo stato dell'arte. Questo ha un duplice vantaggio. In primo luogo, il costo computazionale per il training viene ridotto di molto. Inoltre, viene aperta la strada allo sviluppo di modelli architetturalmente simili a $\text{CamemBERT}_{\text{BASE}}$ che possono essere addestrati per una molteplicità di lingue, per le quali è facilmente reperibile un corpus di soli 4GB di dati da fonti come Common Crawl.

2.4.2 UmBERTo

UmBERTo[1] è un modello linguistico basato su RoBERTa e addestrato su un corpus italiano. Nello specifico, UmBERTo presenta la stessa architettura e le stesse fasi di pre-processing utilizzate da CamemBERT.

UmBERTo è stato rilasciato in due diversi modelli:

- *umberto-wikipedia-uncased-v1*: addestrato su un corpus da circa 7GB di dati, estratto da Wikipedia italiana;

- *umbert-commoncrawl-cased-v1*: addestrato su un corpus italiano ottenuto da OSCAR, di circa 69GB di dati.

Entrambi i modelli presentano la stessa architettura di $\text{BERT}_{\text{BASE}}$.

UmBERTo, nella sua versione Cased, ha ottenuto risultati allo stato dell'arte per il task di PoS tagging per l'italiano. Risultati simili sono stati raggiunti anche dalla versione Uncased, nonostante il dataset di training avesse una dimensione nettamente minore.

In tutti gli esperimenti descritti nella sezione 4.3, viene utilizzato umberto-wikipedia-uncased-v1, sia per il fine-tuning che per il feature embedding.

[1] `https://github.com/musixmatchresearch/umberto`

2.5 Huggingface Transformers

I notevoli progressi nel campo dell'NLP avvenuti negli ultimi anni, sono stati possibili, principalmente, grazie sia all'utilizzo di nuove tecniche di pre-training sia allo sviluppo di nuove architetture, prima fra tutte quella dei Transformers. Quest'ultima ha facilitato la creazione di modelli più performanti, che unitamente a nuove strategie di pre-training ha permesso di raggiungere lo stato dell'arte per una vasta gamma di problemi.

Per facilitare ed espandere l'utilizzo dei modelli basati sull'architettura dei Transformers, nel 2019 è stata creata una libreria open source, chiamata *Huggingface Transformers* [35], il cui obiettivo è di mettere a disposizione della comunità scientifica gran parte di questi nuovi modelli. Inoltre, la libreria garantisce una buona interoperabilità fra i due framework più utilizzati per le applicazioni di ML: *TensorFlow* [36] e *PyTorch* [37].

All'interno della sezione 4.3 si è sfruttata questa libreria per l'utilizzo di UmBERTo nei vari esperimenti, insieme al framework PyTorch.

Capitolo 3

Dependency parsing

L'estrazione automatica delle informazioni presenti in un testo scritto in linguaggio naturale, tramite la sua analisi, è uno dei tanti task facenti parte dell'NLP ed è formata da più fasi. Questo processo è particolarmente delicato in quanto il linguaggio naturale è, per sua natura, ambiguo e ricco di complesse caratteristiche.

Fra le varie elaborazioni a cui un testo viene sottoposto si trova l'analisi sintattica, il cui compito è individuare la struttura sintattica relativa alle singole frasi.

Il parsing a dipendenze fa parte dell'analisi sintattica e sfrutta le *grammatiche a dipendenze* per riuscire a catturare le relazioni sintattiche all'interno di una frase. Tramite l'analisi delle dipendenze è possibile riconoscere, ad esempio, il soggetto della frase, il verbo principale, i suoi argomenti e molti altri elementi sintattici.

In questo capitolo verrà inizialmente descritto il problema del parsing a dipendenze e successivamente saranno analizzate alcune delle tecniche che si possono utilizzare per risolvere questo task. Inoltre, verrà fatta una panoramica su alcuni dei modelli e delle architetture, sia neurali che no, utilizzate per lo svolgimento di questo compito.

3.1 Problema del parsing a dipendenze

Come accennato precedentemente, l'obiettivo del *parser* (o *analizzatore sintattico*), ovvero il programma che esegue il parsing, è di individuare la struttura sintattica relativa ad una frase, grammaticalmente corretta, vista come una sequenza di parole.

A questo proposito, esistono diversi modelli teorici, chiamati *grammatiche formali*, che definiscono il concetto di struttura in modo differente. In generale, quando si parla di *grammatiche* ci si riferisce ad un insieme di regole che specificano come un insieme di simboli, facenti parte di un certo linguaggio, possano essere concatenati l'un l'altro per formare una frase.

L'introduzione delle grammatiche formali nell'ambito della linguistica è dovuta a Noam Chomsky, grazie agli studi fatti a partire dagli anni Cinquanta. Successivamente, Chomsky ha proposto il concetto di *phrase-structure grammar* (o *grammatiche a struttura sintagmatica*), ovvero una tipologia di grammatica generativa costituita da un insieme di regole di produzione che permettono di associare ad ogni frase corretta del linguaggio una struttura ad albero, chiamata *albero sintattico* (o *parse tree*), che rappresenta i collegamenti sintattici esistenti fra le varie parole della frase. I collegamenti sono definiti tramite delle *etichette* (o *label*) che rappresentano le diverse categorie sintattiche. Un esempio di albero sintattico generato da un phrase-structure parsing è mostrato in figura 3.1.

L'albero etichettato ottenuto come risultato da un phrase-structure parsing rappresenta la gerarchia tra diversi gruppi di parole, detti *sintagmi*.

Oltre alla phrase-structure grammar, in letteratura esistono altre famiglie di formalismi grammaticali fra cui le *grammatiche a dipendenze* (o *dependency grammar*), che sono particolarmente importanti nei moderni sistemi di NLP. Questo formalismo è dovuto, principalmente, al lavoro di Tesnière [38], che definisce la nozione di dipendenza come segue:

> La frase è un insieme organizzato, i cui elementi costitutivi sono le parole. Ogni parola che appartiene a una frase cessa di essere isolata

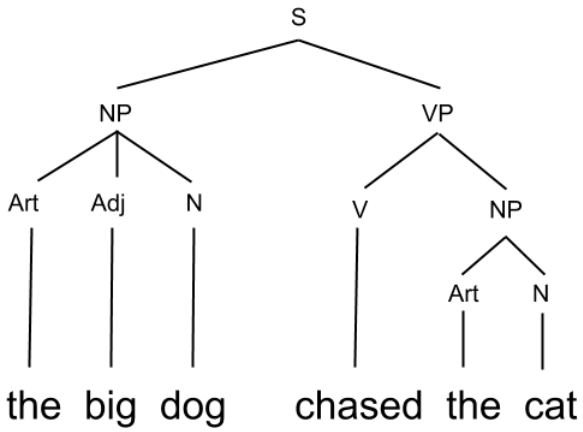


Figura 3.1: Albero sintattico ottenuto tramite phrase-structure parsing per la frase *the big dog chased the cat*. La struttura dei costituenti e le etichette possono anche essere rappresentate tramite un sistema di parentesi etichettate: [NP the big dog] [VP chased [NP the cat]].

> come nel dizionario. Tra la parola e i suoi vicini, la mente percepisce le connessioni, la cui totalità forma la struttura della frase. Le connessioni strutturali stabiliscono relazioni di dipendenza tra le parole. Ogni connessione in linea di principio unisce un termine superiore e un termine inferiore. Il termine superiore è detto *governatore*. Il termine inferiore è detto *subordinato*. [39]

Come si evince dalla definizione, in questi formalismi la struttura sintattica di una frase è descritta esclusivamente in termini di parole che compongono la frase stessa e da un insieme di relazioni grammaticali binarie e asimmetriche presenti tra le parole. Le relazioni vengono chiamate *relazioni di dipendenza* (o *dependency relations*) e sono formate da due elementi: una parola subordinata, detta *dependent* (o *dipendente*), e una parola da cui questa dipende, detta *head* (o *testa*).

Una coppia *testa-dipendente* viene identificata da un arco unidirezionale uscente dalla testa ed entrante nella dipendente. Un esempio di albero sintattico generato da un dependency parsing è mostrato in figura 3.2. Qui sono presenti cinque coppie testa-dipendente e ogni arco presenta un'opportuna

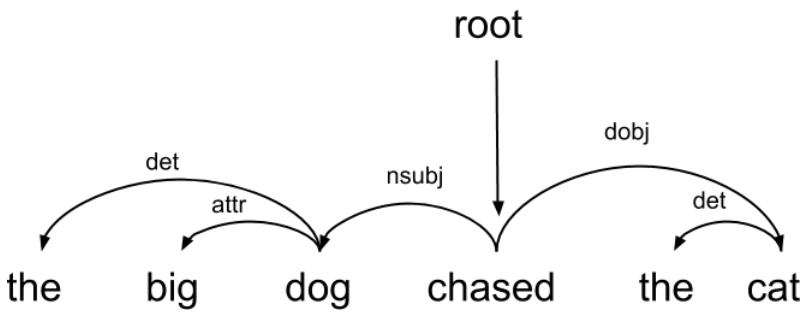


Figura 3.2: Albero sintattico ottenuto tramite dependency parsing per la frase *the big dog chased the cat*. Gli archi unidirezionali partono dalla testa e terminano sulla dipendente. Quando si rappresenta graficamente l'albero si può esplicitare la relazione *root*, considerando la parola $w_0 = root$ oppure non riportarla anche se esiste sempre implicitamente.

etichetta che ne specifica la relazione sintattica.

Dato che nelle relazioni di dipendenza ogni token della frase dipende al più da un altro token della medesima frase, la struttura che può essere utilizzata per rappresentare queste relazioni è un grafo diretto. In particolare, si può considerare il grafo diretto $G = (V, A)$, dove V rappresenta i token all'interno della frase di input $S = w_0, w_1, ..., w_n$, mentre A descrive le relazioni presenti tra gli elementi di V.

Formalmente, come descritto in [40], un grafo delle dipendenze può essere definito come segue:

1. Sia $R = r_1, ..., r_m$ l'insieme dei tipi di etichette consentite per gli archi.

2. Un grafo delle dipendenze per una frase di input $S = w_0, ..., w_n$ è un grafo diretto etichettato $G = (V, A)$, dove:

   (a) V è l'insieme dei nodi, ovvero i token presenti nella frase di input;

   (b) A è l'insieme di archi etichettati (w_i, r, w_j), dove $w_i, w_j \in V, r \in R$.

   Si scrive $w_i < w_j$ per indicare che w_i precede w_j nella frase S; si indica con $w_i \xrightarrow{r} w_j$ un arco da w_i a w_j etichettato con r, mentre con $w_i \rightarrow$

w_j si denota la presenza di un arco indipendentemente dall'etichetta. Inoltre, con $\rightarrow^*$ si indica la chiusura transitiva di un arco.

Un grafo delle dipendenze che soddisfa le seguenti proprietà [41] viene detto *dependency tree* (o *albero delle dipendenze*):

1. (*unique root property*) esiste un solo nodo radice *root* che non ha archi entranti. Formalmente, $\nexists\, w_i \in V$ *tale che* $w_i \rightarrow root$;

2. (*spanning property*) ogni parola della frase S è rappresentata da un nodo nel grafo. Formalmente, $V = \{w_0, w_1, ..., w_n\}$;

3. (*acyclic property*) il grafo non contiene cicli. Formalmente, $\neg\ (w_i \rightarrow w_j \wedge w_j \rightarrow^* w_i)$;

4. (*arc size property*) vale che $|A| = |V| - 1$;

5. (*single head property*) ogni nodo del grafo ha esattamente un arco entrante, ossia ogni parola deve avere una sola testa. Formalmente, $(w_i \rightarrow w_j \wedge w_k \rightarrow w_j) \implies w_i = w_k$;

6. (*connectedness property*) data una coppia di parole $w_i, w_j \in V$ esiste sempre un cammino che le connette, a prescindere dalla direzione della relazione. Formalmente, deve valere che $w_i \rightarrow ... \rightarrow w_j$ oppure $w_j \rightarrow ... \rightarrow w_i$.

Sulla base della tipologia di relazione, un dependency tree può essere classificato in uno dei seguenti tipi: *projective* o *non-projective*.

Una relazione $(w_i, r, w_j) \in A$ è detta *projective* se e solo se la testa associata ad ogni arco ha un percorso nell'albero sintattico che gli permette di raggiungere ogni parola che si trova fra la testa stessa e la dipendente. Questa nozione di proiettività impone un vincolo che è derivato dall'ordine delle parole nella frase di input.

Un *projective dependency tree* è un dependency tree in cui tutte le relazioni $(w_i, r, w_j) \in A$ sono projective.

Al contrario, un *non-projective dependency tree* è un dependency tree in cui almeno una relazione $(w_i, r, w_j) \in A$ non è projective.

Graficamente, un modo per rendersi conto se un dependency tree è projective consiste nell'esaminare gli archi delle relazioni di dipendenza: se tutti gli archi possono essere disegnati senza intersecarsi l'un l'altro, allora il dependency tree è projective.

È bene notare che nonostante la frase analizzata nell'esempio di figura 3.2 sia la medesima utilizzata per il phrase-structure parsing in 3.1, il risultato è nettamente differente. Questo evidenzia il fatto che la scelta dell'approccio da utilizzare condiziona quelle che sono le possibili relazioni sintattiche catturabili dal parser.

Le grammatiche utilizzate per il dependency parsing sono le dependency grammar. Alcuni dei loro vantaggi, rispetto alle phrase-structure grammar, descritti in [39] sono:

- la capacità di riuscire a trattare lingue morfologicamente ricche e con un "word-order" più flessibile e libero, se paragonato all'inglese. Questo deriva dal fatto che le dipendenze fra parole distanti in una frase sono più difficilmente catturabili da un parser a costituenti;
- le relazioni testa-dipendente sono d'aiuto anche nell'analisi semantica, specialmente in alcuni task come il *Question Answering*;
- il processo di parsing è, generalmente, più semplice in quanto ogni nodo corrisponde esattamente ad una parola e il compito del parser è quello di riuscire a connetterle correttamente.

Inoltre, Abney in [42] sostiene che "la mente umana sembra operare il parsing secondo una strategia a dipendenze".

Esistono diversi approcci che possono essere utilizzati per risolvere il problema del dependency parsing. In generale, possiamo dividerli in due categorie:

1. *approccio grammar-based*: si utilizzano grammatiche formali per definire linguaggi formali con l'obiettivo di categorizzare le frasi;

2. *approccio data-driven*: si utilizzano tecniche di apprendimento automatico su dati linguistici per far apprendere il modello di parsing.

Nelle sezioni successive saranno discussi i due metodi più utilizzati nel dependency parsing: *transition-based* (sezione 3.3) e *graph-based* (sezione 3.4). In particolare, entrambe le metodologie verranno trattate secondo un approccio *data-driven* in cui il problema viene diviso in due fasi: la fase di apprendimento, in cui il modello di parsing viene addestrato attraverso l'uso di un training set annotato; la fase di parsing, in cui il modello precedentemente addestrato viene utilizzato per costruire i dependency tree a partire da nuove frasi.

La scelta di concentrarsi maggiormente su un approccio data-driven, deriva dal fatto che nel capitolo 4 verranno descritte e utilizzate tecniche di machine learning per risolvere il compito del parsing a dipendenze per la lingua italiana.

3.2 Universal Dependency

Ciò che è alla base della costruzione di un qualsiasi modello di parsing data-driven è un insieme D di frasi annotate sintatticamente, chiamato *treebank*.

Il treebank è un insieme di N coppie $(S^{(i)}, T^{(i)})$, dove $S^{(i)}$ è una frase e $T^{(i)}$ il corrispondente dependency tree annotato:

$$D = \{(S^{(i)}, T^{(i)})\}_{i=1}^{N} \tag{3.1}$$

In generale, esistono diverse tipologie di treebank che possono essere utilizzate in base al tipo di parsing considerato. Alcuni esempi sono i *phrase-structure treebank* o i *dependency treebank*.

Le modalità di costruzione dei treebank si sono evolute nel corso del tempo. Inizialmente le annotazioni venivano fatte tutte manualmente da parte di esperti linguisti, mentre oggi si utilizza una combinazione più o meno elaborata di lavoro manuale ed elaborazione automatica.

Una delle considerazioni più importanti nell'annotazione di un treebank da parte degli esperti è garantire la coerenza, ovvero assicurare che fenomeni linguistici simili vengano annotati in modo simile all'interno di tutto il corpus. Questo è un requisito fondamentale per gli utilizzatori del treebank poiché permette loro di avere uno standard corretto di riferimento, noto anche come *gold standard*.

