Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Studio e implementazione di un sistema ensemble per il parsing dell'italiano

Relatore:
Chiar.mo Prof.
FABIO TAMBURINI

Presentata da:
ORONZO ANTONELLI

Sessione I
Anno Accademico 2017/2018

Introduzione

La diffusione di Internet, in particolare dei social media, ha portato alla creazione di enormi quantità di contenuti da parte degli utenti attivi sulle diverse piattaforme. L'analisi sintattica del testo (parsing) dei contenuti provenienti dal dominio dei social media, in particole di Twitter, sta acquisendo sempre più importanza all'interno del settore di ricerca dell'elaborazione del linguaggio naturale.

Determinare la corretta struttura sintattica di un tweet permette di aumentare le prestazioni di specifici task, come sentiment analysis e opinion mining, che sono strettamente legati all'analisi del testo e consentono di ricavare informazioni sul contenuto. Tipicamente il contenuto sui social media rispetto, ad esempio, a web news o fonti enciclopediche è caratterizzato da: maggiori errori grammaticali; una dimensione limitata, impostata dalla piattaforma; un uso della lingua non convenzionale.

Utilizzando due corpora in lingua italiana presenti nelle Universal Dependencies, uno di dominio generico e l'altro sul dominio dei social media, si sperimenta se l'apprendimento dal corpus di dominio social media possa portare ad un significativo incremento dell'accuratezza di parsing rispetto all'apprendimento dal corpus di dominio generico. I parser presi in considerazione per condurre gli esperimenti sono otto, tutti dello stesso tipo, e considerano modelli di analisi sintattica dipendente, che verranno definiti nel Capitolo 1, con un'architettura basata su reti neurali deep.

In seguito, utilizzando i modelli di parsing appresi, si sono volute sperimentare differenti tecniche di ensemble per combinare i modelli dei singoli

parser e valutare i risultati di questi ultimi confrontandoli con il miglior parser singolo ottenuto in precedenza. Le tecniche di ensemble considerate sono: *voting*, ogni parser contribuisce alla creazione dell'albero sintattico finale esprimendo una preferenza per ogni relazione contenuta nella frase di input; *reparsing*, si utilizzano algoritmi MST (maximum spanning tree) per costruire l'albero di dipendenza finale. È stata, inoltre, sperimentata un'ulteriore tecnica basata sull'agreement, detta *distilling*, che apprende un modello di parsing ex novo a partire dalle predizioni dei singoli parser.

In base alle informazioni ricavate dalla letteratura esistente, questo è il primo lavoro che valuta modelli di parsing basati su reti neurali deep ottenuti a partire da treebank in lingua italiana, combinandoli per migliorarne le prestazioni.

Nel Capitolo 1 è definito il task di dependency parsing nell'approccio transition-based e graph-based. Sono introdotti, inoltre, lo standard e il formato Universal Dependencies e le metriche con cui è possibile valutare le prestazioni dei parser. Nel Capitolo 2 sono descritti i principali modelli di reti neurali deep sui quali si basano le architetture dei parser considerati negli esperimenti. Nel Capitolo 3 sono riportate le architetture dei parser, i due corpora usati per l'apprendimento dei modelli e i risultati delle valutazioni per i diversi esperimenti. Nel Capitolo 4 sono descritte le tecniche di ensemble e riportati i risultati delle valutazioni dei modelli combinati. Nell'Appendice A è descritto l'utilizzo di un ambiente virtuale creato ad hoc per l'esecuzione di training e testing, al fine di rendere ripetibili gli esperimenti su differenti macchine.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Dependency Parsing

In questo capitolo saranno presentate le tecniche e i formalismi dell'approccio all'analisi sintattica automatica del linguaggio naturale noto in letteratura con il nome di **dependency parsing**. Prima ancora di parlare di dependency parsing sarà brevemente discusso cosa vuol dire fare analisi sintattica e dove si colloca questo processo all'interno della vasta area di ricerca conosciuta con il nome di **natural language processing** (NLP) o elaborazione del linguaggio naturale.

In Clark et al. (2010) il termine *parsing*, nel contesto dell'elaborazione del linguaggio naturale, è definito come quel processo di analisi automatica di una data frase, vista come sequenza di parole, che permetta di determinarne la struttura sintattica sottostante. Lo strumento che esegue il *parsing* è chiamato *parser*. In breve, lo scopo di un parser è quello di ricostruire, nel modo più accurato possibile, le relazioni che intercorrono tra la parti di una frase. Prima di poter ricostruire queste relazioni, è necessario definire dei modelli che stabiliscano come queste possano essere state generate, ovvero definirne la **grammatica**. Fare assunzioni e utilizzare un particolare formalismo piuttosto che un altro determina le relazioni individuabili all'interno di una frase e la complessità del problema.

Quando si parla di **grammatica formale** si intende un insieme di regole che specifica come gli elementi del linguaggio si possono combinare per for-

mare una frase. In letteratura esistono diverse grammatiche, alcune delle più note sono: *phrase structure grammars*, *tree adjoining grammars*, *combinatory categorial grammars*. Le grammatiche alla base del dependency parsing sono le dependency grammar. Prima di introdurre e definire quest'ultimo tipo di grammatica, è utile mostrare qualche esempio per chiarire quali differenze possono esserci tra due parser che utilizzino due diverse grammatiche.

In Figura 1.1 è mostrata la struttura sintattica della frase *John loves Mary* nel caso in cui si utilizzi il *dependency parsing* o il *phrase structure parsing*.

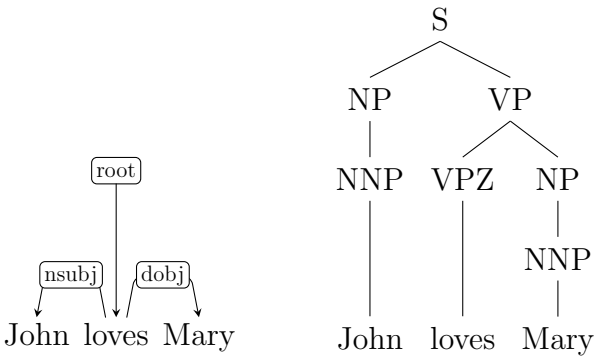


Figura 1.1: Analisi della frase *John loves Mary* tramite dependency parsing (a sinistra) e phrase structure parsing (a destra).

Nonostante la frase in analisi sia la stessa la struttura sintattica è differente. Nel caso del dependency parsing la struttura della frase è identificata costruendo relazioni binarie tra le parole, dove ciascuna è connessa all'altra tramite un arco diretto. Nel caso del phrase structure parsing la struttura che ne risulta è un albero etichettato che rappresenta la gerarchia tra gruppi di parole note come *sintagmi*.

Scegliere l'approccio da adottare stabilisce quali relazioni siamo in grado di catturare tramite la struttura sintattica. Questa scelta si ripercuote anche sui moduli che ricevono come input la struttura sintattica della frase per altri scopi (e.g., question answering e machine translation). Ciò rende il par-

sing uno dei punti centrali all'interno dei sistemi che elaborano il linguaggio naturale, dove l'accuratezza del parser impatta le varie applicazioni.

Il dependency parsing ha avuto grande successo negli ultimi anni generando sempre più interesse nella comunità scientifica. Il motivo è legato, in parte, alla realizzazione di grandi fonti di dati etichettati (*dependency treebank*) a partire dalle quali è possibile costruire parser sempre più accurati. Per questo motivo la Conference on Natural Language Learning (CoNLL) ha condotto negli anni diversi tasks relativi al dependency parsing che hanno permesso il confronto e lo sviluppo tra le diverse metodologie per il dependency parsing. In Kübler et al. (2009) sono delineati alcuni vantaggi che l'approccio dependency ha rispetto ad un approccio phrase structure. Tra questi il più importante riguarda la capacità, per le dependency grammar, di saper trattare lingue morfologicamente ricche e con un word-order più flessibile e libero. Ad esempio, in alcuni casi si potrebbe verificare che due parole in relazione tra loro, possano essere distanti nella frase. In un approccio phrase structure è difficile per l'albero riuscire a catturare la relazione precedente, questo perché la struttura ad albero gestisce termini che sono vicini tra loro; invece nell'approccio dependency-based è sufficiente connettere i due termini tramite un arco. In Jurafsky and Martin (2017), l'utilizzo di un approccio dependency-based è giustificato dal fatto che questo sia in grado di catturare tramite le relazioni sintattiche anche quelle semantiche.

1.1 Dependency Grammars

L'ipotesi alla base delle *dependency grammar* si fonda sull'idea che la struttura sintattica sia costituita da parole legate tra loro da relazioni binarie asimmetriche, che prendono il nome di **dependency relations**, o *relazioni di dipendenza*. Una relazione di dipendenza consiste di una parola subordinata detta **dependent**, o *dipendente*, e di una parola da cui questa dipende detta **head**, o *testa*. I criteri su come le relazioni di dipendenza devono essere scelte e su cosa significhino sono importanti per definire la grammatica.

Una coppia *head-dependent*, o testa-dipendente, può essere identificata con un arco unidirezionale uscente dalla *head* che termina sulla *dependent*. L'esempio di Figura 1.1 presenta due coppie testa-dipendente, ovvero *(testa) loves → John (dipendente)* e *(testa) loves → Mary (dipendente)*. Ogni coppia è accompagnata da un'etichetta che identifica e specifica quale relazione sintattica intercorre tra la testa e la dipendente, nel caso d'esempio *nsubj* e *dobj*.

Definizione 1. Una frase (o sentence) è una sequenza di parole (o token):

$$S = w_0 w_1 \dots w_n$$

dove $w_0 = \text{ROOT}$ si assume essere una parola fittizia che indica la radice della frase.