Idealmente, la progettazione di un treebank dovrebbe essere motivata dall'uso specifico previsto. Al contempo, dato che lo sviluppo di un treebank è particolarmente laborioso, spesso si cerca di progettarne la costruzione in modo tale che possa servire a più scopi contemporaneamente [43].

La prima scelta da fare durante la progettazione dei treebank è se includere la lingua scritta, quella parlata o entrambe. In generale, come descritto in [44], la lingua parlata è molto meno rappresentata rispetto a quella scritta, soprattutto perché gli approcci per la rappresentazione sintattica si sono concentrati maggiormente su quest'ultima. Tuttavia, ad oggi, esistono anche treebank, soprattutto per l'inglese, che utilizzano dati derivati dalla lingua parlata come il progetto *Christine* [45].

Un altro aspetto da tenere in considerazione durante la progettazione dei treebank è la scelta delle fonti da cui attingere i testi. Infatti, in base a questa scelta si possono trovare dei testi che utilizzano un gergo generico oppure specifico, legato ad un determinato genere o ambito. Ad esempio, un treebank costituito da testi provenienti da sole riviste specialistiche nel campo giuridico rende il modello appreso più adatto a task che riguardano il parsing di frasi con la stessa terminologia giuridica.

In generale, la maggior parte dei treebank, come riportato in [44], si basano sul testo dei quotidiani contemporanei, in quanto hanno la caratteristica di essere facilmente accessibili. Un noto esempio è il Penn Treebank (PTB) [46], formato in larga parte da testi provenienti dal Wall Street Journal.

In particolare, il PTB è il primo treebank ampiamente utilizzato per l'inglese ed è stato preso come modello per lo sviluppo dei treebank in altre lingue. Il PTB contiene circa due milioni di parole ed è basato sulle

grammatiche phrase-structure.

Per quanto riguarda i dependency treebank, lo standard de facto sono le *Universal Dependencies* (UD), progetto avviato e portato avanti da Nivre et al. [47], giunto, recentemente, alla versione 2.7[1].

Le Universal Dependencies sono il risultato di uno sforzo da parte della comunità scientifica, rappresentato da una risorsa condivisa comune e un framework che permetta la standardizzazione del processo di annotazione per più lingue. L'obiettivo è quello di mettere a disposizione di tutti una sempre più ampia collezione di treebank (oltre 180 nella versione 2.7 in più di 100 lingue), per permettere l'apprendimento di modelli di parsing in varie lingue e garantire valutazioni confrontabili nei vari task.

Parte degli schemi di dipendenza delle UD sono stati ereditati da progetti precedenti, fra cui le Stanford Dependencies [48].

Il formato dei file adottato dalle UD è il *CoNLL-U*, un'estensione rivista del formato *CoNLL-X*.

CoNLL (*Computational Natural Language Learning*) è una conferenza che si tiene annualmente in cui vengono proposti diversi *shared task*, ovvero delle competizioni in cui si valutano i diversi modelli realizzati dai partecipanti, a partire da uno stesso treebank. Le competizioni vertono sempre su tematiche legate al mondo dell'NLP.

Il formato CoNLL-X è stato utilizzato durante lo shared task del 2006 [49] per uniformare in uno standard comune i diversi treebank impiegati durante lo svolgimento del task. In questo formato le frasi devono essere memorizzate in chiaro all'interno di un file di testo con codifica UTF-8 e separate tra loro da un carattere di linea vuota. Ogni frase è formata da uno o più token disposti ognuno su una riga. I token sono composti da dieci attributi separati fra loro da un carattere di tabulazione. Nello specifico, come descritto in [49], gli attributi per ogni token sono:

1. *ID*: contatore per i token. Comincia da uno per ogni nuova frase.

[1] `https://universaldependencies.org/`

2. *FORM*: parola della frase o simbolo di punteggiatura.
3. *LEMMA*: lemma o radice della parola. Se non disponibile si utilizza il carattere underscore.
4. *CPOSTAG*: PoS tag generico, detto *coarse-grained*.
5. *POSTAG*: PoS tag specifico, detto *fine-grained*. Se non disponibile, il valore è uguale a quello dell'attributo precedente.
6. *FEATS*: insieme non ordinato delle caratteristiche sintattiche e/o morfologiche. Se non disponibili si utilizza il carattere underscore. Ogni caratteristica è separata dal carattere " | " (barra verticale).
7. *HEAD*: la testa della parola corrente. Può essere o l'attributo ID di un altro token o zero (" 0 ") se il token è la radice della frase.
8. *DEPREL*: etichetta della relazione testa-dipendente. Se HEAD=0, l'etichetta della relazione può essere significativa o avere il valore di default ROOT.
9. *PHEAD*: testa projective del token corrente. Può essere o l'attributo ID di un altro token o zero (" 0 ") se il token è la radice della frase, o underscore se non disponibile. La struttura ottenuta tramite questo attributo garantisce proprietà projective.
10. *PDEPREL*: etichetta della relazione testa-dipendente relativa a PHEAD. Se non disponibile si utilizza il carattere underscore.

Gli attributi CPOSTAG, POSTAG, DEPREL e PDEPREL hanno un insieme di possibili valori strettamente dipendenti dal treebank utilizzato.

Come accennato precedentemente, il formato CoNLL-U è stato definito a partire dal CoNLL-X, ridefinendone alcuni campi.

Nel formato CoNLL-U le frasi devono essere memorizzate in chiaro all'interno di un file di testo con codifica UTF-8, ma a differenza del formato CoNLL-X quest'ultime vengono separate tra loro dal carattere LF, presente

anche alla fine del file. Anche in CoNLL-U ogni token occupa una linea del file di testo ed è formato da dieci attributi, separati tra loro da un carattere di tabulazione. In più, CoNLL-U permette di creare dei commenti e/o metadati all'interno del file di testo, anteponendo il carattere " # " all'inizio della linea. Le righe di commento e metadati non sono, però, consentite all'interno di una frase, ovvero tra le righe corrispondenti ai token. Il contenuto dei commenti e dei metadati è illimitato, ma dalla versione 2.0 delle UD sono obbligatori due commenti per ogni frase: un ID univoco a livello dello specifico treebank da cui proviene una determinata frase (*sent_id*) e una stringa che rappresenta l'intera frase non annotata (*text*).

I dieci attributi per i token nel formato CoNLL-U sono:

1. *ID*: contatore per i token. Comincia da uno per ogni nuova frase. Può essere definito da un intervallo per rappresentare i token composti da più parole o può essere un numero decimale compreso fra uno e zero per i nodi vuoti.

2. *FORM*: parola della frase o simbolo di punteggiatura.

3. *LEMMA*: lemma o radice della parola.

4. *UPOS*: PoS tag generico, definito negli *universal part-of-speech tag* delle linee guida per le UD versione 2. I tag sono riportati in tabella 3.1.

5. *XPOS*: PoS tag specifico per la lingua utilizzata. Se non disponibile si utilizza il carattere underscore.

6. *FEATS*: lista non ordinata delle caratteristiche morfologiche ottenute dalle *universal feature inventory* o definite da un'estensione nelle *language-specific extension*. Se non disponibili si utilizza il carattere underscore. L'elenco delle caratteristiche è riportato in tabella 3.2.

7. *HEAD*: la testa della parola corrente. Può essere o l'attributo ID di un altro token o zero (" 0 ").

8. *DEPREL*: etichetta della relazione testa-dipendente ottenute dalle *universal dependency relation* o da un loro sottotipo specifico per una certa lingua. L'etichetta è ROOT se e solo se HEAD=0. Le relazioni sono riportate in tabella 3.3.

9. *DEPS*: dependency graph migliorato definito sotto forma di un elenco di coppie HEAD-DEPREL.

10. *MISC*: qualsiasi altra annotazione.

I campi DEPS e MISC del formato CoNLL-U sostituiscono i campi obsoleti PHEAD e PDEPREL del formato CoNLL-X. Inoltre, è stato modificato l'uso dei campi ID, FORM, LEMMA, XPOSTAG, FEATS e HEAD che nel formato CoNLL-U devono rispettare i seguenti vincoli:

- i campi non devono essere vuoti;
- i campi diversi da FORM e LEMMA non possono contenere il carattere spazio;
- il carattere underscore è usato per specificare i valori non presenti in tutti gli attributi ad eccezione di ID. Inoltre, nei treebank UD gli attributi UPOS, HEAD e DEPREL non possono essere lasciati senza specifica.

Una delle caratteristiche chiave del formato CoNLL-U è l'obbligo di utilizzare un insieme di valori prestabiliti nelle linee guida delle UD per gli attributi UPOS, FEATS e DEPREL. Così facendo i valori assegnabili a questi attributi non sono più dipendenti dalla lingua utilizzata e dal treebank originale, come accadeva per il formato CoNLL-X.

Per quanto riguarda la codifica di un dependency tree, gli attributi strettamente necessari sono ID, FORM, HEAD e DEPREL. Nonostante questo, l'utilizzo di altri attributi per rappresentare la categoria lessicale della parola e le sue caratteristiche morfosintattiche, possono essere utili durante la fase di apprendimento del parser.

Parole open class	Parole closed class	Altro
ADJ	ADP	PUNCT
ADV	AUX	SYM
INTJ	CCONJ	X
NOUN	DET	
PROPN	NUM	
VERB	PART	
	PRON	
	SCONJ	

Tabella 3.1: Elenco delle etichette per i PoS tag definite nelle UD versione 2.

Caratteristiche lessicali	Caratteristiche di inflessione	
	Nominali	*Verbali*
PronType	Gender	VerbForm
NumType	Animacy	Mood
Poss	NounClass	Tense
Reflex	Number	Aspect
Foreign	Case	Voice
Abbr	Definite	Evident
Typo	Degree	Polarity
		Person
		Polite
		Clusivity

Tabella 3.2: Elenco delle caratteristiche morfologiche definite nelle UD versione 2.

	Nominals	**Clauses**	**Modifier words**	**Function words**
Core arguments	nsubj obj iobj	csubj ccomp xcomp		
Non-core dependents	obl vocative expl dislocated	advcl	advmod discourse	aux cop mark
Nominal dependents	nmod appos nummod	acl	amod	det clf case
Coordination	**MWE**	**Loose**	**Special**	**Other**
conj cc	fixed flat compound	list parataxis	orphan goeswith reparandum	punct root dep

Tabella 3.3: In tabella sono elencati i 37 tipi di relazioni sintattiche universali utilizzate nelle UD versione 2. Nella parte superiore della tabella, le righe si riferiscono a categorie funzionali in relazione alla testa, mentre le colonne corrispondono a categorie strutturali della dipendente. Nella parte inferiore della tabella sono evidenziate le relazioni che non sono relazioni di dipendenza in senso stretto.

```
# sent_id = isst_tanl-2083
# text = È nella natura delle cose e degli uomini.
1 È essere AUX V Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin 4 cop 4:cop _
2-3 nella _ _ _ _ _ _ _ _
2 in in ADP E _ 4 case 4:case _
3 la il DET RD Definite=Def|Gender=Fem|Number=Sing|PronType=Art 4 det 4:det _
4 natura natura NOUN S Gender=Fem|Number=Sing 0 root 0:root _
5-6 delle _ _ _ _ _ _ _ _
5 di di ADP E _ 7 case 7:case _
6 le il DET RD Definite=Def|Gender=Fem|Number=Plur|PronType=Art 7 det 7:det _
7 cose cosa NOUN S Gender=Fem|Number=Plur 4 nmod 4:nmod:di _
8 e e CCONJ CC _ 11 cc 11:cc _
9-10 degli _ _ _ _ _ _ _ _
9 di di ADP E _ 11 case 11:case _
10 gli il DET RD Definite=Def|Gender=Masc|Number=Plur|PronType=Art 11 det 11:det _
11 uomini uomo NOUN S Gender=Masc|Number=Plur 7 conj 4:nmod:di|7:conj:e SpaceAfter=No
12 . . PUNCT FS _ 4 punct 4:punct _
```

Figura 3.3: Frase annotata nel formato CoNLL-U presente nel test set della UD Italian ISDT versione 2.6. Si possono notare alcune peculiarità del formato CoNLL-U, come le parole “nella”, “delle” e “degli” che vengono riportate utilizzando l’intervallo degli ID delle due parole da cui sono composte e la parola “uomini” che presenta l’attributo MISC il cui valore *SpaceAfter=No* indica che nel testo originale questo token non era seguito da un carattere di spazio. Inoltre, nelle prime due righe sono presenti due metadati sotto forma di commento: il primo indica il treebank di provenienza della frase, in questo caso ISST-TANL; il secondo riporta il testo completo della frase.

Infine, è sempre possibile convertire un file dal formato CoNLL-U al formato CoNLL-X, nonostante si possano perdere alcune informazioni riguardanti gli ultimi due attributi di CoNLL-U in quanto completamente ridefiniti rispetto a quelli presenti in CoNLL-X.

Una descrizione più dettagliata della UD utilizzata in questo lavoro di tesi è presente nella sezione 4.2, mentre in figura 3.3 è riportato un esempio di frase annotata nel formato CoNLL-U, in cui sono evidenziate alcune delle differenze con il formato CoNLL-X.

3.3 Transition-based dependency parsing

Il transition-based dependency parsing, chiamato anche *shift-reduce dependency parsing*, è un approccio *stack-based*, ispirato dal parsing deterministico *shift-reduce* per le grammatiche libere da contesto. Concettualmente, si utilizza un modello parametrizzato sulle transizioni di una macchina astratta per derivare l'albero delle dipendenze di una data frase.

I parser transition-based sono formati da due componenti principali: un *sistema di transizioni*, che rappresenta il modello di parsing che deve essere introdotto durante la fase di apprendimento; un *algoritmo di parsing* che utilizza il sistema di transizioni per costruire l'albero delle dipendenze, data una frase.

In particolare, come descritto in [39], dato un insieme R di tipi di dipendenza e un insieme $V = \{w_0, w_1, ..., w_n\}$ di parole, una *configurazione* per una frase $S = w_0, w_1, ..., w_n$ è definita da una tripla $c = (\sigma, \beta, A) \in C$, dove:

- σ è lo *stack* delle parole $w_i \in V$;
- β è il *buffer* delle parole $w_i \in V$;
- A è l'insieme delle relazioni di dipendenza $(w_i, r, w_j) \in V \times R \times V$.

L'idea alla base di una configurazione è che quest'ultima rappresenti un'analisi parziale della frase di input, in cui le parole presenti sullo stack σ sono state analizzate dal parser, le parole nel buffer β devono ancora essere analizzate e A rappresenta un dependency tree parzialmente costruito.

Pertanto, un *sistema di transizioni* (o *transition system*) [50] per un transition-based dependency parser può essere definito come una quadrupla $TS = (C, T, c_s, C_t)$, dove:

- C è l'insieme di configurazioni;
- T è l'insieme di transizioni, ognuna delle quali è una funzione parziale $t : C \to C$ che associa ad una data configurazione $c \in C$, una nuova configurazione valida $c' \in C$;

- c_s è la *funzione di inizializzazione*, che associa una frase $S = w_0, w_1, ..., w_n$ ad una configurazione con $\beta = [1, ..., n]$;
- $C_t \subseteq C$ è l'insieme di configurazioni terminali.

Per ogni frase $S = w_0, w_1, ..., w_n$, la funzione di inizializzazione fornisce la *configurazione iniziale*:

$$c_0 = ([w_0]_\sigma, [w_1, ..., w_n]_\beta, \emptyset) \tag{3.2}$$

Mentre la configurazione finale C_t ha la forma:

$$(\sigma, [\,]_\beta, A) \; \forall \; \sigma, A \tag{3.3}$$

Infine, una *computazione* (o *sequenza di transizioni*) per una frase S viene definita come una sequenza di $m \geq 0$ configurazioni $c_0, ..., c_m$, dove:

$$c_i = t_i(c_{i-1}) \forall i \; con \; 1 < i \leq m \tag{3.4}$$

per una qualche transizione valida $t_i \in T$.

La computazione è detta *completa* quando c_0 corrisponde alla configurazione iniziale e $c_m = (\sigma_m, [\,]_\beta, A_m) \in C_t$, dove A_m è il dependency tree ben formato per la frase S.

Come descritto dall'equazione 3.4, ogni configurazione, ad eccezione di quella iniziale, è il risultato di una transizione applicata alla configurazione precedente. Ciò significa che durante la *fase di parsing* un algoritmo di parsing deterministico, per ogni configurazione della sequenza, eccetto quelle terminali, deve scegliere quale funzione di transizione utilizzare per riuscire a costruire l'albero delle dipendenze. Questa scelta viene fatta dalla funzione *oracle*, che fornisce sempre la transizione ottimale per ogni configurazione.