Una parola w_i può essere rappresentata o rappresentare parole composte, o un carattere di punteggiatura, a seconda della lingua considerata.

Definizione 2. Definiamo $R = \{r_1, \dots, r_m\}$ come l'insieme finito dei possibili tipi di relazione che possono esserci tra due parole in una frase.

Il tipo di relazione è l'etichetta che compare sull'arco di una relazione di dipendenza (e.g., $R = \{\text{nsubj}, \text{dobj}, \dots\}$).

Definizione 3. Un **dependency graph** $G = (V, A)$ è un grafo diretto etichettato tale che per ogni frase $S = w_0 w_1 \dots w_n$ e un insieme di etichette dei tipi di relazione R vale che:

1. $V \subseteq \{w_0, w_1, \dots, w_n\}$
2. $A \subseteq V \times R \times V$
3. se $(w_i, r, w_j) \in A$ allora $(w_i, r', w_j) \notin A$ per ogni $r' \neq r$

La tripla (w_i, r, w_j) rappresenta quella che finora abbiamo chiamato relazione di dipendenza tra la testa w_i e la dipendente w_j con etichetta r. L'ultima condizione della definizione precedente impedisce che il dependency graph possa essere un multi-grafo. Ad esempio la relazione *loves* $\xrightarrow{dobj}$ *Mary* è rappresentata dalla tripla (loves, dobj, Mary).

Definizione 4. Un **dependency tree** è un dependency graph che soddisfa le seguenti proprietà:

1. (root property) Esiste un solo nodo radice ROOT che non ha archi entranti. Formalmente, non esiste $w_i \in V$ tale che $w_i \to w_0$.

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

3. (acyclicity property) Il grafo non contiene cicli. Formalmente, se $w_i \to w_j$, allora non esiste nessun cammino per cui $w_j \to \cdots \to w_i$.

4. (arc size property) Vale che $|A| = |V| - 1$.

5. (single-head property) Ogni nodo del grafo ha esattamente un arco entrante, ovvero una parola può avere una sola testa. Formalmente, per ogni coppia di parole $w_i, w_j \in V$ se $w_i \to w_j$ allora non esiste alcuna $w_k \in V$ tale che $k \neq i$ e $w_k \to w_j$.

6. (connectedness property) Presa 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 $w_i \to \cdots \to w_j$ o $w_j \to \cdots \to w_i$.

A seconda di alcune restrizioni sulla struttura, un dependency tree può essere classificato in uno dei due tipi seguenti: *projective* o *non-projective*.

Definizione 5. Una relazione $(w_i, r, w_j) \in A$ è detta **projective** se e solo se $w_i \to \cdots \to w_k$ per ogni $i < k < j$ quando $i < j$, o $j < k < i$ quando $j < i$.

In altre parole una relazione *testa* $\to$ *dipendente* è detta projective se per ogni parola w_k che occorre nella frase tra la *testa* e la *dipendente* esiste un cammino nell'albero sintattico che connette la *testa* a w_k.

Definizione 6. Un **projective dependency tree** è un dependency tree in cui tutte le relazioni $(w_i, r, w_j) \in A$ sono projective.

Definizione 7. Un **non-projective dependency tree** è un dependency tree in cui almeno una relazione $(w_i, r, w_j) \in A$ non è projective.

In Figura 1.2 è riportato un projective dependency tree, mentre in Figura 1.3 un non-projective dependency tree. Per rappresentare un dependency tree graficamente si può, per convenzione, rendere la relazione ROOT esplicita inserendo la parola $w_0 = \text{ROOT}$ nella frase (come nelle Figure 1.2-1.3) o non riportarla anche se esiste implicitamente (come in Figura 1.1).

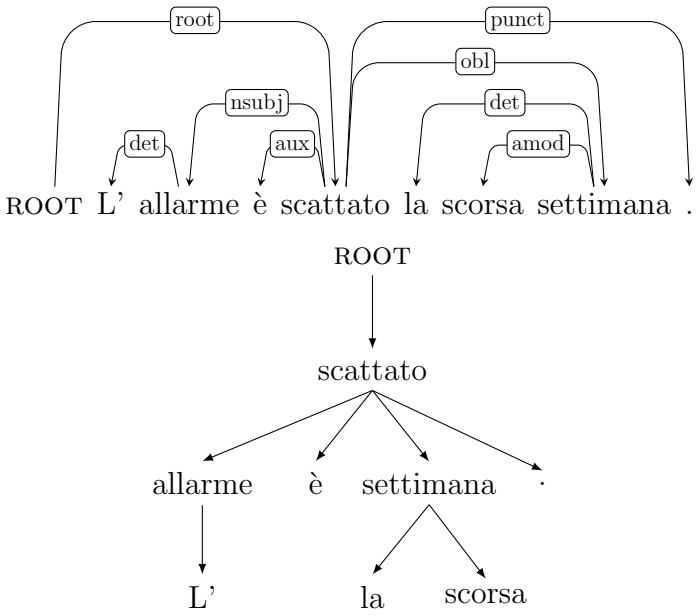


Figura 1.2: Esempio di un dependency tree projective. Nell'immagine superiore sono mostrate le relazioni sintattiche tra le parole della frase, mentre in quella inferiore è stata messa in risalto la struttura ad albero.

Tramite la rappresentazione ad albero della struttura della frase è semplice capire se un dependency tree è projective o meno. Prendiamo in considerazione la relazione *settimana* → *la* di Figura 1.2. L'insieme delle parole comprese tra *settimana* e *la* è il singoletto $W = \{scorsa\}$. È facile osservare, nella struttura ad albero (in basso), che esiste un cammino che lega la parola *settimana* alla parola *scorsa*. Si può verificare che questa proprietà

vale per ciascuna relazione testa-dipendente all'interno della frase, per cui il dependency tree di Figura 1.2 è projective.

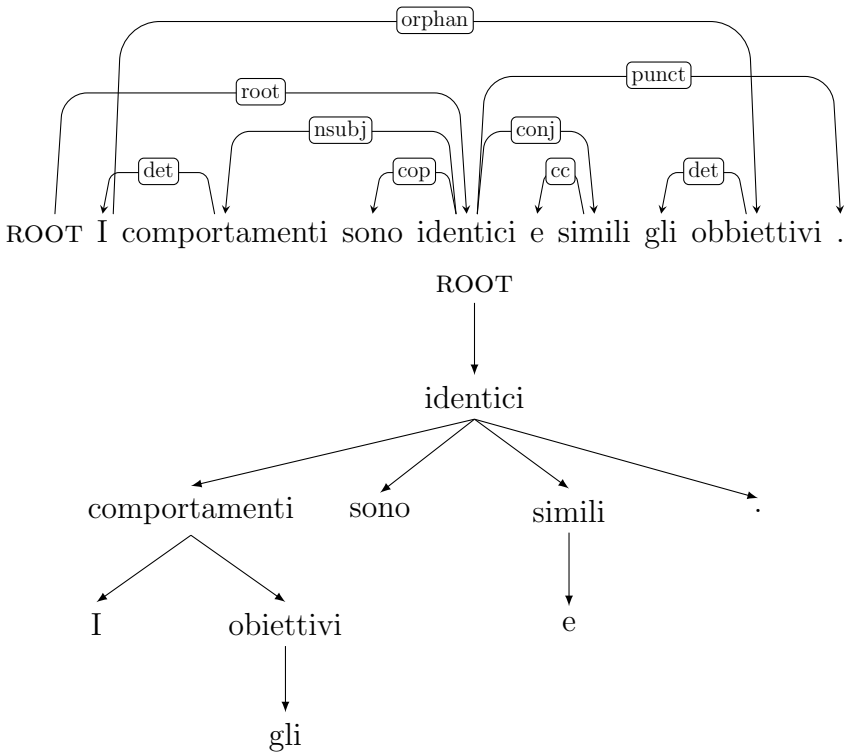


Figura 1.3: Esempio di un dependency tree non-projective. Nell'immagine superiore sono mostrate le relazioni sintattiche tra le parole della frase, mentre in quella inferiore è stata messa in risalto la struttura ad albero.

Consideriamo adesso la relazione *comportamenti* $\rightarrow$ *obiettivi* di Figura 1.3. L'insieme delle parole comprese tra *comportamenti* e *obiettivi* sono $W = \{\text{sono}, \text{identici}, \text{e}, \text{simili}, \text{gli}\}$. Si può osservare, nella struttura ad albero, come non esista alcun cammino che colleghi la testa *comportamenti* con la parola $w_k = \text{sono} \in W$. Un altro criterio per individuare se un dependency tree è projective consiste nel verificare se gli archi delle relazioni possono essere disegnati sul piano della frase senza intersecarsi con altri archi (come in Figura 1.2), in tal caso il dependency tree è projective.

1.2 Processo di Parsing

Il compito del parser, come già accennato in precedenza, è quello di produrre, a partire dalla frase in analisi, un dependency tree etichettato usando le relazioni di dipendenza. Esistono due approcci differenti per risolvere questo problema:

- **Data-driven parsing**, fa uso di tecniche di machine learning per apprendere il modello di parsing;
- **Grammar-based parsing**, è basato sulla definizione di grammatiche e linguaggi formali. Data una frase, il parser si occupa di capire se questa può appartenere al linguaggio definito da una grammatica.

L'approccio seguito in questo lavoro esplora la prima categoria di parser, in particolare si utilizzeranno metodi di apprendimento supervisionati, i quali presuppongono che le frasi siano fornite al parser già annotate con la struttura sintattica corretta. Nel dependency parsing supervisionato si affrontano due problemi:

- **Learning**: Dato un insieme $\mathcal{D}$ di frasi annotate sintatticamente come dependency tree, detto training set, ricavare un modello M che può essere utilizzato per fare il parsing di nuove frasi.
- **Parsing**: Dato un modello M e una frase S, stabilire il miglior dependency tree T per S a partire dal modello M.

Nel seguito saranno discussi i due metodi più utilizzati nel dependency parsing: **transition-based** (Sezione 1.4) e **graph-based** (Sezione 1.5).

1.3 Dependency Treebank

Come discusso nella Sezione 1.2, nell'approccio data-driven per costruire un modello di parsing M occorre partire da un insieme $\mathcal{D}$ di frasi annotate sintatticamente; questo insieme è noto come *treebank*.

Definizione 8. Un **treebank** è un insieme di N coppie $(S^{(i)}, T^{(i)})$, dove $S^{(i)}$ è una frase e $T^{(i)}$ il relativo dependency tree annotato:

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

Come riportato in Clark et al. (2010), avere a disposizione un treebank con annotazioni di buona qualità e un considerevole numero di frasi permette di costruire un modello di parsing con buone performance. Chi annota un treebank è di solito un gruppo di esperti linguisti che concordano sulla struttura sintattica da assegnare ad una particolare frase in modo che chi utilizzi il treebank possa avere uno standard corretto di riferimento, noto anche come **gold standard**. A volte invece, data la mole eccessiva dei dati da etichettare, si preferisce un approccio semi-automatico dove dapprima si procede annotando automaticamente, tramite uno strumento software, il treebank e successivamente si procede alla correzione manuale. In base al tipo di parsing che stiamo considerando possiamo aver bisogno di una tipologia di treebank piuttosto che un'altra (e.g., phrase-structure treebank, dependency treebank).

Adottare un treebank comune per l'apprendimento di diversi parser è desiderabile poiché rende i vari modelli appresi confrontabili in quanto costruiti a partire dalla stessa fonte. Apprendere da una fonte comune e standardizzata permette di classificare in base alla valutazione delle prestazioni i migliori modelli costruiti a partire dagli stessi dati. Questa è l'idea alla base degli **shared task**, ovvero competizioni che permettono di valutare i migliori modelli costruiti a partire da uno stesso treebank.

Prima di annotare i testi, questi devono essere raccolti da fonti esterne e tokenizzati. Le fonti dalle quali costruire un treebank possono essere differenti (e.g., testi, internet, audio) ed è buona norma specificare anche il tema trattato nelle fonti da cui sono stati estratti i testi. Questo è utile poiché il gergo del testo può essere più o meno legato ad un particolare argomento. Ad esempio, costruire un treebank con una raccolta di testi presi solo da riviste specialistiche del settore medico rende il modello appreso più adatto a task che riguardano il parsing di frasi con lo stesso gergo medico. Di solito, le

fonti preferite per la costruzione di un treebank generico non hanno carattere troppo specialistico e sono di solito articoli giornalistici (e.g., news) o voci enciclopediche (e.g., Wikipedia).

Il primo treebank ampiamente utilizzato, inizialmente per l'inglese, è stato il Penn Treebank (PTB), descritto in Marcus et al. (1993), basato sulle grammatiche phrase-structure. Il motivo della sua popolarità è legato alla grande quantità di testi annotati (circa 2 milioni di parole) presenti nel treebank. Per quanto riguarda i dependency treebank, i vari tentativi di cercare uno standard comune negli anni hanno portato alla realizzazione del recente progetto **Universal Dependencies** (UD) di Nivre et al. (2016), che al momento in cui si scrive è giunto versione 2.1 e comprende una collezione di oltre 100 treebank in più di 60 lingue[1]. Le Universal Dependencies nascono con l'obiettivo di fornire una risorsa condivisa comune e un framework in grado di facilitare e standardizzare il processo di annotazione per più lingue. Il fine è quello di permettere l'apprendimento di modelli di parsing multi-lingua e garantire valutazioni che siano confrontabili nei vari task ed esperimenti basati sul parsing. Le UD nascono dalla fusione di altre iniziative separate, tra cui le Stanford dependencies in de Marneffe and Manning (2008) e de Marneffe et al. (2014), da cui hanno ereditato alcuni schemi di dipendenza con l'intento di definire un nuovo standard de facto.

Il formato dei file annotati adottato dalle UD è il **CoNLL-U**, quest'ultimo a sua volta è un'estensione del formato **CoNLL-X** adottato durante il CoNLL-X Shared Task proposto in Buchholz and Marsi (2006). Il formato CoNLL-X è stato creato per poter organizzare in uno standard comune i diversi treebank nelle 13 lingue adottate durante il task. Il formato CoNLL-X stabilisce che le frasi debbano essere memorizzate in un file di testo in chiaro con codifica unicode (UTF-8) e separate tra loro da un carattere di linea vuota. Ciascuna frase è costituita da uno o più token (o parole), uno su ogni riga, e ciascun token è formato da 10 attributi separati tra loro da un carattere di tabulazione. Gli attributi per ciascun token sono:

[1]Il progetto è disponibile al link `http://universaldependencies.org/`

1. **ID**: Contatore token, comincia da 1 per ogni nuova frase.
2. **FORM**: Parola della frase o simbolo di punteggiatura.
3. **LEMMA**: Lemma o radice della parola.
4. **CPOSTAG**: Part-of-speech tag generico, detto coarse-grained;
5. **POSTAG**: Part-of-speech tag specifico, detto fine-grained. Se non disponibile è identico al campo precedente.
6. **FEATS**: Insieme non ordinato delle caratteristiche sintattiche e/o morfologiche; underscore se non disponibili. Gli elementi sono separati tra loro dal carattere '—' (barra verticale).
7. **HEAD**: La testa della parola corrente. Può essere o il valore ID di un altro token o zero ('0') se il token è la radice della frase.
8. **DEPREL**: Etichetta della relazione dependency relativa alla testa. Se HEAD=0, la relazione può essere o un'etichetta significativa o avere il valore di default ROOT.
9. **PHEAD**: Testa projective del token corrente, può essere o il valore di un ID o zero ('0'), o un underscore se non disponibile. La struttura ottenuta da PHEAD è garantita essere projective.
10. **PDEPREL**: Relazione dependency relativa al PHEAD, o un underscore se non disponibile.

A partire dal CoNLL-X è stato definito il formato CoNLL-U, che ridefinisce alcuni campi. Nel formato CoNLL-U le frasi sono memorizzate all'interno di un file di testo in chiaro con codifica unicode (UTF-8), come nel caso del formato CoNLL-X. L'interruzione di linea tra una frase l'altra è espressa dal carattere LF, lo stesso vale per la fine del file. Come per il formato CoNLL-X ogni token occupa una linea del file di testo e ciascuno è composto da 10 attributi separati tra loro da un carattere di tabulazione TAB. Il formato

CoNLL-U, rispetto al CoNLL-X, dà la possibilità di creare commenti anteponendo il carattere hash '#' all'inizio della linea. Gli attributi per i token nel formato CoNLL-U sono:

1. **ID**: Contatore token, comincia da 1 per ogni nuova frase. Può essere espresso come intervallo per rappresentare token composti da più parole; può essere un numero decimale per nodi vuoti.

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

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

4. **UPOSTAG**: Part-of-speech tag generico definito negli *universal part-of-speech tags* delle lingue guida delle UD. I tag sono riportati in Tabella 1.1.

5. **XPOSTAG**: Part-of-speech tag specifico per la lingua; underscore se non disponibile.

6. **FEATS**: Lista non ordinata delle caratteristiche morfologiche prese dalle *universal feature* o da un'estensione nelle *language-specific extension* definite nelle linee guida delle UD; underscore se non disponibile. Le feature sono riportate in Tabella 1.2.

7. **HEAD**: Testa della parola corrente, può essere o il valore di un ID o zero (0).

8. **DEPREL**: Etichetta della relazione dependency relativa alla testa e presa dalle *universal dependency relations*, definite nelle linee guida UD, o un sottotipo specifico per la lingua. La parola è ROOT se e solo se HEAD=0. Le relazioni sono riportate in Tabella 1.3.

9. **DEPS**: Dependency graph migliorato espresso sotto forma di un elenco di coppie HEAD-DEPREL. La linee guida per la rappresentazione *enhanced* permettono di rendere esplicite alcune relazioni implicite in modo da semplificare la disambiguazione.

10. **MISC**: Qualunque 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 nuovo formato devono rispettare i seguenti requisiti:

- I campi non devono essere vuoti.
- I campi diversi da FORM e LEMMA non possono contenere caratteri di spazio.
- Il carattere underscore '_' è usato per indicare i valori non specificati in tutti i campi ad eccezione del campo ID. Inoltre, nei treebank UD i campi UPOSTAG, HEAD, e DEPREL non posso essere lasciati senza specifica.

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 1.1: Etichette adottate come standard per il POS tags nelle Universal Dependencies v2.

La differenza sostanziale tra i due formati è che nel caso del CoNLL-X le etichette delle annotazioni dipendevano unicamente da quelle dei treebank originari e potevano essere diverse da lingua a lingua. Nel caso del formato

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

Tabella 1.2: Lista delle caratteristiche morfologiche e sintattiche definite nelle Universal Dependencies v2. Sono utilizzate per distinguere proprietà grammaticali e lessicali aggiuntive non catturate dalle etichette del POS tags.

CoNLL-U, invece, i campi UPOSTAG, FEATS e DEPREL sono stabiliti e fissati nelle linee guida[2] per la creazione dei treebank.