3.3.1 Fase di parsing

In [50] una funzione oracle per un sistema di transizioni $TS = (C, T, c_s, C_t)$ è definita come $o : C \rightarrow T$.

Quindi, dato un sistema di transizioni TS e un oracolo o, è possibile ottenere un algoritmo *greedy* per il parsing deterministico di un generico transition-based dependency parser:

```
function PARSE(S = (w_0, w_1, ..., w_n))
    c ← ([w_0]_σ, [w_1, ..., w_n]_β, ∅)
    while c ≠ (σ, [ ]_β, A)
        t ← o(c)
        c ← t(c)
    return c
```

Figura 3.4: Algoritmo greedy per il parsing deterministico transition-based.

L'algoritmo restituisce la configurazione finale contenente il dependency tree per la frase S di input.

La complessità temporale di questo algoritmo di parsing è $O(n)$, dove n rappresenta il numero di parole nella frase S, supponendo che la funzione oracle $o(c)$ e la transizione $t(c)$ vengano calcolate in tempo costante. Mentre la complessità spaziale è data da $O(|c|)$, per una generica configurazione $c \in C$, in quanto solo una configurazione alla volta deve essere memorizzata in un dato momento.

L'efficienza computazionale di questo approccio deriva dal fatto che viene eseguita una singola analisi della frase, prendendo le decisioni in modo greedy, senza considerare eventuali alternative. Inoltre, una volta presa una decisione non è più possibile tornare indietro.

Un metodo alternativo consiste nell'esplorare diverse sequenze di transizioni ad ogni step, selezionando la migliore fra le varie possibilità. Come proposto in [51], si potrebbe utilizzare una *beam search* che combina una ricerca in ampiezza con un'euristica che permette di limitare il, potenzialmente enorme, numero di sequenze. In particolare, invece di scegliere la miglior transizione ad ogni passo, si applicano tutte le transizioni valide ad ogni stato contenuto in un'*agenda*, che inizialmente contiene solo la configu-

razione iniziale. Le nuove configurazioni ottenute dall'applicazione di tutte le transizioni valide si aggiungono all'agenda che ha, però, una dimensione limitata, detta *beam width*. Una volta che l'agenda raggiunge il limite massimo, verranno aggiunte solo le configurazioni migliori rispetto a quelle già presenti, mentre le altre verranno rimosse, iniziando dalle peggiori. Questa ricerca continuerà fintantoché all'interno dell'agenda compare almeno uno stato non finale.

Utilizzando l'approccio con beam search è necessario definire una nozione di punteggio, che può essere calcolato tramite una funzione *score*. Questa funzione ha il compito di assegnare ad ogni configurazione un punteggio che ne valuti la bontà, permettendo così di discriminare le migliori dalle peggiori.

Formalmente il problema di parsing consiste nel selezionare la miglior transizione $\bar{t}$ che massimizzi lo score:

$$\bar{t} = \underset{t \in T}{arg\ max}\ \ score(c, t) \tag{3.5}$$

3.3.2 Fase di learning

Nella pratica, è impossibile avere un oracolo come descritto nella sezione precedente, ma quello che si cerca di fare è di trovarne la miglior approssimazione possibile. Infatti, la definizione e costruzione di una corretta e ottimale funzione oracle è proprio il problema centrale nell'analisi delle dipendenze per un transition-based dependency parser e viene affrontata durante la *fase di learning*.

Esistono diversi metodi per approssimare la funziona oracle, come l'utilizzo di grammatiche formali ed euristiche di disambiguazione. Tuttavia, la strategia che viene maggiormente utilizzata dai parser allo stato dell'arte è fondata sul machine learning supervisionato, che permette di approssimare un oracolo tramite un classificatore, addestrato a partire da un treebank.

Idealmente, l'oracolo lo si può immaginare come una enorme tabella contenente la transizione giusta per ogni possibile configurazione $c \in C$. Pertanto, ad ogni step della computazione è sufficiente scegliere la transizione

valida associata ad una particolare configurazione, semplicemente leggendone la riga corrispondente nella tabella.

Tuttavia, per far sì che questo problema risulti trattabile dagli algoritmi di machine learning supervisionato, è necessario introdurre un'astrazione sull'ipoteticamente infinito insieme di possibili configurazioni. Questo viene fatto attraverso una funzione $f(c)$ che associa ad ogni configurazione un vettore di feature.

Nello specifico, la *rappresentazione in feature* di una data configurazione $c \in C$ è una funzione $f(c) : C \rightarrow F^m$ che associa la configurazione c ad un vettore m-dimensionale composto da singole feature $f_i(c)$ per $1 \leq i \leq m$:

$$f(c) = \langle f_1(c), f_2(c), ..., f_m(c) \rangle \tag{3.6}$$

Generalmente, il vettore di feature per una certa configurazione è definito da attributi arbitrari, categorici o numerici, contenuti nel treebank. Ad esempio, il PoS tag relativo ad una parola presente nello stack σ o nel buffer β è una feature categorica, il cui valore viene preso da un insieme di possibili tag.

Formalmente, come definito in [39], data una configurazione c, una *feature* $f_i(c)$ è formata da:

- una *address function*, ovvero una funzione che accede ad una particolare posizione dello stack σ o del buffer β e restituisce la parola $w_i \in V$ in quella posizione;
- una *attribute function* che data una parola $w_i \in V$ accede ad un determinato attributo presente nel treebank e ne restituisce il valore. Tipicamente queste tipologie di funzioni si riferiscono a proprietà linguistiche delle parole che possono essere date in input al parser o calcolate durante il processo di analisi.

Una caratteristica delle address function è che queste possono essere a loro volta composte da funzioni più semplici, che operano su diverse componenti di una stessa configurazione. Pertanto, è possibile avere una address

function composta da due funzioni, in cui, ad esempio, la prima estrae l'i-esima parola w_i dalla cima dello stack σ; mentre la seconda mappa w_i con un'altra parola tra quelle raggiungibili, utilizzando specifiche funzioni per visitare l'albero delle relazioni di dipendenza parzialmente predette. Alcuni esempi di funzioni impiegate per la visita di un albero sono *leftmost/rightmost child*, *leftmost/rightmost sibling* ecc. Inoltre, è anche possibile concatenare diversi attributi per una stessa parola.

Le feature estratte dalle attribute function possono essere suddivise in due categorie:

- feature *statiche*, il cui valore è costante in ogni configurazione per una data frase, in quanto annotato nel treebank di riferimento;
- feature *dinamiche*, la cui disponibilità varia in base alle relazioni di dipendenza parzialmente predette durante il parsing.

Combinando diverse feature f_i, è possibile costruire un *feature template* f che consente di generare automaticamente le feature specificate all'interno del template, dato un training set e una configurazione c.

È bene notare che le feature utilizzate per addestrare i sistemi transition-based possono variare in base a diversi fattori e queste vengono stabilite prima della fase di apprendimento. Successivamente all'avvento del deep learning si è eliminata la complessità dovuta alla corretta selezione delle feature e sono state migliorate le performance dei parser transition-based, come evidenziato in [52].

Una volta ottenuta la rappresentazione delle feature, ciò che si vuole è apprendere una funzione che determini la corretta transizione t_i, per ogni configurazione c_i, data in input la rappresentazione in feature $f(c_i)$. Nel campo del machine learning questo può essere trattato come un problema di classificazione, in cui le istanze da classificare sono le rappresentazioni in feature delle configurazioni, mentre le possibili classi da predire sono le transizioni, come definite dal sistema di transizione. Quindi, un siffatto clas-

sificatore è ciò che può essere utilizzato come approssimazione della funzione oracle.

All'interno di uno scenario di apprendimento supervisionato, un'istanza di training per un classificatore è nella forma $(f(c), t)$, dove $t = o(c)$. Tuttavia, i treebank sono nella forma

$$D = \{(S^{(j)}, T^{(j)})\}_{j=1}^{N} \tag{3.7}$$

in cui ad ogni frase corrisponde il suo albero delle dipendenze. Quindi, per riuscire ad addestrare correttamente un classificatore è fondamentale convertire il treebank in un training set formato da configurazioni associate alla loro transizione corretta:

$$D^{(train)} = \{(f(c_j), t_j)\} \tag{3.8}$$

dove $t_j = o(c_j)$.

Per eseguire la conversione da D a $D^{(train)}$ è possibile utilizzare il seguente schema algoritmico, come riportato in [39]:

1. per ogni istanza $(S^{(j)}, T^{(j)}) \in D$, si costruisce la corrispettiva sequenza di transizioni $C^j_{0,m} = (c_0, c_1, ..., c_m)$ tale che:

   (a) $c_0 = c_0(S^{(j)}) = ([w_0]_\sigma, [w_1, ..., w_n]_\beta, \emptyset)$

   (b) $T^{(j)} = (V^{(j)}, A_m)$

2. per ogni configurazione non terminale $c^j_i \in C^j_{0,m}$, si aggiunge a $D^{(train)}$ l'istanza formata da $(f(c^j_i), t^j_i)$, dove $c^j_i = t^j_i(c^j_{i-1})$.

Il presupposto alla base di questo schema è che per ogni frase $S^{(j)}$ che ha come albero delle dipendenze $T^{(j)}$, sia possibile costruire una sequenza di transizioni che abbia nella sua configurazione finale proprio $T^{(j)}$.

Infine, nella fase di learning si cerca di approssimare l'oracolo tramite un classificatore lineare:

$$o(c) = \underset{t \in T}{arg\,max}\ \ w \cdot score(f(c,t)) \tag{3.9}$$

dove w sono i pesi appresi durante il training fatto sul treebank.

3.3.3 Sistemi di transizione

Ciò che caratterizza e differenzia i parser transition-based l'uno dall'altro è il tipo di sistema di transizione adottato. Quest'ultimo definisce in modo chiaro quali modifiche vengono apportate da una transizione rispetto agli elementi di una configurazione.

I sistemi più noti e utilizzati in letteratura sono tre:

1. *Arc-standard* [53] (accennato inizialmente in [54])
2. *Arc-eager* [41]
3. *Arc-hybrid* [55]

Successivamente, si utilizzerà la notazione $\sigma|w_i$ per indicare lo stack σ che ha in cima la parola w_i e $w_i|\beta$ per rappresentare il buffer β che ha come prima parola w_i.

Nel sistema arc-standard, formalmente definito in tabella 3.4, sono presenti tre tipi di possibili transizioni:

1. LEFT-ARC: aggiunge una relazione di dipendenza (w_j, r, w_i), per una qualsiasi relazione $r \in R$, all'insieme delle relazioni A, dove w_j è la prima parola sullo stack e w_i la seconda parola sullo stack. Inoltre, rimuove dallo stack la parola w_i. Questa transizione può essere applicata se $w_i \neq root$.
2. RIGHT-ARC: aggiunge una relazione di dipendenza (w_i, r, w_j), per una qualsiasi relazione $r \in R$, all'insieme delle relazioni A, dove w_j è la prima parola sullo stack e w_i la seconda parola sullo stack. Inoltre, rimuove dalla cima dello stack la parola w_j.

Transizioni	
LEFT-ARC	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_j, \beta, A \cup (w_j, r, w_i))$
RIGHT-ARC	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_i, \beta, A \cup (w_i, r, w_j))$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$

Tabella 3.4: Sistema di transizioni Arc-standard.

3. SHIFT: rimuove la prima parola w_i nel buffer e la aggiunge in cima allo stack.

Le precondizioni per questo sistema richiedono che durante l'utilizzo delle transizioni LEFT-ARC e RIGHT-ARC ci siano almeno due elementi nello stack σ e non è possibile utilizzare LEFT-ARC quando *root* è il secondo elemento dello stack, altrimenti verrebbe meno la condizione generale per cui *root* non può avere archi entranti (unique root property). Inoltre, questo sistema adotta una strategia puramente bottom-up per la costruzione degli archi.

Data la natura delle transizioni, gli alberi delle dipendenze generati da questo sistema possono essere solo projective.

Il sistema arc-eager rappresenta una valida alternativa all'arc-standard. In particolare, l'arc-eager differisce dall'arc-standard in quanto le relazioni di dipendenza a sinistra sono catturate in maniera bottom-up, mentre quelle a destra utilizzando un approccio top-down. Questo garantisce che gli archi vengano aggiunti al dependency graph immediatamente, non appena le coppie testa-dipendente sono disponibili.

Nel sistema arc-eager, formalmente definito in tabella 3.5, sono presenti quattro tipi di possibili transizioni:

1. LEFT-ARC: aggiunge una relazione di dipendenza (w_j, r, w_i), per una qualsiasi relazione $r \in R$, all'insieme delle relazioni A, dove w_j è la prima parola nel buffer e w_i è la parola in cima allo stack. Inoltre, rimuove dallo stack la parola w_i. Questa transizione può essere applicata se $w_i \neq root$.

Transizioni	
LEFT-ARC	$(\sigma\|w_i, w_j\|\beta, A \Rightarrow (\sigma, w_j\|\beta, A \cup (w_j, r, w_i)$
RIGHT-ARC	$(\sigma\|w_i, w_j\|\beta, A \Rightarrow (\sigma\|w_i\|w_j, \beta, A \cup (w_i, r, w_j)$
REDUCE	$(\sigma\|w_i, \beta, A) \Rightarrow (\sigma, \beta, A)$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$

Tabella 3.5: Sistema di transizioni Arc-eager.

2. RIGHT-ARC: aggiunge una relazione di dipendenza (w_i, r, w_j), per una qualsiasi relazione $r \in R$, all'insieme delle relazioni A, dove w_j è la prima parola nel buffer e w_i è la parola in cima allo stack. Inoltre, sposta w_j in cima allo stack.

3. REDUCE: rimuove la prima parola w_i in cima allo stack.

4. SHIFT: rimuove la prima parola w_i nel buffer e la aggiunge in cima allo stack.

Le precondizioni $(w_k, r', w_i) \notin A$ e $(w_j, r, w_i) \in A$ per le transizioni LEFT-ARC e REDUCE, rispettivamente, verificano l'esistenza di un arco che abbia come dipendente l'elemento in cima allo stack. Questo è fondamentale per LEFT-ARC in quanto evita che una parola abbia due teste diverse, mentre per la REDUCE evita che una parola senza testa venga rimossa.

Infine, l'ultimo sistema di transizioni è chiamato arc-hybrid ed è formalmente definito in 3.6. Come si evince dal nome, questo sistema utilizza un approccio misto derivante dai due sistemi precedentemente analizzati. In particolare, vengono sfruttate le transizioni RIGHT-ARC e SHIFT del sistema arc-standard e la transizione LEFT-ARC del sistema arc-eager, fatta eccezione per la sua precondizione.

In questo sistema, come anche nell'arc-standard, le relazioni di dipendenza vengono costruite secondo un approccio bottom-up.

A questo punto, facendo riferimento all'algoritmo di figura 3.4, in particolare alla terza riga, è possibile costruire la funzione oracle o utilizzando come

Transizioni	
LEFT-ARC	$(\sigma\|w_i, w_j\|\beta, A) \Rightarrow (\sigma, w_j\|\beta, A \cup (w_j, r, w_i))$
RIGHT-ARC	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_i, \beta, A \cup (w_i, r, w_j))$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$

Tabella 3.6: Sistema di transizioni Arc-hybrid.

sistema di transizione l'arc-standard, come descritto in [39]. In particolare, considerando una frase S con il corrispondente dependency tree annotato $T^{(j)} = (V^{(j)}, A^{(j)})$ è possibile calcolare la funzione oracle nel seguente modo:

$$o(c) = \begin{cases} \text{LEFT-ARC} & \text{se } (\sigma_1, \text{ r, } \sigma_2) \in A^{(j)} \\ \text{RIGHT-ARC} & \text{se } (\sigma_2, \text{ r, } \sigma_1) \in A^{(j)} \text{ e } \forall \text{ w, } r', \\ & \text{se } (\sigma_1, r', \text{ w}) \in A^{(j)} \text{ allora } (\sigma_1, r', \text{ w}) \in \text{A} \\ \text{SHIFT} & \text{altrimenti} \end{cases} \tag{3.10}$$

Questa costruzione per l'oracolo è possibile partendo dal presupposto che per ogni frase S sia possibile costruire una sequenza di transizioni che dia come risultato finale un albero delle dipendenze projective.

Il risultato dell'applicazione della corretta transizione alle diverse configurazioni permette di ottenere la computazione completa, in cui nella configurazione finale si avranno tutte le relazioni di dipendenza del dependency tree per la frase S di input.