Per codificare un dependency tree in uno dei due formati CoNLL potrebbero bastare solamente i campi ID, FORM, HEAD e DEPREL. Il motivo per cui ad ogni token sono associati altri attributi è che questi sono utili, in fase di learning, per poter apprendere i modelli di parsing. Il part-of-speech (POS) tag rappresenta la categoria lessicale della parola che può essere etichettata come aggettivo (ADJ), avverbio (ADV) o in una delle altre categorie riportate in Tabella 1.1. Le caratteristiche morfologiche e sintattiche sono raccolte in una lista di coppie attributo-valore che servono ad identificare la forma verbale (VerbForm), il genere (Gender) e le altre caratteristiche riportate in Tabella 1.2.

[2]Le linee guida complete e dettagliate sono disponibili al link `http://universaldependencies.org/guidelines.html`.

	Nominals	**Clauses**	**Modifier**	**Function**
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 1.3: La tabella contiene i 37 tipi di relazioni sintattiche universali usate per etichettare i tipi di relazioni di dipendenza nelle Universal Dependencies v2. Nella parte superiore della tabella le righe corrispondono a categorie funzionali in relazione alla testa, mentre le colonne a categorie strutturali della dipendente. Nella parte inferiore della tabella sono espresse relazioni che non sono relazioni di dipendenza in senso stretto (il termine MWE sta per multiword expressions).

In Figura 1.4 è riportata la codifica di un dependency tree sia in formato CoNLL-U che CoNLL-X. Consideriamo la parola *deve* presente nel file `sentence.conllu`. Dagli attributi annotati si può stabilire che *deve* è un ausiliare (*AUX*) nella forma presente indicativo (*Tense=Pres*, *Mood=Ind*) terza persona singolare (*Person=3*, *Number=Sing*) del verbo (*VerbForm=Fin*) *dovere*. Inoltre, la parola *deve* è connessa dalla relazione di dipendenza con etichetta *det* al token con ID=4, cioè *avere*, che è la testa della relazione.

sentence.conllu

```
# text = Il Governo deve avere la fiducia delle due Camere.
1 Il il DET RD Definite=Def|Gender=Masc|Number=Sing|PronType=Art 2 det _ _
2 Governo governo NOUN S Gender=Masc|Number=Sing 4 nsubj _ _
3 deve dovere AUX VM Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin 4 aux _ _
4 avere avere VERB V VerbForm=Inf 0 root _ _
5 la il DET RD Definite=Def|Gender=Fem|Number=Sing|PronType=Art 6 det _ _
6 fiducia fiducia NOUN S Gender=Fem|Number=Sing 4 obj _ _
7-8 delle _ _ _ _ _ _ _ _
7 di di ADP E _ 10 case _ _
8 le il DET RD Definite=Def|Gender=Fem|Number=Plur|PronType=Art 10 det _ _
9 due due NUM N NumType=Card 10 nummod _ _
10 Camere camera NOUN S Gender=Fem|Number=Plur 6 nmod _ SpaceAfter=No
11 . . PUNCT FS _ 4 punct _ _
```

sentence.conllx

```
1 Il il DET DET_RD Definite=Def|Gender=Masc|Number=Sing|PronType=Art 2 det _ _
2 Governo governo NOUN NOUN_S Gender=Masc|Number=Sing 4 nsubj _ _
3 deve dovere AUX AUX_VM Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin 4 aux _ _
4 avere avere VERB VERB_V VerbForm=Inf 0 root _ _
5 la il DET DET_RD Definite=Def|Gender=Fem|Number=Sing|PronType=Art 6 det _ _
6 fiducia fiducia NOUN NOUN_S Gender=Fem|Number=Sing 4 obj _ _
7 di di ADP ADP_E _ 10 case _ _
8 le il DET DET_RD Definite=Def|Gender=Fem|Number=Plur|PronType=Art 10 det _ _
9 due due NUM NUM_N NumType=Card 10 nummod _ _
10 Camere camera NOUN NOUN_S Gender=Fem|Number=Plur 6 nmod _ _
11 . . PUNCT PUNCT_FS _ 4 punct _ _
```

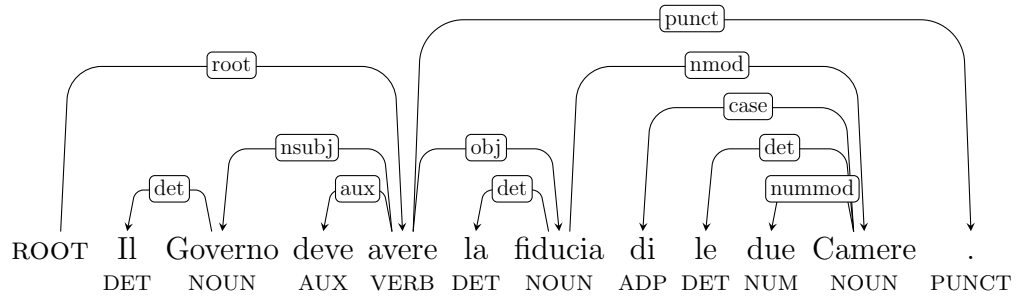


Figura 1.4: Codifica di un dependency tree (sotto) sia nel formato CoNLL-U (sopra) che nel formato CoNLL-X (centro). Per motivi di impaginazione i caratteri di tabulazione sono stati sostituiti da uno spazio.

Per individuare alcune differenze tra i due formati analizziamo l'esempio di Figura 1.4. Nel file con formato CoNLL-U la parola *della* è riportata utilizzando l'intervallo degli ID delle due parole da cui è composta, ovvero *di* e *la*. Questo nel formato CoNLL-X non è permesso, per cui sono riportati solo i due token di e *la* senza la parola composta *della*. Se consideriamo la parola *camera*, l'attributo MISC del formato CoNLL-U, riporta il valore `SpaceAfter=No` che sta ad indicare che nel testo originale il token non era seguito da un carattere di spazio, questo non è riportato nel formato CoNLL-X.

Poiché il formato CoNLL-U è un'estensione del formato CoNLL-X, nella conversione dal primo al secondo si potrebbe avere una perdita di informazione per alcuni campi, in particolare negli ultimi due che sono totalmente ridefiniti, ma questo non è un grande problema se i parser durante l'apprendimento non fanno uso delle informazioni di questi ultimi due campi.

1.4 Transition-Based Dependency Parsing

L'approccio **transition-based** dependency parsing introduce la nozione di sistema di transizione per apprendere il modello M di parsing da cui è possibile ricavare un dependency tree T data una frase S. Il problema dell'apprendimento consiste nella costruzione di un modello M che sappia predire il successivo stato di transizione data la storia delle precedenti transizioni. Il parsing, invece, a partire dal modello M costruisce la sequenza di transizioni ottimale per una data frase S, restituendo il dependency tree T. A volte ci si riferisce a questo metodo anche con il termine *shift-reduce dependency parsing*, dal momento che l'intero approccio è ispirato dal parsing deterministico shift-reduce delle grammatiche context-free.

Definizione 9. Un **transition system**, o *sistema di transizioni*, è una macchina astratta composta da un insieme $\mathcal{C}$ di **configurazioni**, o stati, e di un insieme $\mathcal{T}$ di transizioni applicabili alle configurazioni.

Definizione 10. Dato un insieme R di tipi di dipendenza e un insieme di parole $V = \{w_0, w_1, \ldots, w_n\}$, una **configurazione** per una frase $S = w_0 w_1 \ldots w_n$ è una tripla $c = (\sigma, \beta, A) \in \mathcal{C}$, dove

1. σ è lo **stack** delle parole $w_i \in V$;

2. β è il **buffer** delle parole $w_i \in V$;

3. A è l'insieme delle relazioni di dipendenza $(w_i, r, w_j) \in V \times R \times V$.

Una configurazione rappresenta l'analisi parziale di una frase: le parole nello stack σ sono parzialmente analizzate, quelle nel buffer β sono le restanti da analizzare nella frase di input e l'insieme A rappresenta la costruzione parziale del dependency tree.

Definizione 11. Per ogni frase $S = w_0 w_1 \ldots w_n$,

- la *configurazione iniziale* è $([w_o]_\sigma, [w_1, \ldots, w_n]_\beta, \emptyset)$.

- la *configurazione finale* è nella forma $(\sigma, [\,]_\beta, A)$ per qualunque σ e A.

Definiamo ora cosa è una transizione. Intuitivamente, una transizione è un'azione che modifica uno degli elementi della configurazione, cioè lo stack σ, il buffer β o l'albero parziale A.

Definizione 12. Una transizione $t \in \mathcal{T}$ è una funzione parziale $t : \mathcal{C} \to \mathcal{C}$ che mappa una data configurazione $c \in \mathcal{C}$ in una nuova configurazione valida $c' \in \mathcal{C}$.

Data una frase S e una configurazione iniziale c_0 costruita a partire da S, il parser applica una sequenza di m transizioni valide in modo da poter raggiungere una configurazione finale $c_m = (\sigma_m, []_\beta, A_m)$ dove A_m è il dependency tree (ben formato) della frase S. Analizziamo in dettaglio come avviene il parsing nel caso di sistemi transition-based.

1.4.1 Parsing

Dato un insieme non vuoto $\mathcal{T}$ di transizioni, per eseguire il parsing deterministico di una frase S occorre determinare per ogni configurazione c non finale qual è, tra le varie transizioni $t \in \mathcal{T}$, quella da applicare a c.