L'oracolo descritto nell'equazione 3.10 è detto *statico*, in quanto calcola una sola sequenza di transizioni per ogni dependency tree. Pertanto, utilizzando un oracolo statico, il parser viene addestrato a classificare solo le configurazioni dalle quali è possibile ottenere il corretto dependency tree. Poiché è probabile che il parser commetta un qualche errore durante la fase di test incontrando configurazioni che non ha visto nella fase precedente, questa metodologia di addestramento può risultare problematica [56].

Per risolvere questo problema in [57, 58] viene proposto un oracolo *dinamico*, il quale, per ogni dependency tree può generare più sequenze di transi-

zioni, così da potersi adattare dinamicamente al cambiamento delle configurazioni. Utilizzando questo oracolo, una transizione è considerata ottimale per una data configurazione se il dependency tree che può essere generato dopo aver eseguito questa transizione, non è peggiore del dependency tree che sarebbe potuto essere costruito prima di eseguire quest'ultima transizione, in termini di accuratezza rispetto al gold standard tree.

3.4 Graph-based dependency parsing

Gli approcci graph-based per il parsing a dipendenze cercano nello spazio dei possibili alberi di dipendenza per una data frase, quello che massimizza un certo score. Lo spazio di ricerca viene codificato come il grafo ottenuto connettendo fra di loro tutte le parole della frase di input, mentre lo score rappresenta il punteggio che viene associato ad ogni dependency tree facente parte dell'insieme dei dependency tree candidati per una data frase.

Formalmente, lo score per un dependency tree T può essere visto come la somma degli score attribuibili ai suoi sotto-grafi $G_1, ..., G_m$:

$$score(T) = \sum_{i=1}^{m} score(G_i) \tag{3.11}$$

Tramite l'utilizzo di questo approccio, durante la *fase di apprendimento* si assegna uno score ai possibili dependency tree associati alla frase di input; mentre nella *fase di parsing* si cerca, dato un modello, il dependency tree con lo score più alto.

Generalmente, i parser graph-based raggiungono un'accuratezza maggiore rispetto ai transition-based, in quanto esplorano completamente lo spazio di ricerca associato ad una data frase. Inoltre, i parser graph-based possono produrre dependency tree non-projective.

Di contro, la complessità computazionale di un parser graph-based è asintoticamente maggiore rispetto a quella di un parser transition-based.

3.4.1 Fase di parsing

Data una fase S di input, si cerca il miglior dependency tree $\hat{T}$, nell'insieme di tutti i dependency tree candidati $\mathcal{G}_S$, che massimizzi lo score:

$$\hat{T}(S) = \underset{T \in \mathcal{G}_S}{arg\,max}\ score(T) \tag{3.12}$$

Una delle strategie più semplici per definire lo score di un dependency tree è quella descritta in [59], chiamata *arc-factored*. Utilizzando un parser arc-factored si assegna uno score ad ogni singolo arco $(w_i, r, w_j) \in A$, mentre lo score finale per un dependency tree $T = (V, A)$ è dato dalla somma degli score assegnati alle singole relazioni:

$$\hat{T} = \underset{T \in \mathcal{G}_S}{arg\,max}\ score(T) = \underset{T \in \mathcal{G}_S}{arg\,max} \sum_{(w_i, r, w_j) \in A} score(w_i, r, w_j) \tag{3.13}$$

I dependency tree candidati $\mathcal{G}_S$ possono essere tutti contenuti all'interno di uno stesso grafo, formato da tutte le parole della frase S connesse tra loro.

Quindi, data una frase di input S, si costruisce il grafo pesato orientato e completo i cui vertici sono le parole della frase S e gli archi diretti rappresentano le relazioni testa-dipendente. Inoltre, è necessario aggiungere il nodo root con archi uscenti verso tutti gli altri vertici. I pesi associati agli archi del grafo corrispondono ai punteggi delle relazioni testa-dipendente forniti dal modello derivato dalla fase di addestramento.

Formalmente, seguendo l'approccio descritto in [59], si può definire il grafo diretto $\mathcal{G}_S = (V_S, A_S)$ dei possibili alberi candidati, partendo da una frase $S = w_0, w_1, ..., w_n$ e un insieme di relazioni di dipendenza $R = r_1, ..., r_m$, dove:

- $V_S = w_0, w_1, ..., w_n$, dove $w_0 = root$;
- $A_S = (w_i, w_j) | \forall w_i, w_j \in V_s,\ dove\ j \neq 0$
- $score(w_i, w_j) = max_{r \in \mathcal{R}}\ score(w_i, r, w_j)$

Una volta ottenuto il grafo pesato orientato e completo, si deve cercare il dependency tree la cui somma totale dei pesi sugli archi sia massima. Questo problema, nella teoria dei grafi, è noto come *massimo albero di copertura* (o *maximum spanning tree, MST*).

Il massimo albero di copertura per un grafo orientato $G = (V, A)$ è il sottografo $G' = (V', A')$, che ha la somma dei pesi degli archi più alta e soddisfa due condizioni:

- i nodi del sottografo G' sono esattamente gli stessi del grafo originale G: $V' = V$;
- il sottografo G' è un albero orientato.

Quindi, utilizzare un parser arc-factored equivale a trovare il massimo albero di copertura in un grafo pesato orientato e completo. I dependency tree candidati $\mathcal{G}_S$ possono essere sia projective che non-projective. Infatti, non assegnando alcun vincolo durante la ricerca dell'MST, non è possibile stabilire se il dependency tree risultante avrà proprietà projective o non-projective. Un algoritmo senza alcun tipo di restrizione sull'MST ottenuto è quello sviluppato indipendentemente da Chu-Liu [60] ed Edmonds [61], riportato in figura 3.5.

Come descritto nell'algoritmo, il primo passo è quello di ricercare su G_S, per ogni nodo $w_j \in V$, l'arco entrante con lo score massimo per costruire il grafo G_M. Se G_M non contiene cicli, allora è necessariamente il massimo albero di copertura, quindi l'algoritmo termina. In caso contrario, l'algoritmo considera un qualsiasi ciclo di G_M scelto in modo arbitrario. Identificato il ciclo, questo viene utilizzato come parametro per la funzione CONTRACT che compatta il ciclo in un unico nodo. Successivamente vengono ricalcolati i pesi degli archi entranti e uscenti chiamando ricorsivamente l'algoritmo sul nuovo grafo contratto. Prima della conclusione dell'algoritmo viene ricostruito il grafo originale espandendo, tramite la funzione EXPAND, i nodi contratti precedentemente.

```
function ChuLiuEdmonds(G_S = (V_S, A_S), score)
    A ← {(ŵ_i, w_j) | w_j ∈ V, ŵ_i = arg max_{w_i∈V} score(w_i, w_j)}
    G_M ← {V_S, A}
    if G_M non contiene cicli then
        return G_M
    else
        C ← GetCycle(G_M)
        G_C ← Contract(G_M, C, score)
        G ← ChuLiuEdmonds(G_C, score)
        T ← Expand(G,C)
        return T

function Contract(G,C) return grafo contratto

function Expand(G,C) return grafo espanso
```

Figura 3.5: Algoritmo Chu-Liu/Edmonds per la ricerca del massimo albero di copertura in un grafo pesato orientato e completo nel modello arc-factored, come descritto in [62].

Considerando un arbitrario grafo diretto, l'algoritmo impiega un tempo pari a $O(mn)$, dove m è il numero di archi e n il numero di nodi. Siccome l'algoritmo inizia costruendo un grafo completamente connesso $m = n^2$, il tempo totale diventa $O(n^3)$.

Un'implementazione più efficiente è quella proposta in [63] che ha tempo di esecuzione di $O(m + n\ log\ n)$.

Al contrario dell'algoritmo greedy e ricorsivo ChuLiuEdmonds, che può restituire un dependency tree sia projective che non-projective, quello di Eisner [64], descritto in figura 3.7, introduce alcuni vincoli per i quali tutti gli alberi delle dipendenze sono esclusivamente projective.

In questo algoritmo si usa la tecnica della programmazione dinamica sfruttando una tabella $C[s][t][d][c]$ per rappresentare il sottoalbero che ha la mas-

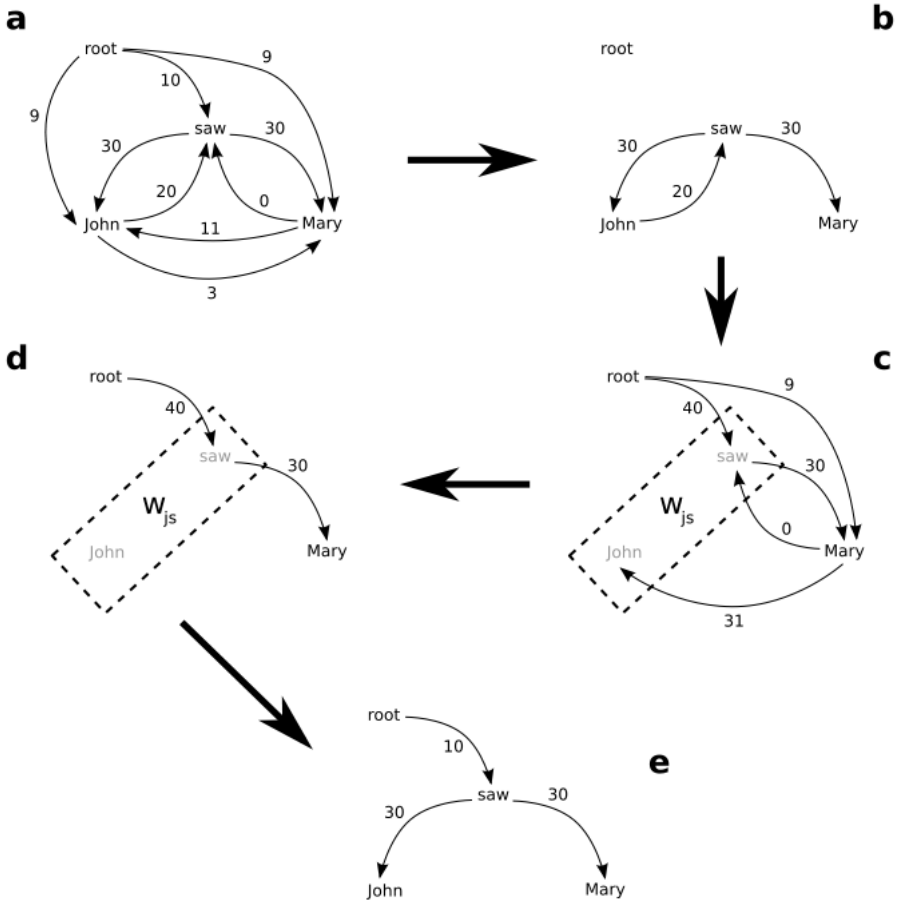


Figura 3.6: Esecuzione dell'algoritmo Chu-Liu/Edmonds per la frase *John saw Mary*, illustrato in [39]. *(a)* rappresenta il grafo dei candidati G_S; *(b)* rappresenta il grafo G_M i cui archi sono quelli con i valori massimi; *(c)* mostra il grafo compatto ottenuto dall'applicazione della funzione CONTRACT all'unico ciclo presente in G_M *John* $\rightarrow$ *saw* $\rightarrow$ *John*; *(d)* è il grafo G_M dei massimi archi entranti ottenuto dal grafo compatto; infine, *(e)* è l'espansione del grafo contratto ottenuto dalla funzione EXPAND.

sima somma dei pesi per le parole da w_s a w_t con $s \leq t$, con direzione $d \in 0, 1$ e indice di completamento $c \in 0, 1$. Nel caso in cui $d = 1$, allora w_s rappresenta la testa del sottoalbero, altrimenti la testa è w_t. Mentre se $c = 1$ allora il sottoalbero è completo, altrimenti è incompleto. L'algoritmo riempie la tabella C in modo bottom-up, trovando sottoalberi ottimali per sottostringhe via via crescenti. La complessità temporale dell'algoritmo è $O(n^3)$, mentre quella spaziale è $O(n^2)$ a causa della memorizzazione della tabella C.

```
function EISNER(S = w_0,w_1, ..., w_n, score)
    C[s][s][d][c] ← 0.0 ∀ s,d,c
    for m ← 1 to n do
        for s ← 1 to n do
            t ← s + m
            if t > n then
                break
            C[s][t][0][1] ← max_{s≤q<t}(C[s][q][1][0] +
                C[q+1][t][0][0] + score(w_t, w_s))

            C[s][t][1][1] ← max_{s≤q<t}(C[s][q][1][0] +
                C[q+1][t][0][0] + score(w_s, w_t))

            C[s][t][0][0] ← max_{s≤q<t}(C[s][q][0][0] +
                C[q][t][0][1])

            C[s][t][1][0] ← max_{s≤q<t}(C[s][q][1][1] +
                C[q][t][1][0])

    return C[0][n][1][0]
```

Figura 3.7: Algoritmo di Eisner per la ricerca del massimo albero di copertura in un grafo pesato orientato e completo nel modello arc-factored, come descritto nell'appendice in [64].

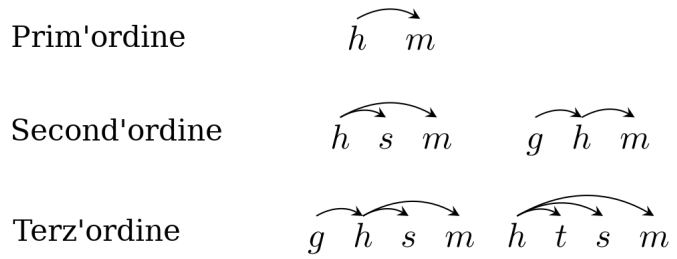


Figura 3.8: Esempi di fattorizzazioni di ordine superiore come riportato in [65]. Ad esempio, la fattorizzazione $h \rightarrow m$ è di prim'ordine in quanto considera una sola relazione di dipendenza. La fattorizzazione $h \rightarrow m$ e $h \rightarrow s$ è di second'ordine sul livello orizzontale in quanto considera due diverse relazioni di dipendenza che, in questo caso, hanno la stessa testa h. Mentre la fattorizzazione $h \rightarrow m$ e $g \rightarrow h$ è sempre di second'ordine, ma espandendosi sul livello verticale.

Higher-order dependency parsing

Come descritto precedentemente, i parser che usano un approccio arc-factored calcolano il punteggio di un dependency tree come la somma dei punteggi assegnati ai singoli archi. È possibile rilassare questo vincolo in modo da considerare anche relazioni più complesse. Ad esempio, nei second-order dependency parsing, dato un arco (w_i, w_j) nel dependency tree, la funzione score può includere sia archi sullo stesso livello dell'albero (dimensione orizzontale, ad esempio i fratelli), sia sui livelli superiori o inferiori (dimensione verticale, ad esempio i nonni). In figura 3.8 sono riportate alcune fattorizzazioni di ordine superiore.

Per i projective dependency graph esistono alcuni algoritmi efficienti per il parsing del secondo e terz'ordine. Ad esempio, modificando leggermente l'algoritmo per il parsing di Eisner, è possibile catturare anche le relazioni di second'ordine mantenendo la complessità invariata.

L'algoritmo Chu-Liu/Edmonds, usato per la ricerca del MST non-projective, considerando gli ordini superiori al primo è *NP-hard*, pertanto intrattabi-

le. Una soluzione euristica, descritta in [66], consiste nell'ottenere prima un dependency tree projective e successivamente combinarlo con tecniche di trasformazione sui grafi per produrre un dependency tree non-projective.

3.4.2 Fase di learning

Durante la fase di apprendimento si ha come obiettivo quello di apprendere la funzione score, utilizzata per assegnare un peso agli archi del grafo dei dependency tree candidati. Per farlo ci si avvale di un treebank utilizzato come training set.

Una buona approssimazione per questa funzione è data da un classificatore formato dalla rappresentazione in feature dei sottografi in cui è possibile suddividere un dependency tree, con i pesi relativi alle feature w. Nel caso di un approccio arc-factored, la rappresentazione in feature $f(w_i, r, w_j)$ è relativa alle singole relazioni di dipendenza, per cui lo score sarà calcolato come:

$$score(w_i, r, w_j) = w \cdot f(w_i, r, w_j) \tag{3.14}$$

La funzione f può includere un qualsiasi tipo di feature rilevante della frase S. Quest'ultime vengono estratte in modo totalmente simile a quanto avviene nell'approccio transition-based, descritto nella sezione 3.3.2.