Ipotizziamo di avere a disposizione un oracolo ORACLE che, data una configurazione c, restituisce la transizione t corretta per c. Nella pratica, però, ciò non è possibile e quello che riusciamo a fare è approssimare l'oracolo con una funzione $\text{ORACLE}_{\approx}$, idealmente vorremmo che $\text{ORACLE}_{\approx}(c) = \text{ORACLE}(c)$. In Figura 1.5 è riportato l'algoritmo di parsing deterministico ottenuto usando l'oracolo approssimato.

function PARSE($S = w_0 w_1 \dots w_n$)
  $c \leftarrow ([w_0], [w_1, \dots, w_n], \emptyset)$
  while $c \neq (\sigma, [\,]_\beta, A)$ **do**
    $t \leftarrow \text{ORACLE}_{\approx}(c)$
    $c \leftarrow t(c)$
  return c

Figura 1.5: Algoritmo greedy per il parsing deterministico di un generico transition-based dependency parser. L'algoritmo restituisce la configurazione finale che contiene il dependency tree per la frase S

La complessità dell'algoritmo di parsing deterministico per un parser transition-based è $O(n)$, dove n è il numero di parole nella frase S passata come input, supposto che la funzione ORACLE e la transizione t siano calcolabili in tempo costante. La configurazione finale c restituita dall'algoritmo conterrà le relazioni di dipendenza A del dependency tree predetto per la frase S.

Una variazione dell'algoritmo di parsing deterministico consiste nel rilassare l'assunzione di determinismo ed esplorare più di una sequenza di transizione ad ogni passo. Come suggerito in Johansson and Nugues (2006), è possibile utilizzare una **beam search** che memorizza le k sequenze del-

le transizioni più promettenti ad ogni passo. Invece di scegliere la miglior transizione, ad ogni passo si applicano tutte le transizioni valide a tutti gli stati contenuti in un'*agenda*, che contiene inizialmente solo la configurazione iniziale. Le nuove configurazioni ottenute dall'applicazione delle transizioni valide sono, a loro volta, aggiunte all'*agenda* che può, però, contenere solo un numero limitato di configurazioni; l'ampiezza massima della *agenda* è detta **beam width**. Quando l'*agenda* raggiunge il limite massimo vengono aggiunte solo le configurazioni migliori di quelle già presenti nell'agenda e rimosse le altre. La ricerca continuerà finché all'interno dell'*agenda* compare almeno uno stato non-finale. L'algoritmo è riportato in Figura 1.6.

function BEAMPARSE($S = w_0 w_1 \dots w_n$, *width*)
  $c_0 \leftarrow ([w_0], [w_1, \dots, w_n], \emptyset)$
  $agenda \leftarrow \{c_0\}$
  while $\exists\, c \in agenda$ tale che $c \neq (\sigma, [\,]_\beta, A)$ **do**
    $newagenda \leftarrow \{\}$
    for each $c \in agenda$ **do**
      for each $t \in$ TRANSIZIONIVALIDE(c) **do**
        $newagenda \leftarrow newagenda \cup \{t(c)\}$
    $agenda \leftarrow$ TOP($width, newagenda$)
  return TOP($1, agenda$)

Figura 1.6: Algoritmo di parsing che integra una beam search con ampiezza *width* per un generico transition-based dependency parser. La funzione TOP restituisce i *width* stati presenti nell'agenda con *score* più alto. L'algoritmo restituisce la configurazione con *score* massimo presente nell'agenda.

L'approccio con beam search richiede anche una nozione di punteggio, in cui una funzione *score* assegna a ciascuna configurazione un punteggio che ne valuta la bontà e che permette di tenere le migliori e scartare le peggiori. Formulando in altri termini, il problema di parsing consiste nello scegliere la

miglior transizione $\hat{t}$ che massimizzi lo *score*:

$$\hat{t} = \arg\max_{t \in T} \text{score}(c, t) \tag{1.1}$$

Se la beam width è 1 ci troviamo nel caso **greedy** descritto nell'algoritmo di Figura 1.5, in cui viene applicata solo la miglior transizione ad ogni passo.

1.4.2 Learning

Il problema dell'apprendimento consiste nella costruzione di una funzione che approssimi ORACLE. Per poter approssimare un oracolo i parser allo stato dell'arte utilizzano metodi di machine learning supervisionati che permettono di apprendere un *classificatore* a partire da un treebank. Possiamo immaginare l'oracolo come una tabella che contiene la giusta transizione per ogni possibile configurazione $c \in \mathcal{C}$. Ad ogni passo basterebbe scegliere la riga corrispondente a c e prendere la transazione valida associata.

Nella pratica, per rendere il problema trattabile, la tabella è rappresentata in maniera più compatta tramite un vettore di *features*. Ciò che vogliamo è costruire un classificatore che predica la transizione t per una configurazione c a partire dalla rappresentazione in feature $\mathbf{f}(c)$ della configurazione.

Definizione 13. Data un configurazione c, una *feature* $\mathbf{f}_i(c)$ è costituita da:

- 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 funzione che data la parola $w_i \in V$ precedente accede ad un determinato attributo presente nel treebank (e.g., DEPREL, FEATS, UPOSTAG) e ne restituisce il valore.

Possiamo identificare una feature $\mathbf{f}_i(c)$ tramite l'elemento d'interesse nella configurazione (σ per lo stack e β per il buffer) inserendo a pedice la posizione a cui accedere e ad apice l'attributo della parola. Ad esempio, la feature $\beta_1^{\text{DEPREL}}(c)$ accede all'attributo DEPREL della prima parola contenuta nel buffer della configurazione c.

Consideriamo una possibile sequenza di configurazioni a partire dalla frase dell'esempio di Figura 1.4:

$$
\begin{aligned}
c_0 = (&[\text{ROOT}]_\sigma, \\
&[\text{Il}, \text{Governo}, \text{deve}, \text{avere}, \text{la}, \text{fiducia}, \text{di}, \text{le}, \text{due}, \text{Camere}]_\beta, \\
&\emptyset_A) \\
c_1 = (&[\text{ROOT}, \text{Il}]_\sigma, \\
&[\text{Governo}, \text{deve}, \text{avere}, \text{la}, \text{fiducia}, \text{di}, \text{le}, \text{due}, \text{Camere}]_\beta, \\
&\emptyset_A) \\
c_2 = (&[\text{ROOT}]_\sigma, \\
&[\text{Governo}, \text{deve}, \text{avere}, \text{la}, \text{fiducia}, \text{di}, \text{le}, \text{due}, \text{Camere}]_\beta, \\
&\{(\text{Governo}, \text{det}, \text{Il})\}_A) \\
&\vdots
\end{aligned}
\tag{1.2}
$$

La feature $\sigma_1^{\text{UPOSTAG}}(c_1)$ restituirà il valore dell'attributo UPOSTAG della prima parola in cima allo stack σ della configurazione c_1, dunque si ha che $\sigma_1^{\text{UPOSTAG}}(c_1) = \text{DET}$. Questa tipologia di feature è *statica*, il valore non cambia durante il parsing poiché fa riferimento ad attributi statici annotati nel treebank di riferimento. È possibile usare feature *dinamiche* che variano in base alle relazioni di dipendenza parzialmente predette dell'insieme A della configurazione. Per poter usare feature dinamiche è necessario introdurre funzioni per visitare l'albero delle relazioni di dipendenza parzialmente predette per una data parola. Ad esempio, la feature $\beta_{LC(1)}^{\text{UPOSTAG}}$ fa uso della funzione LC (leftmost child) che accede al figlio più a sinistra. Nel nostro esempio $\beta_{LC(1)}^{\text{UPOSTAG}}(c_2) = \text{DET}$ perché il figlio più a sinistra della parola *Governo* (prima parola del buffer β), nell'insieme delle relazioni parzialmente predette A, è la parola *Il*, il cui attributo UPOSTAG è DET. È anche possibile concatenare più attributi per una stessa parola o comporre due o più feature per ottenere feature a più parole. Ad esempio la feature *composta* $\sigma_1^{\text{UPOSTAG}} \circ \beta_1^{\text{FORM}}(c_1)$ concatena il risultato ottenuto dalla prima feature a

quello della seconda:

$$\sigma_1^{\text{UPOSTAG}} \circ \beta_1^{\text{FORM}}(c_1) = \sigma_1^{\text{UPOSTAG}}(c_1) \;||\; \beta_2^{\text{FORM}}(c_1) = \text{DET}||\textit{dovere}.$$

Definizione 14. La *rappresentazione in feature* di una configurazione c è una funzione $\mathbf{f} : \mathcal{C} \rightarrow \mathcal{F}^m$ che mappa la configurazione in un vettore m-dimensionale composto da singole *feature* $\mathbf{f}_i(c)$, per $1 \leq i \leq m$:

$$\mathbf{f}(c) = \langle \mathbf{f}_1(c), \mathbf{f}_2(c), \ldots, \mathbf{f}_m(c) \rangle$$

Mettendo assieme diverse feature $\mathbf{f}_i$ possiamo costruire un template $\mathbf{f}$ per una generica configurazione c che estrae la rappresentazione in feature $\mathbf{f}(c)$. Consideriamo il template

$$\mathbf{f}(c) = \langle \sigma_1^{\text{FORM}}(c),\ \beta_1^{\text{UPOSTAG}}(c),\ \beta_{LC(1)}^{\text{DEPREL}}(c),\ \sigma_1^{\text{UPOSTAG}} \circ \beta_2^{\text{LEMMA}}(c) \rangle$$

Se applichiamo $\mathbf{f}(c)$ alla sequenza di configurazioni (1.2) otteniamo:

$$\begin{array}{lllll} \mathbf{f}(c_0) = \langle & \textsc{root}, & \textsc{Det}, & \text{null}, & \text{null}||\text{governo} \rangle \\ \mathbf{f}(c_1) = \langle & \text{Il}, & \textsc{Noun}, & \text{null}, & \textsc{Det}||\text{dovere} \rangle \\ \mathbf{f}(c_2) = \langle & \textsc{Root}, & \textsc{Noun}, & \text{det}, & \text{null}||\text{dovere} \rangle \\ \vdots & & & & \end{array} \tag{1.3}$$

La scelta di quali feature rappresentare è stabilita prima della fase di apprendimento, ma con le recenti tecniche di deep learning le feature possono essere direttamente apprese dall'algoritmo stesso (Capitolo 2). La rappresentazione in feature è alla base della costruzione del modello di parsing. Ciò che vogliamo è apprendere una funzione che mappi la rappresentazione in feature $\mathbf{f}(c_i)$ per ciascuna configurazione c_i, con la relativa transizione t_i. Questo è un problema di *classificazione* dove le istanze da classificare sono le rappresentazioni in feature delle configurazioni e la classe da predire le possibili transizioni. Il classificatore è usato come approssimazione della funzione Oracle. Un'istanza di training per il classificatore sarà nella forma $(\mathbf{f}(c), t)$, dove assumiamo che $t = \textsc{Oracle}(c)$.