Il fatto che entrambi gli approcci, transition-based e graph-based, estraggano le stesse feature non è sorprendente, in quanto in entrambi i casi si devono cercare di acquisire quante più informazioni possibili sulle relazioni testa-dipendente.

3.5 Metriche di valutazione

Le prestazioni di un parser vengono misurate sperimentalmente utilizzando un treebank di test, il quale non deve mai essere preso in considerazione durante la fase di training per non invalidare i risultati finali.

Durante la fase di testing il parser prende in input le frasi $S^{(i)}$ dal treebank di test e per ognuna di esse predice un depencency tree $\hat{T}^{(i)}$, che sarà poi confrontato con il dependency tree corretto $T^{(i)}$ (gold standard).

Il confronto fra $\hat{T}^{(i)}$ e $T^{(i)}$ viene fatto attraverso l'uso di diverse metriche. Quelle più comunemente utilizzate in letteratura per il dependency parsing sono:

- *Label Accuracy score* (LA), ovvero la percentuale di parole predette che hanno la stessa etichetta della relazione di dipendenza presente nel gold standard;
- *Unlabeled Attachment Score* (UAS), ovvero la percentuale di parole predette che hanno la stessa testa della relazione di dipendenza presente nel gold standard;
- *Labeled Attachment Score* (LAS), ovvero la percentuale di parole predette che hanno sia la stessa etichetta che la stessa testa della relazione di dipendenza presente nel gold standard.

Per come sono definite queste metriche, il valore di LAS non può mai essere maggiore di quello di UAS. Questo perché una relazione valida per la metrica LAS ha sia il tipo di dipendenza che la testa corretta; mentre per UAS è sufficiente che siano uguali solo le teste.

Queste metriche sono utilizzate anche nella sezione 4.3, in cui vengono riportati tutti gli esperimenti oggetto di questa tesi.

Se si rilassa la proprietà *unique root property* dei dependency tree oppure si vogliono valutare i tipi di dipendenza singolarmente, allora si possono prendere in considerazione ulteriori metriche:

- *Precision* (P): percentuale di dipendenze con un tipo specifico correttamente predette dal sistema:

$$Precision = \frac{\#\ corrette}{\#\ sistema} \tag{3.15}$$

- *Recall* (C): percentuale di dipendenze con un tipo specifico nel gold standard che sono state correttamente predette:

$$Recall = \frac{\#\ corrette}{\#\ gold} \tag{3.16}$$

- *F1 score*: media armonica fra Precision e Recall:

$$F1\ score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3.17}$$

Punteggiatura

Spesso in letteratura si trovano valutazioni di parser in cui la punteggiatura viene esclusa. In generale, nel task di dependency parsing i token vengono trattati tutti allo stesso modo, assegnando le relazioni di dipendenza sia alle parole che ai segni di punteggiatura sulla base di alcuni criteri. Tuttavia, in [67] vengono descritti alcuni motivi per i quali non è necessario tener conto della punteggiatura durante la valutazione. In primo luogo, spesso le teste delle relazioni di dipendenza per la punteggiatura non sono così ben definite come quelle delle parole. Di conseguenza, la punteggiatura non sempre è annotata in modo coerente nei diversi treebank. Inoltre, i risultati sperimentali hanno mostrato che l'accuratezza del parsing diminuisce nelle frasi in cui sono presenti più segni di punteggiatura, oltre al punto di fine frase. Una possibile ragione di questo risultato è che i parser restituendo dependency tree con proprietà projective soddisfano il vincolo di non avere "collegamenti incrociati" e un errore nella punteggiatura può impedire la corretta creazione di relazioni di dipendenza fra le parole. Infine, anche nei parser transition-based, come quello descritto in [68], la punteggiatura può causare delle imprecisioni. In particolare, nel template di feature utilizzato in [68] sono state incluse le cosiddette *valency feature* che informano il parser sul numero e il tipo dei modificatori di destra e di sinistra associati ad

un predicato, come ad esempio un verbo. Tuttavia, le informazioni derivate dalle valency feature potrebbero risultare scorrette nel momento in cui fra il verbo e il suo modificatore si interpongono uno o più segni di punteggiatura.

In [69] viene messo nuovamente in evidenza come i parser che generano dependency tree projective siano più sensibili all'utilizzo della punteggiatura nelle frasi rispetto ai parser che generano dependency tree non-projective. Inoltre, viene anche dimostrato che i parser neurali siano, mediamente, più sensibili alla punteggiatura rispetto a quelli che non fanno uso di tecniche di machine learning.

Anche durante lo shared task organizzato da *EVALITA* (*Evaluation of NLP and Speech Tools for Italian*) nel 2014, in cui era presente il task di dependency parsing, tutti i risultati sono stati riportati in [70] senza prendere in considerazione i segni di punteggiatura. Infatti, i codici[2] forniti dagli organizzatori e utilizzati per la valutazione dei parser per il task *DPIE* (*Dependency Parsing for Information Extraction*) non tengono in considerazione i segni di punteggiatura.

Pertanto, sembra che lo standard de facto adottato sia quello di non considerare i segni di punteggiatura nella valutazione dei dependency parser come accade anche in quelli più recenti descritti in [71, 72, 73]. Infatti, per tutti questi parser viene utilizzato lo script di valutazione standard eval.pl del CoNLL-X shared task che di default non considera i segni di punteggiatura. Inoltre, nel recente dependency parsing sviluppato da Vacareanu et al. [2] (PaT), descritto nella sezione 4.1, oltre a non considerare i segni di punteggiatura non vengono nemmeno considerati i token il cui valore per l'attributo UPOS delle UD è "SYM", come accade anche in [73, 74]. Perciò, per un più equo confronto, durante la descrizione di tutti gli esperimenti oggetto di questa tesi, presentati nel successivo capitolo 4, verrà utilizzato lo stesso codice eval.pl che è stato usato per calcolare le metriche di valutazione di PaT.

[2] `http://medialab.di.unipi.it/Evalita2014/resources/evaluation_package_DPIE.tgz`

3.6 Modelli neurali per il parsing a dipendenze

Con l'avvento del deep learning tutti gli sforzi legati alla corretta selezione delle feature per i dependency parser si sono gradualmente attenuati, portando una maggior concentrazione verso la creazione di architetture sempre più efficienti.

Data la vastissima gamma di ricerche che sono state fatte nello sviluppo di architetture neurali sempre migliori, nelle sezioni successive verranno presentanti alcuni esempi di parser neurali sia transition-based (sezione 3.6.1) che graph-based (sezione 3.6.2).

3.6.1 Esempi di parser Transition-based

Kiperwasser e Goldberg nel 2016 in [71] hanno pubblicato un nuovo approccio che utilizza un'architettura minimale basata su una LSTM bidirezionale deep con k livelli, che dipende solo dagli embedding ottenuti dalla frase di input e senza feature aggiuntive. In particolare, ad ogni parola della frase vengono associati i vettori di word embedding e PoS tag embedding e calcolato il contesto tramite la BiLSTM. Nonostante l'apparente semplicità del modello, riportano risultati molto interessanti. Il parser[3] è sviluppato in due versioni: transition-based, con sistema di transizione arc-hybrid e oracolo dinamico e graph-based, descritta nella sezione successiva.

Nell'approccio transition-based lo score di una transizione viene appreso dalla rete MLP considerando la rappresentazione di una configurazione ottenuta combinando i vettori di embedding associati alle prime tre parole in cima allo stack e alla prima parola nel buffer. L'algoritmo utilizzato per il parsing del massimo albero di copertura è di tipo greedy deterministico.

Nel parser[4] transition-based sviluppato nel 2018 da Ma et al. [74] viene utilizzata una pointer network, introdotta per la prima volta in [75], congiun-

[3] `https://github.com/elikip/bist-parser`

[4] `https://github.com/XuezheMax/NeuroNLP2`

tamente ad uno stack interno. L'architettura neurale risultante è chiamata *stack-pointer network*.

Le reti di puntatori sono una categoria di reti neurali in grado di apprendere la probabilità condizionata di una sequenza di output i cui elementi sono token discreti (in questo caso, indici di parole in una frase) corrispondenti a precise posizioni in una data sequenza di input. Successivamente, tramite un meccanismo di attention, implementano un puntatore in grado di selezionare come token di output uno presente nella sequenza di input. I problemi che si riescono ad affrontare e risolvere con le reti di puntatori non sono risolvibili con i più classici modelli seq2seq in quanto, le categorie discrete degli elementi di output dipendono dalla dimensione dell'input e non possono essere decise a priori.

L'architettura neurale della stack-pointer network utilizzata da Ma et al. [74] è costituita da un encoder e un decoder. L'encoder è formato da una combinazione di CNN e LSTM bidirezionali. Per ogni parola, la CNN viene utilizzata per estrarre il character embedding che, dopo essere stato concatenato a un word embedding e al PoS embedding, viene dato in input alla BiLSTM che codifica le informazioni sul contesto della parola. Il decoder è implementato tramite una LSTM ed è formato da un sistema di transizioni top-down che sfrutta un buffer, per contenere le parole non ancora collegate da un arco e uno stack, per contenere le parole parzialmente elaborate. Lo stack viene inizializzato solo con il simbolo $ corrispondente alla root.

Le transizioni possibili per il parser sono due: *Shift-Attach-p* e *Reduce*. Data una configurazione e considerata w_i la parola in cima allo stack, la stack-pointer network restituisce l'indice p corrispondente ad una parola della frase di input. Per scegliere quale transizione applicare, viene fatto un semplice controllo: se $p \neq i$ allora la parola puntata w_p è considerata la dipendente di w_i. Quindi il parser sceglie la transizione Shift-Attach-p per spostare w_p dal buffer allo stack e costruire l'arco $w_i \rightarrow w_p$. Se, viceversa, $p = i$ allora w_i si ritiene abbia già collegati tutti i suoi figli per cui viene applicata la transizione Reduce per eliminarla dallo stack.

In sintesi, il modello inizialmente legge l'intera frase $S = w_0, w_1, ..., w_n$ e codifica ogni parola w_i creando uno stato nascosto e_i tramite l'encoder. Per ogni configurazione, il decoder riceve in input lo stato nascosto e_i dell'encoder relativo alla parola w_i in cima alla stack e genera uno stato nascosto d_t.

Successivamente, d_t insieme alla sequenza di stati nascosti generati dall'encoder per ogni parola presente nel buffer più e_i, vengono utilizzati per calcolare il vettore di attention a^t.

Infine, il vettore di attention a^t, a cui è applicata la funzione softmax, viene utilizzato per restituire la posizione p con lo score più alto. L'elaborazione termina quando rimane solo la root nello stack.

Durante la fase di training viene addestrato simultaneamente al parser un classificatore multiclasse, seguendo l'approccio di Dozat e Manning, per predire le etichette delle relazioni di dipendenza.

Il numero di transizioni applicate per la costruzione di un dependency tree per una frase lunga n sono pari a $2n - 1$. Considerando il costo di $O(n)$ per il calcolo dell'attention ad ogni step, si raggiunge una complessità temporale totale di $O(n^2)$.

González e Rodríguez nel 2019 [73] hanno sviluppato un transition-based parser[5] sfruttando esattamente l'architettura neurale progettata da Ma et al. in [74], ma introducendo un più semplice sistema di transizione left-to-right che non richiede né uno stack né un buffer per l'elaborazione della frase di input.

In questo approccio le configurazioni del parser corrispondono a un *focus word pointer* i, ovvero un puntatore all'i-esima parola che viene attualmente elaborata. Il decoder inizia con i che punta alla prima parola della frase di input. Per ogni configurazione il sistema di transizione è formato da una sola transizione: *Attach-p*, che collega la dipendente w_i puntata da i, alla testa in posizione p-esima nella frase. In questo modo si crea la relazione di dipendenza $w_p \rightarrow w_i$ e si sposta il puntatore i di una posizione verso destra. La scelta di p è fatta sempre attraverso un meccanismo di attention e nel

[5] `https://github.com/danifg/Left2Right-Pointer-Parser`

caso in cui $p = 0$, la testa della relazione è root. La fase di parsing termina quando viene creato l'arco la cui dipendente è l'ultimo token della frase di input.

Il decoder richiede n step per la generazione di un dependency tree, a partire da una frase di n parole. Considerando il costo di $O(n)$ per il calcolo dell'attention ad ogni step, si raggiunge una complessità temporale totale di $O(n^2)$ per il decoding.

3.6.2 Esempi di parser Graph-based

Nell'approccio graph-based del parser BIST di Kiperwasser e Goldberg [71], la fase di creazione degli embedding e del contesto, per ogni parola, è speculare a quella utilizzata nell'approccio transition-based. Al vettore di feature restituito dalla BiLSTM si applica un classificatore MLP, con un solo livello nascosto, che ne calcola lo score combinando il vettore di embedding della testa e della dipendente. L'algoritmo usato per il parsing del massimo albero di copertura è Eisner.

Dozat e Manning nel 2017 in [3] presentano un parser graph-based di tipo arc-factored basato sull'architettura del suddetto parser BIST di Kiperwasser e Goldberg. Uno dei loro contributi più significativi è l'aggiunta di un livello MLP *biaffine*, al posto di quello affine originale, modificando in parte la struttura del modello. In particolare, prima dell'utilizzo della trasformazione biaffine, l'output della BiLSTM viene dato in input a diversi MLP a tre livelli con l'obiettivo di eliminare le informazioni irrilevanti. Infine, all'output prodotto dai livelli MLP viene applicata la trasformazione biaffine. L'algoritmo usato per il parsing del massimo albero di copertura è Chu-Liu/Edmonds.

Questo approccio ha portato ottimi risultati raggiungendo lo stato dell'arte al momento della pubblicazione. Ad oggi è disponibile nella sua terza versione[6].

Fondamentalmente, grazie alla struttura biaffine introdotta da Dozat e Manning è possibile codificare le relazioni di prim'ordine genitore-figlio.

[6] `https://github.com/tdozat/Parser-v3`

Nel 2019 Ji et al. [76] hanno sviluppato un dependency parser[7] utilizzando una *GNN* (*Graph Neural Network*), proposta per la prima volta in [77], in modo da riuscire a catturare anche le relazioni di ordine superiore. In particolare, la GNN è utilizzata per l'apprendimento delle rappresentazioni dei nodi nel dependency tree.

L'idea generale è che dato un grafo pesato, una GNN mantiene per ogni livello un insieme di rappresentazioni di nodi, ottenute aggregando le rappresentazioni dei nodi ad essi adiacenti nei livelli inferiori. È possibile, quindi, applicare quest'idea al task del dependency parsing, costruendo una GNN sul grafo completo e pesato ottenuto dalla frase di input seguendo un approccio graph-based. Successivamente, creando un stack di GNN, le rappresentazioni di ogni nodo del grafo tenderanno incrementalmente a catturare anche le reazioni di ordine superiore al primo. La fase iniziale di encoding della frase è simile a quella utilizzata nel parser BIST di Kiperwasser e Goldberg [71], con la differenza che viene utilizzato Glove per il word embedding. Mentre la fase di parsing è analoga a quella utilizzata da Dozat e Manning [3].

Un'estensione dell'architettura del parser BIST di Kiperwasser e Goldberg è quella descritta in [78] da Nguyen et al. la cui seconda versione[8] è stata rilasciata nel 2018. In questo caso, l'architettura neurale proposta cerca di apprendere congiuntamente sia il dependency parsing graph-base di tipo arc-factored, come per il parser BIST, che i PoS tag.

L'idea alla base di questo approccio è nata dal fatto che molti parser neurali utilizzano i PoS tag predetti automaticamente come feature. Tuttavia, nel caso in cui il PoS tagger facesse degli errori, questi verrebbero successivamente propagati in tutto il modello.

La principale modifica rispetto al parser BIST è l'aggiunta del componente che si occupa dell'apprendimento dei PoS tag, formato da una LSTM bidirezionale e da un MLP con uno strato nascosto, il cui numero di nodi è pari al numero dei PoS tag.

[7] `https://github.com/AntNLP/gnn-dep-parsing`
[8] `https://github.com/datquocnguyen/jPTDP`

In particolare, data una frase di input, ciascun token viene rappresentato tramite la concatenazione dei vettori di word embedding e character embedding. La BiLSTM utilizzata per il PoS tagging prende in input il vettore di embedding di ogni parola restituendo il vettore di feature associato. Successivamente, l'output della BiLSTM viene dato in input all'MLP che predice il valore del PoS tag applicando una funzione softmax. Infine, è possibile calcolare e minimizzare la funzione di loss cross-entropy calcolata fra il PoS tag predetto dall'MLP e il PoS tag reale.

Una volta ottenuti i PoS tag, questi vengono rappresentati tramite vettori di embedding e concatenati all'iniziale rappresentazione dei token. L'embedding risultante è utilizzato come input per una BiLSTM, al cui output si applicano due diversi classificatori MLP: uno con un solo nodo di output per calcolare lo score degli archi e l'altro con tanti nodi quante le possibili relazioni di dipendenza. L'algoritmo utilizzato per il parsing del massimo albero di copertura è Eisner.

Capitolo 4

UmBERTo per il dependency parsing

Successivamente all'introduzione di BERT, molti problemi nell'ambito dell'NLP hanno avuto un notevole incremento prestazionale, soprattutto per quanto riguarda quelli che vengono risolti tramite l'impiego di architetture neurali. Tuttavia, al momento del rilascio, BERT era disponibile solamente per la lingua inglese. Successivamente, sono stati sviluppati altri modelli ad-hoc simili a BERT, ovvero basati sempre sull'architettura dei Transformers, ma per altre lingue.

In questo capitolo si esamina l'importanza e l'impatto dovuto a UmBERTo, descritto precedentemente nella sezione 2.4.2, per quanto riguarda il problema del dependency parsing per la lingua italiana. In particolare, nelle sezioni successive si affronterà questo problema sfruttando l'applicazione delle rappresentazioni linguistiche ottenute tramite UmBERTo, utilizzando entrambe le strategie possibili: il fine-tuning (sezione 4.3.1) e il feature extraction (sezione 4.3.2). Successivamente, si applicherà l'approccio del feature extraction di UmBERTo al parser neurale delle dipendenze PaT [2]. Infine, si renderà più complessa l'architettura neurale di PaT per cercare di migliorare le sue prestazioni nel task di dependency parsing per la lingua italiana.

4.1 PaT: Parsing as Tagging

Molto recentemente Vacareanu et al. in [2] hanno proposto un nuovo metodo per il task di dependency parsing sfruttando gli embedding di BERT. In sintesi, nel loro approccio il problema del dependency parsing viene trattato come un problema di *sequence tagging*, dove, per ogni parola della frase di input, si determina la posizione relativa della testa della relazione di dipendenza rispetto alla sua dipendente e successivamente si predice l'etichetta della relazione.

In particolare, considerando che nei dependency tree ogni token ha una singola testa, che può essere un altro token della frase o il token root, si può trasformare il task di parsing in uno di tagging. Ovvero, il "tag" di un token è rappresentato dal valore corrispondente alla posizione relativa della sua testa (*relpos*) calcolata come segue:

$$relpos = \begin{cases} 0, & \text{se la head è root} \\ head - id, & \text{altrimenti} \end{cases} \tag{4.1}$$

dove *id* è la posizione assoluta del token considerato all'interno della frase, iniziando da 1, mentre *head* è la posizione assoluta corrispondente alla testa della relazione di dipendenza per il token considerato. Un esempio di applicazione di questa formula è illustrato nella figura 4.1, prendendo come riferimento la frase *"the big dog chased the cat"* utilizzata precedentemente nella sezione 3.1.

Tuttavia, siccome la posizione relativa deve essere predetta per ogni token di ogni frase e quest'ultime possono essere di tante lunghezze differenti, gli autori di PaT hanno posto un intervallo limite di [-50, 50], scelto empiricamente, come possibili valori predicibili. In particolare, questo intervallo comprende il 99.99% dei valori corrispondenti a quelli assegnabili alle posizioni relative per i token presenti nelle frasi del training set della UD Inglese versione 2.2, utilizzata in PaT. Questo stesso intervallo viene utilizzato per tutti gli esperimenti descritti nella sezione 4.3 e racchiude il 99.85% dei valori

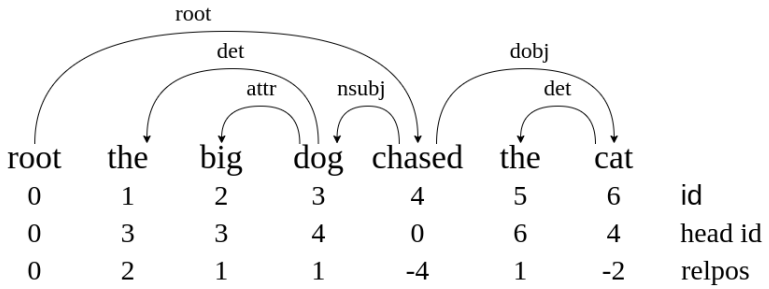


Figura 4.1: Illustrazione grafica del calcolo delle posizioni relative per le parole di una frase nel parser PaT.

assegnabili alle etichette relpos per i token presenti nelle frasi del training set della UD Italian ISDT versione 2.6.

Questa leggera discrepanza fra le due percentuali rappresentanti i valori per relpos compresi nell'intervallo [-50, 50], deriva dal fatto che l'UD Italian ISDT contiene frasi che sono mediamente più lunghe rispetto a quelle della UD Inglese.

Data in input una frase S formata da n token $t_1, ..., t_n$, ogni token t_i viene rappresentato da un vettore di embedding e_i, formato dalla concatenazione degli embedding ottenuti dalla feature extraction di $\text{BERT}_{\text{BASE}}$ Uncased, un word embedding (we), un character encoding ottenuto tramite CNN con max pooling (ce) e un PoS embedding (pos):

$$e_i = \left[t_i^{(bert)} \oplus t_i^{(we)} \oplus t_i^{(ce)} \oplus t_i^{(pos)} \right] \tag{4.2}$$

dove $\oplus$ indica la concatenazione fra vettori.

In particolare, gli embedding relativi ai PoS tag e alle parole vengono appresi durante il training e inizializzati utilizzando l'inizializzazione di *Glorot* o *Xavier* [79]. Questa tipologia di inizializzazione è particolarmente comune per le *Deep Neural Networks* (*DNN*) e comporta che i valori dei parametri siano scelti in modo uniforme nell'intervallo $\pm\sqrt{6/(r+c)}$, dove r e c sono il numero di righe e colonne, rispettivamente, della struttura dello specifico peso della rete.

Per ogni token t_i l'embedding risultante e_i viene dato in input ad una LSTM bidirezionale da cui si ottiene uno stato nascosto h_i che viene utilizzato per predire sia la posizione relativa della testa, sia l'etichetta corrispondente alla relazione di dipendenza precedentemente formatasi.

Nello specifico, per quanto riguarda la predizione della posizione relativa della testa, lo stato nascosto h_i viene dato in input ad un MLP, seguito da un livello lineare e da una softmax. Quest'ultima restituisce la distribuzione di probabilità su tutti i possibili valori assegnabili alle posizioni relative. Per la predizione dell'etichetta corrispondente alla relazione di dipendenza, viene concatenato l'output dell'MLP corrispondente a t_i insieme alla testa predetta t_i^{head} e passato ad un livello lineare, diverso dal precedente, seguito da una softmax. Nuovamente, quest'ultima restituisce la distribuzione di probabilità su tutte le possibili etichette per le relazioni di dipendenza.

La funzione di loss utilizzata nel modello è la cross-entropy, calcolata prendendo in considerazione i valori predetti e quelli originali sia per le etichette delle relazioni di dipendenza sia per le posizioni relative.

L'architettura appena descritta è mostrata in figura 4.2, considerando come input la frase "*(root) John eats cake*".

Analizzando l'architettura del parser e volendolo inserire in una categoria fra transition-based e graph-based, lo si può considerare, in un certo senso, come appartenente a quest'ultima. Infatti, durante la fase di predizione si può generare la probabilità che la parola i-esima sia la testa della relazione di dipendenza formata con la parola j-esima, per ogni coppia di parole w_i e w_j nella frase di input. Questa costruzione forma un grafo completo: ogni parola della frase corrisponde a un nodo e ogni relazione di dipendenza tra due parole (w_i, w_j) corrisponde a un arco da w_i a w_j che rappresenta la probabilità che il nodo w_i sia la testa di w_j. Il grafo dovrà, successivamente, essere modificato per poter ottenere un dependency tree valido. Infatti, non è presente alcuna strategia che eviti la formazione di cicli durante la creazione dell'MST. Per questo motivo, gli autori di PaT hanno introdotto due diversi approcci da poter utilizzare nella fase di post elaborazione per gestire gli

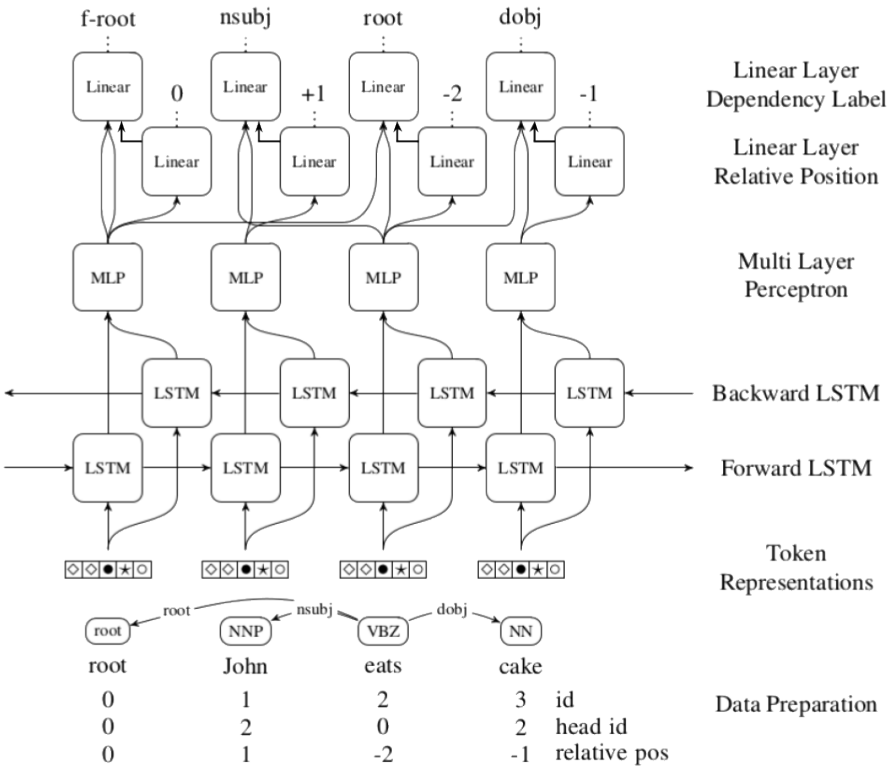


Figura 4.2: Illustrazione grafica delle varie componenti dell'architettura del parser PaT, data in input la frase "*(root) John eats cake*" [2].

eventuali cicli:

1. *Approccio greedy*: vengono ordinate globalmente le relazioni di dipendenza sulla base del peso ad esse associato. Successivamente, si aggiungono in modo incrementale all'albero di output le dipendenze che hanno la probabilità più alta senza creare un ciclo, finché tutti i token della frase non sono stati considerati nelle relazioni;

2. *Approccio ottimale*: si cerca il MST utilizzando l'algoritmo di Chu-Liu/Edmonds, descritto nella sezione 3.4.1;

Un'ultima alternativa è quella di non rimuovere i cicli, lasciando l'output del modello invariato.

Negli esperimenti descritti nelle sezioni successive è stato sempre utilizzato un approccio greedy per la rimozione dei cicli.

4.2 Dataset

In tutti gli esperimenti riportati nelle sezioni successive verrà sempre utilizzato il treebank di dominio generico per l'italiano ISDT[1] (*Italian Stanford Dependency Treebank*) versione 2.6, preso dalla collezione delle UD.

4.2.1 UD Italian ISDT versione 2.6

Il corpus UD Italian, annotato secondo lo schema di annotazione delle Universal Dependencies versione 2, è stato ottenuto tramite la conversione da *ISDT* che presentava come schema di annotazione Stanford Dependencies. La conversione è avvenuta come descritto in [80].

L'ISDT è stato rilasciato e messo a disposizione per la prima volta durante il task di dependency parsing in Evalita 2014 [70] ed è stato ricavato dalla conversione del corpus *MIDT* (*the Merged Italian Dependency Treebank*). A

[1] `https://universaldependencies.org/treebanks/it_isdt/index.html`

	Frasi	Parole	Token
Training set	13.121	276.018	257.616
Validation set	564	11.908	11.133
Test set	482	10.417	9.680
Totale	14.167	298.343	278.429

Tabella 4.1: Suddivisione in train, validation e test set della UD Italian ISDT versione 2.6.

sua volta, il MIDT è il risultato dell'unione e conversione nello stesso schema di annotazione di due dependency treebank preesistenti per la lingua italiana:

- *TUT*, (*Turin University Treebank*) [81];
- *ISST-TANL*, rilasciato inizialmente come *ISST-CoNLL* per il CoNLL 2007 shared task e ottenuto a partire dall'*ISST* (*Italian Syntactic-Semantic Treebank*) [82].

La UD Italian ISDT versione 2.6 è formata da 14.167 frasi, estrapolate da testi di dominio generico, suddivise in train, validation e test set, come riportato nella tabella 4.1. Per completezza, in appendice A sono riportate statistiche dettagliate per la UD in questione.

4.3 Esperimenti

Gli esperimenti oggetto di questa tesi, presentati nelle sezioni successive, sono così suddivisi:

- Primo esperimento: fine-tuning di UmBERTo per il task di dependency parsing, trattato come un problema di sequence tagging;
- Secondo esperimento:

  - sostituzione di $\text{BERT}_{\text{BASE}}$ Uncased con UmBERTo Uncased all'interno del parser PaT;
  - semplificazione dell'architettura di PaT per il task di dependency parsing, in modo tale che il vettore contenente gli embedding di input per ogni token della frase sia formato solo dall'embedding risultante dal feature extraction di UmBERTo;

- Terzo esperimento: test dell'architettura formata dal parser PaT congiuntamente ad UmBERTo Uncased ($\text{PaT}_{\text{UmbUnc}}$), analisi dei risultati ottenuti e confronto con il dependency parser allo stato dell'arte per l'italiano di Dozat e Manning [3] versione 1.0;

- Quarto esperimento: modifiche all'architettura di $\text{PaT}_{\text{UmbUnc}}$ descritta e analizzata nell'esperimento precedente, traendo ispirazione dal parser di Dozat e Manning.

Questi esperimenti hanno un duplice obiettivo: analizzare i risultati ottenibili da UmBERTo per il task di dependency parsing per la lingua italiana e valutare i risultati raggiungibili considerando il task di dependency parsing come uno di sequence tagging sfruttando il recente parser PaT [2], integrato con UmBERTo.

Setup

Tutti gli esperimenti sono stati eseguiti su *Google Colaboratory* [83] (chiamato semplicemente *Colab*): un servizio cloud gratuito di Google, con diverse restrizioni sulla quantità e il tempo di utilizzo delle risorse, basato su *Jupyter Notebooks*. Colab viene principalmente sfruttato per l'addestramento e il testing di modelli di ML, in quanto fra le risorse che mette a disposizione sono presenti anche diverse *GPU*. Inoltre, permette una facile interazione con lo spazio di archiviazione presente su *Google Drive*. Per l'esecuzione di ognuno degli esperimenti, elencati precedentemente, sono stati creati dei *Google Colab Notebooks* ad-hoc, sia per le fasi di training sia per il testing.

Per ogni esperimento le metriche di valutazione considerate sono LAS, UAS e LA, descritte precedentemente nella sezione 3.5. Di ogni metrica viene calcolata la media e la deviazione standard, prendendo in considerazione dieci diverse istanze dei modelli appresi sul training set del corpus UD Italian ISDT 2.6. Si è scelto di valutare dieci diverse istanze dei modelli in quanto, come dimostrato in [84], riportando un singolo punteggio si potrebbe essere fortemente influenzati dall'inizializzazione casuale della rete e, quindi, dal seed del generatore casuale, soprattutto nell'ambito delle DNN. Inoltre, valutando più istanze si ottengono dei risultati più precisi delle prestazioni reali del sistema considerato.

In tutti gli esperimenti, il validation set è sempre stato utilizzato per l'early stopping. A quest'ultimo è stato impostato un valore di *patience* pari a cinque, come suggerito in [2] quando si effettua un addestramento di modelli per applicazioni reali.

Infine, siccome il corpus utilizzato è annotato nel formato CoNLL-U sono stati gestiti sia i metadati sia le parole composte presenti in esso.

4.3.1 Primo esperimento: UmBERTo Fine-tuning

Il sistema utilizzato per il fine-tuning di UmBERTo Uncased per il task di dependency parsing, utilizzando lo stesso approccio di PaT, consiste in alcune modifiche apportate allo script `run_ner.py` presente nella libreria Huggingface Transformers versione 4.2.0.