Poiché i treebank sono nella forma

$$\mathcal{D} = \{(S^{(i)}, T^{(i)})\}_{i=1}^{N} \tag{1.4}$$

è necessario convertirli nella forma $\mathcal{D}^{(\text{train})} = \{(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)})\}$ dove la k-esima istanza di training $\boldsymbol{x}^{(k)}$ è la rappresentazione in feature di una configurazione e la classe da predire $\boldsymbol{y}^{(k)}$ è la relativa transizione corretta:

$$\mathcal{D}^{(\text{train})} = \{(\mathbf{f}(c_j), t_j) \mid t_j = \textsc{Oracle}(c_j)\} \tag{1.5}$$

Per ogni frase $S^{(i)}$ nel gold standard treebank costruiamo la configurazione iniziale e, a partire da questa, applichiamo ad ogni passo la transizione t_j corretta ottenuta dall'oracolo. A partire dalla configurazione c_j ricaviamo la relativa rappresentazione in feature $\mathbf{f}(c_j)$ e aggiungiamo la coppia $(\mathbf{f}(c_j), t_j)$ al training set $\mathcal{D}^{(\text{train})}$. L'algoritmo è riportato in Figura 1.7.

function TRAININGORACLE($\mathcal{D} = \{(S^{(1)}, T^{(1)}), \ldots, (S^{(N)}, T^{(N)})\}$)
  $\mathcal{D}^{(\text{train})} \leftarrow \{\}$
  for each $(S = w_0 w_1 \ldots w_n,\ T) \in \mathcal{D}$ **do**
    $c_0 \leftarrow ([w_0], [w_1, \ldots, w_n], \emptyset)$
    $j \leftarrow 0$
    while $c_j \neq (\sigma, [\,]_\beta, A)$ **do**
      $t_j \leftarrow \textsc{Oracle}(c_j)$
      $c_{j+1} \leftarrow t_j(c_j)$
      $\mathcal{D}^{(\text{train})} \leftarrow \mathcal{D}^{(\text{train})} \cup \{(\mathbf{f}(c_j), t_j)\}$
      $j \leftarrow j + 1$
  return $\mathcal{D}^{(\text{train})}$

Figura 1.7: Algoritmo per costruire il training set $\mathcal{D}^{(\text{train})}$ per un generico transition-based dependency parser a partire da un gold treebank $\mathcal{D}$. La transizione restituita da $\textsc{Oracle}(c)$ è ottenibile a partire dal sistema di transizione adottato.

Nella fase di learning cerchiamo di approssimare la funzione ORACLE apprendendo un classificatore:

$$\hat{t} = \textsc{Oracle}_{\approx}(c) = \arg\max_{t \in \mathcal{T}} \boldsymbol{w} \cdot \text{score}(\mathbf{f}(c), t) \tag{1.6}$$

dove $\boldsymbol{w}$ sono i pesi appresi dal training treebank e score valuta la rappresentazione in feature in base alla transizione valida scelta.

1.4.3 Transition Systems

In base alle modifiche compiute da una transizione sugli elementi di una configurazione possiamo distinguere diverse tipologie di sistemi di transizione. Descriviamo i sistemi di transizione più noti e utilizzati in letteratura. Nel seguito sarà utilizzata la notazione $\sigma|w_i$ per rappresentare lo stack che ha in cima la parola w_i, e $w_i|\beta$ per rappresentare un buffer che ha come prima parola w_i.

Il sistema di transizione **arc-standard**, proposto in Nivre (2004), è costituito da tre tipi di transizione:

1. LEFT-ARC: Aggiunge una relazione di dipendenza (w_j, r, w_i), per un qualsiasi $r \in R$, all'insieme delle relazioni A, dove w_j è la prima sullo stack e w_i la seconda parola sullo stack. In più, rimuove dallo stack la parola w_i. La transizione può essere applicata se $w_i \neq$ ROOT.

2. RIGHT-ARC: Aggiunge una relazione di dipendenza (w_i, r, w_j), per un qualsiasi $r \in R$ all'insieme A, dove w_j è la prima sullo stack e w_i la seconda parola sullo stack. Inoltre, rimuove dalla cima dello stack la parola w_j.

3. SHIFT: Rimuove la prima parola w_i nel buffer e la aggiunge in cima alla stack.

Le transizioni sono applicabili solo se lo stack e il buffer non sono vuoti. Data la natura delle transizioni, i dependency tree generati da questo sistema possono essere solo projective.

Un'alternativa al sistema arc-standard è rappresentata dal sistema di transizioni **arc-eager**, descritto in Nivre (2003). La differenza tra arc-standard e arc-eager è che in quest'ultimo le relazioni di dipendenza più a destra sono catturate prima rispetto al sistema arc-standard. Più precisamente, il processo cattura le relazioni di dipendenza a sinistra con approccio bottom-up e quelle a destra in maniera top-down. In questo modo gli archi sono aggiunti al dependency graph non appena le coppie testa-dipendente

sono disponibili. Il sistema è costruito variando leggermente le transizioni LEFT-ARC e RIGHT-ARC e aggiungendone una nuova di nome REDUCE che rimuove dalla cima dello stack la prima parola w_i.

L'ultimo sistema chiamato **arc-hybrid**, proposto in Kuhlmann et al. (2011), considera un approccio misto dei due sistemi precedenti. Il modello è un ibrido perché utilizza le transizioni SHIFT e RIGHT-ARC del sistema arc-standard, mentre utilizza la transizione LEFT-ARC del sistema arc-eager. Come il sistema arc-standard e a differenza di quello arc-eager, il sistema arc-hybrid costruisce le relazioni di dipendenza in maniera bottom-up. Poiché le transizioni sono parametrizzate su ogni possibile tipo di dipendenza, il totale delle transizioni dipende dall'insieme dei tipi di relazioni R. Ad esempio, nel sistema arc-standard il numero totale di transizioni è $|\mathcal{T}| = 2|R| + 1$. Tutti i sistemi di transizione descritti finora sono riportati nella Tabella 1.4.

Riprendiamo l'algoritmo di Figura 1.7 e vediamo come è possibile costruire il training set usando il sistema di transizione arc-standard. Consideriamo una frase $S^{(i)}$ e il relativo dependency tree annotato $T^{(i)} = (V^{(i)}, A^{(i)})$, presi dal gold treebank $\mathcal{D}$. Supposto che tutti i dependency tree siano projective possiamo calcolare la funzione ORACLE come segue:

$$\textsc{Oracle}(c_j) = \begin{cases} \textsc{Left-Arc}_r & \text{se } (\sigma_1, r, \sigma_2) \in A^{(i)} \\ \textsc{Right-Arc}_r & \text{se } (\sigma_2, r, \sigma_1) \in A^{(i)} \textbf{ e} \text{ per ogni } w, r', \\ & \text{se } (\sigma_1, r', w) \in A^{(i)} \text{ allora } (\sigma_1, r', w) \in A \\ \textsc{Shift} & \text{altrimenti} \end{cases} \tag{1.7}$$

La sequenza di transizioni $t_0, t_1, \ldots, t_{m-1}, t_m$ ottenuta ci permette di costruire la sequenza di configurazioni $c_0 \xrightarrow{t_0} c_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{m-1}} c_m \xrightarrow{t_m} c_{m+1}$ la cui configurazione finale $c_{m+1} = (\sigma, [\,], A_{m+1})$ conterrà le relazioni del dependency tree della frase $S^{(i)}$, ovvero $A_{m+1} = A^{(i)}$.

L'oracolo in (1.7) è definito *statico* perché produce una singola sequenza statica di transizioni per generare il dependency tree. In Goldberg and Nivre (2012) viene proposto un oracolo *dinamico* che, al contrario del precedente,

Transizione	Azione
	arc-standard
LEFT-ARC$_r$	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_j, \beta, A \cup \{(w_j, r, w_i)\})$
RIGHT-ARC$_r$	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_i, \beta, A \cup \{(w_i, r, w_j)\})$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$
	arc-eager
LEFT-ARC$_r$	$(\sigma\|w_i, w_j\|\beta, A) \Rightarrow (\sigma, w_j\|\beta, A \cup \{(w_j, r, w_i)\})$
RIGHT-ARC$_r$	$(\sigma\|w_i, w_j\|\beta, A) \Rightarrow (\sigma\|w_i\|w_j, \beta, A \cup \{(w_i, r, w_j)\})$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$
REDUCE	$(\sigma\|w_i, \beta, A) \Rightarrow (\sigma, \beta, A)$
	arc-hybrid
LEFT-ARC$_r$	$(\sigma\|w_i, w_j\|\beta, A) \Rightarrow (\sigma, w_j\|\beta, A \cup \{(w_j, r, w_i)\})$
RIGHT-ARC$_r$	$(\sigma\|w_i\|w_j, \beta, A) \Rightarrow (\sigma\|w_i, \beta, A \cup \{(w_i, r, w_j)\})$
SHIFT	$(\sigma, w_i\|\beta, A) \Rightarrow (\sigma\|w_i, \beta, A)$

Tabella 1.4: Sistemi di transizioni per il transition-based dependency parsing. Il simbolo $\Rightarrow$ è usato per separare la configurazione prima dell'applicazione della transizione (sinistra) da quella dopo l'applicazione della transizione (destra).

per ogni dependency tree può generare più di una sequenza di transizioni costruita in modo da adattarsi dinamicamente al cambiamento delle configurazioni. Per ogni configurazione si sceglie la transizione ottima, cioè quella che non induce il parser in errori di predizione.

1.5 Graph-Based Dependency Parsing

Nell'approccio **graph-based** il modello di parsing è costruito direttamente sulla struttura del dependency tree senza utilizzare rappresentazioni intermedie, come avviene per la conversione in sequenza di configurazioni nell'approccio transition-based. L'idea alla base è che i dependency tree sia-

no alberi orientati che connettono tra loro le parole all'interno della frase a partire da una parola radice ROOT.

Definiamo il problema di parsing come la ricerca di un dependency tree $\hat{T}$ tra tutti i possibili dependency tree candidati che si possono ottenere partendo dal grafo ottenuto connettendo tra loro tutte le parole della frase. Per scegliere quale dependency tree tra i candidati sia il migliore è necessario introdurre il concetto di *score* che assegna un punteggio a ciascun dependency tree presente nell'insieme di tutti i dependency tree candidati $\mathcal{G}_S$ per una data frase S.

Definizione 15. Sia T un dependency tree scomponibile nei sotto-grafi $G_1, \ldots, G_m$. Definiamo la funzione *score* che assegna il punteggio ad un dependency tree T come la somma dello *score* dei suoi sotto-grafi $G_1, \ldots, G_m$:

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

In base a come definiamo la funzione score e scomponiamo il dependency tree possiamo ottenere diversi sistemi graph-based.

1.5.1 Parsing

Sia $\mathcal{G}_S$ l'insieme dei possibili dependency tree per una data frase S, il parsing nell'approccio graph-based consiste nel restituire il dependency tree $\hat{T}$, tra tutti i candidati in $\mathcal{G}_S$, che massimizza lo score per la frase S:

$$\hat{T} = \underset{T \in \mathcal{G}_S}{\arg\max}\ \text{score}(T) \tag{1.8}$$

Il modello più semplice, descritto in McDonald et al. (2005), è noto come **arc-factored** e consiste nel definire lo score per ogni singolo arco $(w_i, r, w_j) \in A$. In altre parole lo score di un dependency tree $T = (V, A)$ è la somma degli score assegnati alle singole relazioni:

$$\hat{T} = \underset{T \in \mathcal{G}_S}{\arg\max}\ \text{score}(T) = \underset{T \in \mathcal{G}_S}{\arg\max} \sum_{(w_i, r, w_j) \in A} \text{score}(w_i, r, w_j) \tag{1.9}$$

I dependency tree candidati $\mathcal{G}_S$ possono essere contenuti simultaneamente all'interno di un unico grafo pesato orientato e completo in cui tutte le parole della frase S sono connesse tra loro.

Data una frase $S = w_0 w_1 \dots w_n$, di cui vogliamo fare il parsing, e un insieme di relazioni di dipendenza $R = \{r_1, \dots, r_m\}$ possiamo costruire il grafo $G_S = (V_S, A_S)$ dei possibili alberi candidati nel seguente modo:

- $V_S = \{w_0, w_1, \dots, w_n\}$
- $A_S = \{(w_i, w_j) \mid \text{per ogni } w_i, w_j \in V_S, \text{ dove } j \neq 0\}$
- $\text{score}(w_i, w_j) = \max_{r \in R} \text{score}(w_i, r, w_j)$

L'ultima condizione in elenco ci permette di ignorare le etichette del tipo di dipendenza r e considerare unicamente gli archi (w_i, w_j) con lo score maggiore. Prima di rimuovere l'etichetta r è importante memorizzare l'arco (w_i, w_j) a cui questa è associata, in modo che si possa in seguito ricostruire l'intera relazione di dipendenza (w_i, r, w_j). Rimuovendo il tipo di dipendenza r gli archi del grafo G_S non sono più etichettati, dunque il grafo è *unlabeled*. G_S è un grafo completo considerato l'insieme di nodi $V_S - \{w_0\}$ ed esiste, per ogni parola della frase, un arco entrante che parte da w_0.

A partire dal grafo pesato senza etichette G_S dei candidati dobbiamo cercare l'albero la cui somma totale dei pesi sugli archi sia massima e, successivamente, ricostruire le etichette r originali, salvate in precedenza, per ciascun arco (w_i, w_j) dell'albero massimale. Nella teoria dei grafi questo albero è noto come *massimo albero di copertura*.

Definizione 16. Un **maximum spanning tree** (massimo albero di copertura) o MST di un grafo orientato $G = (V, A)$ è il sottografo $G' = (V', A')$, che ha la somma del dei pesi degli archi più alta e che soddisfa le seguenti condizioni:

- $V' = V$, cioè G' copre tutti i nodi originali di G;
- G' è un albero orientato.

Data una frase $S = w_0 w_1 \dots w_n$, fare parsing usando il modello arc-factored è equivalente a trovare l'MST di G_S. Il grafo G_S dei candidati alberi è composto da dependency tree projective e non-projective. Imponendo o meno delle restrizioni durante la ricerca dell'MST su G_S possiamo restituire come massimo albero di copertura un dependency tree projective o non-projective.

L'algoritmo Chu-Liu/Edmonds, riportato in Figura 1.8, non impone alcuna restrizione sullo spazio di ricerca dei possibili candidati, per cui l'MST restituito può essere anche un dependency tree non-projective.

function ChuLiuEdmonds($G_S = (V_S, A_S)$, score)
  $A \leftarrow \{(\hat{w}_i, w_j) \mid w_j \in V,\ \hat{w}_i = \arg\max_{w_i \in V} \text{score}(w_i, w_j)\}$
  $G_M \leftarrow \{V_S, A\}$
  if G_M non contiene cicli **then**
    return G_M
  else
    $C \leftarrow$ GetCycle(G_M)
    $G_C \leftarrow$ Contract(G_M, C, score)
    $G \leftarrow$ ChiuLiuEdmonds(G_C, score)
    $T \leftarrow$ Expand(G, C)
    return T

Figura 1.8: L'algoritmo Chu-Liu/Edmonds per la ricerca del maximum spanning tree (MST), nel modello arc-factored, sul grafo dei candidati G_S. L'algoritmo restituisce il dependency tree che massimizza lo score che può essere sia non-projective che projective.

Il primo passo dell'algoritmo consiste nella ricerca su G_S (Figura 1.9a) dell'arco entrante, per ogni nodo $w_j \in V$ con lo score massimo, in modo da costruire il grafo G_M (Figura 1.9b). Si verifica che G_M non contenga alcun ciclo, in tal caso il grafo è necessariamente un MST e quindi l'algoritmo termina. Se, invece, G_M contiene almeno un ciclo allora l'algoritmo ne considera uno scelto arbitrariamente. Nel caso di Figura 1.9b l'unico ciclo presente è

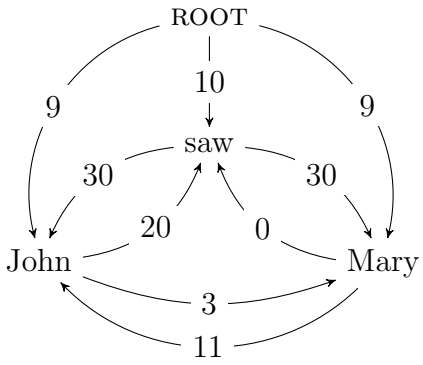
John → saw → John. Individuato il ciclo si richiama la funzione CONTRACT che si occupa di compattare il ciclo in un unico nodo e ricalcolare il peso degli archi entranti e uscenti (Figura 1.9c) richiamando l'algoritmo ricorsivamente sul nuovo grafo contratto (Figura 1.9d). L'algoritmo si conclude con la ricostruzione del grafo originale espandendo i nodi contratti prima della chiamata ricorsiva (Figura 1.9e).

Il numero massimo di chiamate ricorsive è limitato da $O(n)$ poiché il grafo non può essere contratto più di n volte. Ogni chiamata ricorsiva necessita di $O(n^2)$ passi per trovare l'arco con lo score massimo per ciascuna parola, identificare il ciclo e compattare il grafo. Dunque, la complessità totale dell'algoritmo è $O(n^3)$. Nella pratica, tuttavia, si utilizza una versione migliorata che è $O(n^2)$. Come osservato in McDonald et al. (2005) la ricerca sull'intero spazio degli alberi candidati senza restrizioni può comportare degli svantaggi. Alcune lingue, nonostante possano contenere costrutti sintattici non-projective, sono principalmente costituite da costrutti projective. Ricercando tra tutti i possibili alberi candidati non-projective si corre il rischio di trovare dependency tree non desiderabili.

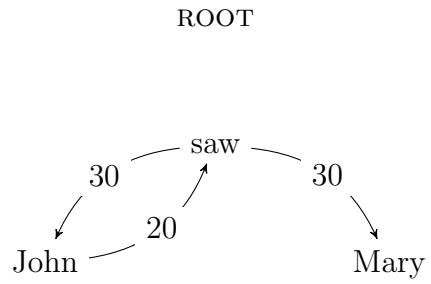
L'algoritmo di Eisner, descritto in McDonald et al. (2006), introduce delle restrizioni nella ricerca dell'MST in modo da restituire esclusivamente dependency tree che abbiano proprietà projective. A differenza dell'algoritmo Chu-Liu/Edmonds, che è di tipo greedy e ricorsivo, usa principi di programmazione dinamica. La procedura è sviluppata seguendo un approccio bottom-up che mantiene il vincolo, ad ogni passo, che l'albero sia projective.