Per il fine-tuning viene aggiunto sopra all'architettura di UmBERTo un classificatore formato da un layer lineare al cui output viene applicata la funzione softmax. Il layer lineare ha tanti output quante le possibili classi a cui l'input può appartenere, mentre la funzione softmax restituisce la classe più probabile di appartenenza dell'input. Il modello risultante viene utilizzato per predire sia la posizione relativa della testa di una relazione di dipendenza rispetto alla dipendente (relpos), sia l'etichetta della relazione stessa.

I risultati ottenuti dal modello sono riportati nella tabella 4.2.

	Dev set		Test set	
	μ	σ	μ	σ
LAS	88.79%	0.31	89.17%	0.25
UAS	91.66%	0.31	91.91%	0.29
LA	94.89%	0.10	95.28%	0.13

Tabella 4.2: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello appreso sul train set del corpus UD Italian ISDT 2.6.

4.3.2 Secondo esperimento: UmBERTo Feature extraction

In questo secondo esperimento è stata utilizzata l'architettura messa a disposizione da PaT, in cui dapprima si è sostituito $\text{BERT}_{\text{BASE}}$ Uncased con UmBERTo Uncased e successivamente è stato utilizzato solo quest'ultimo per l'estrazione delle feature dalle frasi di input. Pertanto, data in input una frase S formata da n token $t_1, ..., t_n$, ogni token t_i viene rappresentato da un vettore di embedding e_i ottenuto tramite il feature extraction di UmBERTo:

$$e_i = \left[t_i^{(UmBERTo)} \right] \tag{4.3}$$

Il resto del modello di PaT è stato semplificato in modo che e_i fosse utilizzato come input per una LSTM bidirezionale, formata da un singolo layer ricorrente, seguita da un livello di classificazione con una softmax come funzione di attivazione. Quest'ultimo livello ha la medesima struttura di quello utilizzato nell'esperimento precedente.

Questa tipologia di architettura viene spesso trovata in letteratura [85, 1] quando si applicano le rappresentazioni linguistiche ottenute da BERT o dai modelli da esso derivati, tramite la strategia del feature extraction.

I risultati ottenuti dal modello sono riportati nella tabella 4.3.

	Dev set		Test set	
	μ	σ	μ	σ
LAS	89.02%	0.32	89.18%	0.23
UAS	92.42%	0.13	92.54%	0.19
LA	92.75%	0.29	92.90%	0.29

Tabella 4.3: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello appreso sul train set del corpus UD Italian ISDT 2.6.

4.3.3 Risultati e confronti

I risultati ottenuti per le metriche LAS e UAS utilizzando UmBERTo Uncased per il feature extraction (tabella 4.3), sono leggermente migliori rispetto a quelli conseguiti tramite il fine-tuning di UmBERTo Uncased (tabella 4.2). Nello specifico, dai risultati emerge che il feature extraction sia migliore rispetto al fine-tuning di circa lo 0.1% per la metrica LAS e dello 0,6% per la metrica UAS.

Al contrario, per la metrica LA, che prende in considerazione solo l'etichetta delle parole, il fine-tuning è migliore di circa il 2% rispetto al feature extraction. Questa differenza può essere spiegata considerando che il fine-tuning di UmBERTo Uncased nei token-level task come, ad esempio, il PoS tagging, raggiunge un'accuracy di oltre il 98%, come riportato dagli stessi autori[2], utilizzando come dataset la UD Italian ISDT. Pertanto, osservando che i DEPREL tag della UD Italian ISDT versione 2.6 sono più del doppio rispetto ai PoS tag, i risultati sono in linea con quelli attesi. Inoltre, va anche tenuto in considerazione che in nessuno di questi due esperimenti è stato fatto un tuning dei parametri. Di quest'ultimo aspetto, l'architettura utilizzata per il feature extraction, in particolare la BiLSTM, ne risente di più rispetto all'architettura del fine-tuning, in quanto i Transformers sono, in generale, meno sensibili alle scelte degli iperparametri [86].

Sintetizzando, in termini di prestazioni, nessuno dei due metodi di ap-

[2] `https://github.com/musixmatchresearch/umberto#umberto-wikipedia-uncased`

	Dev set		Test set	
	μ	σ	μ	σ
LAS	93.41%	0.22	93.31%	0.27
UAS	95.24%	0.15	95.21%	0.24
LA	96.05%	0.16	95.99%	0.21

Tabella 4.4: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello appreso sul train set del corpus UD Italian ISDT 2.6.

plicazione delle rappresentazioni linguistiche è nettamente superiore all'altro rispetto a tutte le metriche analizzate.

Infine, in entrambi gli esperimenti descritti precedentemente, la differenza fra i risultati ottenuti sul test set e quelli ottenuti sul dev set è minima. Questo è, probabilmente, dovuto al fatto che i due insiemi contengono frasi provenienti dalle stesse fonti e la cui lunghezza è mediamente simile.

4.3.4 Terzo esperimento: $\text{PaT}_{\text{UmbUnc}}$

Una volta analizzati e confrontati i risultati ottenuti applicando le sole rappresentazioni linguistiche di UmBERTo Uncased al task di dependency parsing, quest'ultimo è stato sostituito a $\text{BERT}_{\text{BASE}}$ Uncased all'interno del parser PaT. Il resto dell'architettura di PaT è rimasta invariata così come la configurazione di tutti gli iperparametri, mantenendo quelli suggeriti dagli autori.

I risultati ottenuti con questa modifica sono mostrati in tabella 4.4.

Nel 2019 in [87] sono stati confrontati e riportati i risultati ottenuti da diversi dependency parser neurali delle dipendenze per l'italiano, utilizzando come treebank la UD Italian ISDT versione 2.1. Il parser risultato essere il più performante è quello di Dozat e Manning (*DMv1*) [3], allora disponibile nella sola versione 1.0. In tabella 4.5 sono messi a confronto i risultati ottenuti dal parser DMv1 con quelli ottenuti da $\text{PaT}_{\text{UmbUnc}}$.

	Dev set				Test set			
	LAS		UAS		LAS		UAS	
	μ	σ	μ	σ	μ	σ	μ	σ
DMv1	91.37%	0.24	93.37%	0.27	91.84%	0.18	93.72%	0.14
$\text{PaT}_{\text{UmbUnc}}$	**93.41%**	0.22	**95.24%**	0.15	**93.31%**	0.27	**95.21%**	0.24

Tabella 4.5: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello $\text{PaT}_{\text{UmbUnc}}$ appreso sul train set del corpus UD Italian ISDT 2.6, confrontati con media e deviazione standard dei risultati ottenuti su cinque istanze di training del parser DMv1 appreso sul train set del corpus UD Italian ISDT 2.1, come riportato in [87]. In grassetto sono evidenziati i risultati migliori.

Come si evince dalla tabella, il parser $\text{PaT}_{\text{UmbUnc}}$ ha migliorato le prestazioni del parser DMv1 [3] rispetto a tutte le metriche considerate. In particolare, si sono ottenuti dei miglioramenti pari a circa il 2% sul dev set e all'1.5% sul test set, sia per LAS che per UAS.

Dal punto di vista architetturale, va considerato che $\text{PaT}_{\text{UmbUnc}}$ è strutturalmente più semplice rispetto a qualsiasi versione del parser di Dozat e Manning, in quanto quest'ultimi fanno uso, in più moduli, del meccanismo di attention, di un character embedding ottenuto tramite una BiLSTM e di un livello MLP biaffine. Pertanto, si è deciso di rendere più complessa l'architettura di $\text{PaT}_{\text{UmbUnc}}$, con l'obiettivo di provare a migliorare ulteriormente i risultati ottenuti finora. La descrizione delle modifiche e i risultati ottenuti sono riportati nella sezione successiva.

4.3.5 Quarto esperimento: modifiche a $\text{PaT}_{\text{UmbUnc}}$

Data l'architettura di $\text{PaT}_{\text{UmbUnc}}$ e ispirandosi al parser di Dozat e Manning [3], sono state apportare due modifiche:

(setup0): Aggiunta di un character embedding architetturalmente simile a quello presente nel parser di Dozat e Manning, i cui iperparametri sono quelli suggeriti dagli autori stessi in [3];

	Dev set		Test set	
	μ	σ	μ	σ
LAS	93.45%	0.19	93.40%	0.26
UAS	95.31%	0.17	95.27%	0.29
LA	96.10%	0.21	96.04%	0.16

Tabella 4.6: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello con il *setup0* appreso sul train set del corpus UD Italian ISDT 2.6.

	Dev set		Test set	
	μ	σ	μ	σ
LAS	93.58%	0.17	93.55%	0.21
UAS	95.51%	0.14	95.47%	0.15
LA	96.09%	0.19	96.05%	0.17

Tabella 4.7: Media e deviazione standard dei risultati ottenuti su dieci istanze di training del modello con il *setup1* appreso sul train set del corpus UD Italian ISDT 2.6.

(setup1): Aggiunta di un meccanismo di self-attention nella parte ricorrente di PaT. Quest'ultima è la componente che influisce maggiormente sulle prestazioni del parser di Dozat e Manning [3].

I risultati ottenuti con questi due setup sono riportati nelle tabelle 4.6 e 4.7, rispettivamente.

Un ulteriore tentativo è stato fatto provando ad utilizzare un meccanismo di Multi-Head Self-Attention nel *setup1*, considerando il numero di teste come un iperparametro. I risultati così ottenuti dai vari test effettuati non hanno apportato miglioramenti rispetto a quelli descritti nella tabella 4.7.

4.4 Risultati e confronti

Confrontando e analizzando i risultati emersi dal terzo e quarto esperimento, si nota come il *setup1*, descritto nella sezione 4.3.5, abbia apportato dei miglioramenti, seppur modesti, al modello $\text{PaT}_{\text{UmbUnc}}$ utilizzato nel terzo esperimento. In particolare, si è ottenuto un miglioramento di circa lo 0.25% sul test set sia per la metrica LAS che UAS, mentre per la metrica LA non si è riscontrata alcuna miglioria.

Nonostante gli incrementi delle prestazioni risultati dagli esperimenti siano minimi, va considerato che per nessuno degli esperimenti descritti è stato fatto tuning degli iperparametri. Infatti, le configurazioni utilizzate sono quelle suggerite dagli autori di PaT [2], fatta eccezione per i moduli aggiunti successivamente che utilizzano la configurazione consigliata da Dozat e Manning, in quanto ispirati dal loro parser. Pertanto, potrebbe essere presente un ulteriore margine di miglioramento per tutti gli esperimenti.

Infine, per mantenere la stessa architettura descritta per PaT in [2] va tenuto presente che, negli esperimenti tre e quattro, ad UmBERTo non è stato, precedentemente, applicato un fine tuning.

Conclusioni e sviluppi futuri

In questo lavoro di tesi si è mostrato come l'utilizzo di UmBERTo, un modello derivato da BERT pre-addestrato specificatamente per l'italiano, dia un contributo essenziale al task di dependency parsing. In particolare, grazie alle sole applicazioni delle sue rappresentazioni linguistiche attraverso il fine-tuning si ottengono risultati pari all'88.79%, 91.66% e 94.89% per le metriche LAS, UAS e LA, rispettivamente, mentre attraverso il feature extraction si hanno risultati pari all'89.02%, 92.42% e 92.75% per le metriche LAS, UAS e LA, rispettivamente.

Inoltre, si è mostrato come l'applicazione del feature extraction di UmBERTo al dependency parser PaT abbia apportato alcuni miglioramenti rispetto al parser allo stato dell'arte di Dozat e Manning versione 1.0. Nello specifico, si sono ottenuti miglioramenti di circa il 2% sul dev set e all'1.5% sul test set, sia per la metrica LAS che UAS. Da questi risultati si evince, ancora una volta, la bontà di UmBERTo per quanto riguarda i task in lingua italiana.

Infine, apportando alcune modifiche al parser PaT, si è riusciti a migliorare, seppur di poco, le prestazioni per due delle tre metriche considerate. Ovvero, si sono ottenuti dei miglioramenti di circa lo 0.25% per le metriche LAS e UAS sul test set, mentre per la metrica LA non si sono riportati ulteriori incrementi.

Quanto sperimentato in questa tesi fa ben sperare per i futuri task di NLP in lingua italiana, grazie, soprattutto, ai modelli ad-hoc derivati da BERT. Per quanto concerne il task di dependency parsing, come già accennato pre-

cedentemente nella sezione 4.4, molto probabilmente, con alcune modifiche, si potrebbero riuscire a migliorare ancora i risultati fin qui ottenuti. In particolare, tutti gli esperimenti potrebbero essere estesi facendo un tuning degli iperparametri e provando a creare un'architettura neurale più profonda rispetto a quella attuale, prendendo ispirazione dal modulo encoder utilizzato nell'architettura Transformers [4].

Inoltre, potrebbe essere interessante, avendo a disposizione risorse adeguate, indagare sulle prestazioni ottenute in termini temporali per i vari esperimenti, sia in fase di train che di test.

Oltre a ciò, in questo lavoro di tesi si sono sfruttate estensivamente le rappresentazioni linguistiche ottenute da UmBERTo, ma esistono anche altri due modelli, piuttosto recenti, derivati da BERT per la lingua italiana: AlBERTo[3] [88] e GilBERTo[4] [89]. Pertanto, si potrebbero replicare alcuni esperimenti oggetto di questa tesi utilizzando gli altri due modelli linguistici, per poi analizzarne e confrontarne i risultati ottenuti, come descritto, ad esempio, in [90].

Infine, potrebbe essere interessante provare a prevedere direttamente i PoS tag tramite PaT, così da poterli, successivamente, utilizzare durante la previsione delle relazioni di dipendenza. Come mostrato in [52], i PoS tag e i DEPREL tag pare condividano diverse somiglianze semantiche, similmente a quanto accade per le parole. Pertanto, l'unione fra il task di PoS tagging e quello di dependency parsing potrebbe apportare miglioramenti ad entrambi.

Un'idea per lo sviluppo di un'architettura combinata per più task è presentata in [91], in cui viene utilizzato un singolo modello per quattro diversi task di NLP. Questi task sono accuratamente scelti in modo tale da trarre dei vantaggi l'uno dall'altro.

[3] `https://github.com/marcopoli/AlBERTo-it`

[4] `https://github.com/idb-ita/GilBERTo`

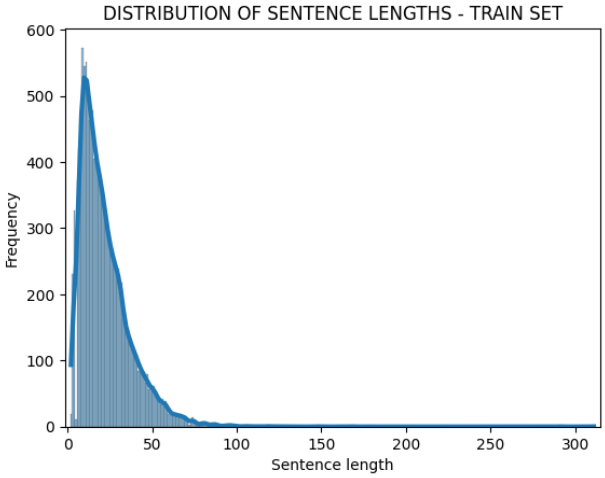
Appendice A

Statistiche UD Italian ISDT versione 2.6

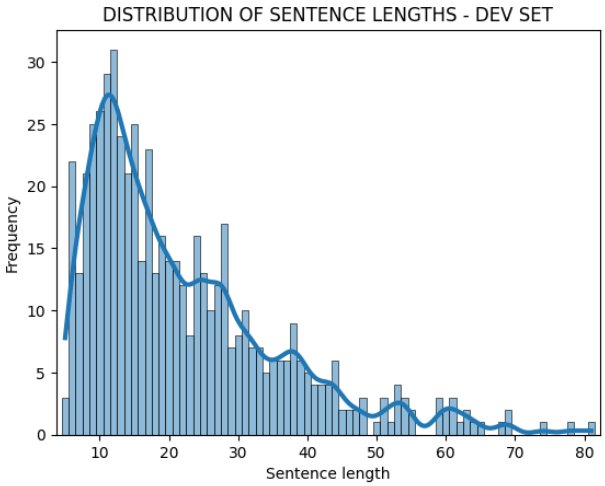
Di seguito verranno mostrate più in dettaglio alcune caratteristiche del dataset utilizzato negli esperimenti descritti nella sezione 4.3.

Deprel Tag	**Quantità**	**Deprel Tag**	**Quantità**
acl	2.945	expl:impers	425
acl:relcl	3.172	expl:pass	374
advcl	3.763	fixed	1.083
advmod	10.466	flat	570
amod	16.682	flat:foreign	145
appos	916	flat:name	3.478
aux	6.036	goeswith	1
aux:pass	2.234	iobj	693
case	41.808	mark	6.290
cc	8.158	nmod	23.940
comp	1.455	nsubj	12.820
compound	758	nsubj:pass	2.309
conj	10.121	nummod	3.559
cop	3.432	obj	10.239
csubj	314	obl	17.222
csubj:pass	2	obl:agent	1.138
dep	14	orphan	42
det	46.345	parataxis	420
det:poss	1.870	punct	33.883
det:predet	411	root	14.167
discourse	62	vocative	101
dislocated	32	xcomp	2.266
expl	2.182	**TOTALE**	**298.343**

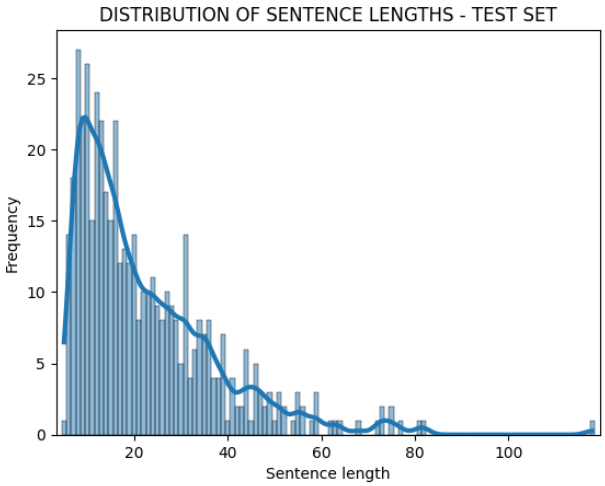
Tabella A.1: Quantità e tipologia di DEPREL tag presenti nella UD Italian ISDT versione 2.6.



(a)



(b)



(c)

Figura A.1: Frequenza delle frasi rispetto alla loro lunghezza nel train set (figura *a*), validation set (figura *b*) e test set (figura *c*) della UD Italian ISDT versione 2.6

Bibliografia

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 4171–4186, Association for Computational Linguistics, June 2019.

[2] R. Vacareanu, G. C. G. Barbosa, M. A. Valenzuela-Escárcega, and M. Surdeanu, "Parsing as tagging," in *Proceedings of The 12th Language Resources and Evaluation Conference*, pp. 5225–5231, 2020.

[3] T. Dozat and C. D. Manning, "Deep biaffine attention for neural dependency parsing," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, OpenReview.net, 2017.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, pp. 5998–6008, 2017.

[5] T. M. Mitchell, *Machine Learning*. USA: McGraw-Hill, Inc., 1 ed., 1997.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[7] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[8] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization.," *Journal of machine learning research*, vol. 12, no. 7, 2011.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[10] E. L. Thorndike, "Animal intelligence: an experimental study of the associative processes in animals.," *The Psychological Review: Monograph Supplements*, vol. 2, no. 4, p. i, 1898.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. USA: Prentice Hall Press, 3rd ed., 2009.

[13] F. Rosenblatt, *Principles of neurodynamics: perceptrons and the theory of brain mechanisms*. Washington: Spartan Books., 1962.

[14] C. M. Bishop, *Neural Networks for Pattern Recognition*. USA: Oxford University Press, Inc., 1995.

[15] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on Machine Learning* (F. Bach and D. Blei, eds.), vol. 37 of *Proceedings of Machine Learning Research*, (Lille, France), pp. 448–456, PMLR, 07–09 Jul 2015.

[16] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[17] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[18] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[19] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[20] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[21] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2013.

[22] M. Sahlgren, "The distributional hypothesis," *Italian Journal of Disability Studies*, vol. 20, pp. 33–53, 2008.

[23] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543, 2014.

[24] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[26] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," *CoRR*, vol. abs/1703.03130, 2017.

[27] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[28] M. Schuster and K. Nakajima, "Japanese and korean voice search," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5149–5152, IEEE, 2012.

[29] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[30] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[31] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, "Unsupervised cross-lingual representation learning at scale," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 8440–8451, Association for Computational Linguistics, July 2020.

[32] L. Martin, B. Muller, P. J. Ortiz Suárez, Y. Dupont, L. Romary, É. de la Clergerie, D. Seddah, and B. Sagot, "CamemBERT: a tasty French language model," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, (Online), pp. 7203–7219, Association for Computational Linguistics, July 2020.

[33] T. Kudo and J. Richardson, "SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, (Brussels, Belgium), pp. 66–71, Association for Computational Linguistics, Nov. 2018.

[34] P. J. Ortiz Suárez, B. Sagot, and L. Romary, "Asynchronous Pipeline for Processing Huge Corpora on Medium to Low Resource Infrastructures," in *7th Workshop on the Challenges in the Management of Large Corpora (CMLC-7)* (P. Bański, A. Barbaresi, H. Biber, E. Breiteneder, S. Clematide, M. Kupietz, H. Lüngen, and C. Iliadi, eds.), (Cardiff, United Kingdom), Leibniz-Institut für Deutsche Sprache, July 2019.

[35] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, (Online), pp. 38–45, Association for Computational Linguistics, Oct. 2020.

[36] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, pp. 8026–8037, 2019.

[38] L. Tesnière, "Éléments de syntaxe structurale," 1959.

[39] S. Kübler, R. McDonald, and J. Nivre, *Dependency Parsing*. Synthesis Lectures on Human La, Morgan & Claypool, 2009.

[40] J. Nivre, *Inductive dependency parsing*. Springer, 2006.

[41] J. Nivre, "An efficient algorithm for projective dependency parsing," in *Proceedings of the Eighth International Conference on Parsing Technologies*, pp. 149–160, 2003.

[42] S. P. Abney, "A computational model of human parsing," *Journal of psycholinguistic Research*, vol. 18, no. 1, pp. 129–144, 1989.

[43] A. Abeillé, "Treebanks: Building and using parsed corpora," vol. 20, Springer Science & Business Media, 2012.

[44] A. Lüdeling and M. Kytö, "Corpus linguistics. volume 1," ch. 13, Walter de Gruyter, 2008.

[45] G. Sampson, "Thoughts on two decades of drawing trees," in *Treebanks*, pp. 23–41, Springer, 2003.

[46] M. Marcus, B. Santorini, and M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," 1993.

[47] J. Nivre, M.-C. De Marneffe, F. Ginter, Y. Goldberg, J. Hajic, C. D. Manning, R. McDonald, S. Petrov, S. Pyysalo, N. Silveira, *et al.*, "Universal dependencies v1: A multilingual treebank collection," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pp. 1659–1666, 2016.

[48] M.-C. De Marneffe and C. D. Manning, "The stanford typed dependencies representation," in *Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation*, pp. 1–8, 2008.

[49] S. Buchholz and E. Marsi, "Conll-x shared task on multilingual dependency parsing," in *Proceedings of the tenth conference on computational natural language learning (CoNLL-X)*, pp. 149–164, 2006.

[50] J. Nivre, "Algorithms for deterministic incremental dependency parsing," *Computational Linguistics*, vol. 34, no. 4, pp. 513–553, 2008.

[51] R. Johansson and P. Nugues, "Investigating multilingual dependency parsing," in *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pp. 206–210, 2006.

[52] D. Chen and C. D. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 740–750, 2014.

[53] J. Nivre, "Incrementality in deterministic dependency parsing," in *Proceedings of the workshop on incremental parsing: Bringing engineering and cognition together*, pp. 50–57, 2004.

[54] S. P. Abney and M. Johnson, "Memory requirements and local ambiguities of parsing strategies," *Journal of Psycholinguistic Research*, vol. 20, no. 3, pp. 233–250, 1991.

[55] M. Kuhlmann, C. Gómez-Rodríguez, and G. Satta, "Dynamic programming algorithms for transition-based dependency parsers," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 673–682, 2011.

[56] M. Ballesteros, C. Dyer, Y. Goldberg, and N. A. Smith, "Greedy transition-based dependency parsing with stack lstms," *Computational Linguistics*, vol. 43, no. 2, pp. 311–347, 2017.

[57] Y. Goldberg and J. Nivre, "A dynamic oracle for arc-eager dependency parsing," in *Proceedings of COLING 2012*, pp. 959–976, 2012.

[58] Y. Goldberg and J. Nivre, "Training deterministic parsers with non-deterministic oracles," *Transactions of the association for Computational Linguistics*, vol. 1, pp. 403–414, 2013.

[59] R. McDonald, F. Pereira, K. Ribarov, and J. Hajic, "Non-projective dependency parsing using spanning tree algorithms," in *Proceedings of human language technology conference and conference on empirical methods in natural language processing*, pp. 523–530, 2005.

[60] Y.-J. Chu, "On the shortest arborescence of a directed graph," *Scientia Sinica*, vol. 14, pp. 1396–1400, 1965.

[61] J. Edmonds, "Optimum branchings," *Journal of Research of the national Bureau of Standards B*, vol. 71, no. 4, pp. 233–240, 1967.

[62] D. Jurafsky and J. H. Martin, "Speech and language processing (3nd edition draft)." 2020.

[63] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan, "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica*, vol. 6, no. 2, pp. 109–122, 1986.

[64] R. McDonald, K. Crammer, and F. C. Pereira, "Spanning tree methods for discriminative training of dependency parsers," *Technical Reports (CIS)*, p. 55, 2006.

[65] J. Eisenstein, "Natural language processing," 2018.

[66] J. Nivre and J. Nilsson, "Pseudo-projective dependency parsing," in *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, (Ann Arbor, Michigan), pp. 99–106, Association for Computational Linguistics, June 2005.

[67] J. Ma, Y. Zhang, and J. Zhu, "Punctuation processing for projective dependency parsing," in *Proceedings of the 52nd Annual Meeting of the*

Association for Computational Linguistics (Volume 2: Short Papers), pp. 791–796, 2014.

[68] Y. Zhang and J. Nivre, "Transition-based dependency parsing with rich non-local features," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pp. 188–193, 2011.

[69] A. Søgaard, M. de Lhoneux, and I. Augenstein, "Nightmare at test time: How punctuation prevents parsers from generalizing," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, (Brussels, Belgium), pp. 25–29, Association for Computational Linguistics, Nov. 2018.

[70] A. Lavelli, "Comparing state-of-the-art dependency parsers for the evalita 2014 dependency parsing task," *Proceedings of EVALITA*, vol. 2014, 2014.

[71] E. Kiperwasser and Y. Goldberg, "Simple and accurate dependency parsing using bidirectional lstm feature representations," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 313–327, 2016.

[72] M. Grella and S. Cangialosi, "Non-projective dependency parsing via latent heads representation (lhr)," *arXiv preprint arXiv:1802.02116*, 2018.

[73] D. Fernández-González and C. Gómez-Rodríguez, "Left-to-right dependency parsing with pointer networks," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, (Minneapolis, Minnesota), pp. 710–716, Association for Computational Linguistics, June 2019.

[74] X. Ma, Z. Hu, J. Liu, N. Peng, G. Neubig, and E. Hovy, "Stack-pointer networks for dependency parsing," in *Proceedings of the 56th Annual*

Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), (Melbourne, Australia), pp. 1403–1414, Association for Computational Linguistics, July 2018.

[75] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in neural information processing systems*, vol. 28, pp. 2692–2700, 2015.

[76] T. Ji, Y. Wu, and M. Lan, "Graph-based dependency parsing with graph neural networks," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2475–2485, 2019.

[77] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[78] D. Q. Nguyen and K. Verspoor, "An improved neural network model for joint POS tagging and dependency parsing," in *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, (Brussels, Belgium), pp. 81–91, Association for Computational Linguistics, Oct. 2018.

[79] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, JMLR Workshop and Conference Proceedings, 2010.

[80] G. Attardi, S. Saletti, and M. Simi, "Evolution of italian treebank and dependency parsing towards universal dependencies," *Proceedings of CLIC-it*, 2015.

[81] C. Bosco, V. Lombardo, D. Vassallo, and L. Lesmo, "Building a treebank for Italian: a data-driven annotation schema," in *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*, (Athens, Greece), European Language Resources Association (ELRA), May 2000.

[82] S. Montemagni, F. Barsotti, M. Battista, N. Calzolari, O. Corazzari, A. Zampolli, F. Fanciulli, M. Massetani, R. Raffaelli, R. Basili, M. T. Pazienza, D. Saracino, F. Zanzotto, N. Mana, F. Pianesi, and R. Delmonte, "The Italian syntactic-semantic treebank: Architecture, annotation, tools and evaluation," in *Proceedings of the COLING-2000 Workshop on Linguistically Interpreted Corpora*, (Centre Universitaire, Luxembourg), pp. 18–27, International Committee on Computational Linguistics, Aug. 2000.

[83] E. Bisong, "Google colaboratory," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*, pp. 59–64, Springer, 2019.

[84] N. Reimers and I. Gurevych, "Reporting score distributions makes a difference: Performance study of LSTM-networks for sequence tagging," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, (Copenhagen, Denmark), pp. 338–348, Association for Computational Linguistics, Sept. 2017.

[85] M. E. Peters, S. Ruder, and N. A. Smith, "To tune or not to tune? adapting pretrained representations to diverse tasks," in *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*, (Florence, Italy), pp. 7–14, Association for Computational Linguistics, Aug. 2019.

[86] S. Ruder, M. E. Peters, S. Swayamdipta, and T. Wolf, "Transfer learning in natural language processing," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Tutorials*, pp. 15–18, 2019.

[87] O. Antonelli and F. Tamburini, "State-of-the-art italian dependency parsers based on neural and ensemble systems," *IJCoL. Italian Journal of Computational Linguistics*, vol. 5, no. 5-1, pp. 33–55, 2019.

[88] M. Polignano, P. Basile, M. de Gemmis, G. Semeraro, and V. Basile, "AlBERTo: Italian BERT Language Understanding Model for NLP Challenging Tasks Based on Tweets," in *Proceedings of the Sixth Italian Conference on Computational Linguistics (CLiC-it 2019)*, vol. 2481, CEUR, 2019.

[89] P. J. O. Suárez, B. Sagot, and L. Romary, "Asynchronous pipeline for processing huge corpora on medium to low resource infrastructures," in *7th Workshop on the Challenges in the Management of Large Corpora (CMLC-7)*, Leibniz-Institut für Deutsche Sprache, 2019.

[90] F. Tamburini, "How "BERTology" changed the state-of-the-art also for italian nlp," in *Proceedings of the Seventh Italian Conference on Computational Linguistics (CLiC-it 2020), Bologna, Italy, March 1-3, 2021*, vol. 2769, CEUR-WS.org, 2020.

[91] V. Sanh, T. Wolf, and S. Ruder, "A hierarchical multi-task approach for learning embeddings from semantic tasks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 6949–6956, 2019.

Ringraziamenti

Colgo l'occasione per ringraziare tutte le persone che mi hanno aiutato e sostenuto durante il mio percorso universitario.

Ringrazio il mio relatore, il Professor Fabio Tamburini, per la sua grande professionalità e disponibilità.

Ringrazio i miei genitori, perché, ancora una volta, è anche grazie ai loro sacrifici se sono riuscito a raggiungere questo importante obiettivo.

Ringrazio tutti i parenti che mi hanno sostenuto durante questa indimenticabile esperienza universitaria.

Ringrazio la mia fidanzata Sonia che mi sopporta e supporta da più di quattro anni in tutte le mie scelte, riuscendo continuamente a tirare fuori la parte migliore di me. Semplicemente grazie, perché, per me, ci sei sempre stata.

Ringrazio tutta la famiglia della mia fidanzata, perché devo anche a loro, alla loro gentilezza, disponibilità e ospitalità il raggiungimento di questo traguardo.

Ringrazio tutti i miei amici di Rimini con i quali ho condiviso e condividerò le gioie e i traguardi della mia vita.

Ringrazio tutti i componenti del "FilippoTeam", un affiatato gruppo di amici informatici con cui ho condiviso tantissimi bei momenti che mi faranno ricordare con il mio solito sorriso e un pizzico di malinconia questi due splendidi anni. Una menzione particolare per Lorenzo S. con il quale mi sono confrontato più volte durante lo svolgimento di questo lavoro di tesi e che mi ha dispensato utili consigli e spunti di riflessione.

RINGRAZIAMENTI

Ringrazio Adamo, amico e compagno con cui ho condiviso lo sviluppo di diversi progetti fin dall'inizio del mio percorso universitario.