L'algoritmo, riportato in Figura 1.10, fa uso di una tabella $E[s][t][d][c]$ che si occupa di rappresentare il sotto-albero con la somma massima dei pesi che copre le parole da w_s a w_t, dove $s \leq t$, con direzione $d \in \{0, 1\}$ e indicatore di completamento $c \in \{0, 1\}$. Se $d = 1$ allora w_s è la testa del sotto-albero, altrimenti la testa è w_t. Se $c = 1$ allora il sotto-albero non è completo, altrimenti è completo.

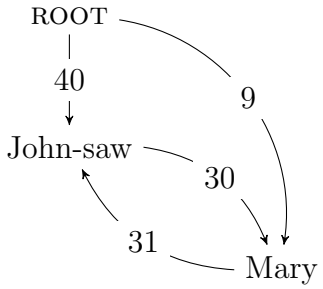
L'algoritmo comincia inizializzando a zero tutti i valori della tabella. Nel ciclo interno la prima fase consiste nel calcolare il valore massimo ottenuto



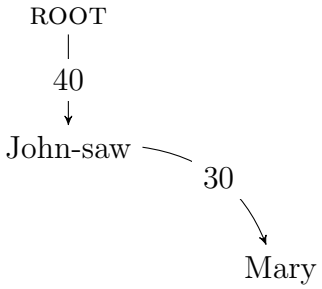
(a) Grafo dei candidati G_S



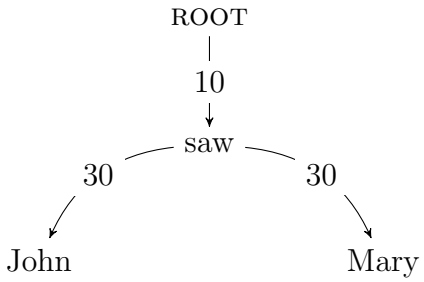
(b) Grafo G_M dei massimi archi entranti



(c) Grafo compatto ottenuto applicando CONTRACT sul ciclo John → saw → John



(d) Grafo G_M dei massimi archi entranti ottenuto dal grafo compatto



(e) Espansione del grafo contratto applicando EXPAND

Figura 1.9: L'esempio in figura mostra l'applicazione dell'algoritmo Chu-Liu/Edmonds ad un grafo G_S.

dalla fusione dei due sotto-alberi con tutti gli archi delle parole w_q, comprese tra w_s e w_t. Dopo aver aggiunto i nuovi archi l'algoritmo tenta di aggiungere i corrispondenti sottoalberi agli archi precedentemente aggiunti. Al termine dell'algoritmo lo score del miglior dependency tree si trova all'interno dell'elemento $E[0][n][1][0]$. L'algoritmo di Eisner ha complessità temporale $O(n^3)$ e complessità spaziale $O(n^2)$, data dalla tabella E.

function EISNER($S = w_0 w_1 \dots w_n$, score)
  $E[s][s][d][c] \leftarrow 0.0$ per ogni s, d, c
  for $m \leftarrow 1$ **to** n **do**
    for $s \leftarrow 1$ **to** n **do**
      $t \leftarrow s + m$
      if $t > n$ **then**
        break
      $E[s][t][0][1] \leftarrow \max_{s \le q < t}(E[s][q][1][0] + E[q+1][t][0][0] + \text{score}(w_t, w_s))$
      $E[s][t][1][1] \leftarrow \max_{s \le q < t}(E[s][q][1][0] + E[q+1][t][0][0] + \text{score}(w_s, w_t))$
      $E[s][t][0][0] \leftarrow \max_{s \le q < t}(E[s][q][0][0] + E[q][t][0][1])$
      $E[s][t][1][0] \leftarrow \max_{s < q \le t}(E[s][q][1][1] + E[q][t][1][0])$
  return $E[0][n][1][0]$

Figura 1.10: L'algoritmo di Eisner per la ricerca del maximum spanning tree (MST), nel modello arc-factored, sul grafo dei candidati G_S. L'algoritmo restituisce il dependency tree projective che massimizza lo score.

1.5.2 Learning

Nella fase di parsing si fa uso di una funzione score che assegna un peso agli archi del grafo dei dependency tree candidati. La fase di learning si pone come obiettivo quello di apprendere la funzione score a partire da un treebank usato come training set $\mathcal{D}^{(train)} = \{(S^{(i)}, T^{(i)})\}_{i=1}^{N}$.

La funzione score può essere approssimata da un classificatore composto dalla rappresentazione in feature dei sottografi di un dependency tree T con i relativi pesi delle feature $\boldsymbol{w}$. Nel caso del modello arc-factored la rappresentazione in feature $\mathbf{f}(w_i, r, w_j)$ è relativa alle singole relazioni di dipendenza, dunque lo score sarà calcolato come:

$$\text{score}(w_i, r, w_j) = \boldsymbol{w} \cdot \mathbf{f}(w_i, r, w_j) \tag{1.10}$$

La funzione $\mathbf{f}$ può includere qualunque tipo di feature rilevante della frase S catturate similmente all'approccio transition-based, già discusso nella Sezione 1.4.2. Poiché nell'approccio graph-based si parametrizza il modello direttamente sulla struttura a grafo, non c'è bisogno di alcuna trasformazione del training set, al contrario di quanto avviene nell'approccio transition-based.

1.5.3 Higher-Order Parsing

Nel modello arc-factored lo score è parametrizzato su ogni singola relazione di dipendenza nel dependency tree, si può rilassare questo vincolo in modo da considerare relazioni più complesse. Fissato un arco (w_i, w_j) nel dependency tree possiamo considerare archi sullo stesso livello dell'albero (dimensione orizzontale) o su livelli superiori o inferiori (dimensione verticale).

Definizione 17. Una fattorizzazione di d-esimo ordine considera nello score di un dependency tree esattamente d archi, l'arco fissato più i $d - 1$ vicini (orizzontali o verticali).

In altre parole, l'ordine stabilisce il numero di dipendenze che lo score contiene. In Figura 1.11 sono riportate le possibili fattorizzazioni di ordine superiore considerando sia la dimensione verticale sia quella orizzontale.

Il modello arc-factored è una fattorizzazione di primo ordine, o **first-order**, poiché considera solo una relazione di dipendenza $h \to m$. Se passiamo ad una fattorizzazione del secondo ordine sul contesto orizzontale, oltre alla relazione $h \to m$ considereremo anche la relazione $h \to s$, in questo caso entrambe le relazioni hanno la stessa testa h. Se invece ci espandiamo sul

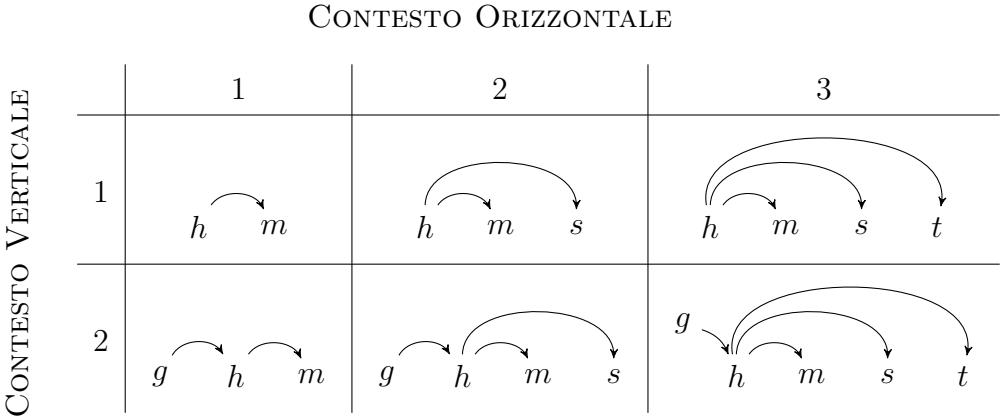


Figura 1.11: In Figura sono riportate le diverse fattorizzazioni di ordine superiore in base al contesto orizzontale e verticale considerato come riportato in McDonald and Nivre (2014).

contesto verticale anziché su quello orizzontale, oltre alla relazione $h \rightarrow m$ considereremo la relazione $g \rightarrow h$, in questo caso h è testa solo per la prima relazione, mentre è dipendente nella seconda.

In McDonald and Pereira (2006) sono stati sviluppati dei metodi per il parsing del secondo ordine. In questo caso lo score dell'albero score(w_i, w_k, w_j) è definito come la somma di coppie di archi adiacenti (w_i, w_k) e (w_k, w_j). L'algoritmo di Eisner per il parsing può essere esteso ad un arbitraria fattorizzazione di ordine m mantenendo la complessità di $O(n^{m+1})$ per $m > 1$. Per gli ordini superiori al primo la ricerca di un MST non-projective, tramite l'algoritmo Chu-Liu/Edmonds, è *NP-hard*, dunque intrattabile. Per aggirare questo problema si può utilizzare una variante dell'algoritmo di Eisner che ha complessità $O(n^3)$, per fattorizzazioni del secondo ordine ed approssima la ricerca degli alberi non-projective. L'idea alla base di questa variante è di trovare, come avviene per l'algoritmo standard di Eisner, il projective dependency tree che massimizzi lo score e successivamente riordini gli archi uno alla volta in modo da incrementare lo score complessivo senza violare i vincoli che garantiscono le proprietà non-projective del dependency tree.

1.6 Valutazione

Per valutare le prestazioni di un parser occorre avere a disposizione un treebank $\mathcal{D}^{(\text{test})}$ i cui dependency tree non siano stati utilizzati in alcun modo durante la fase di apprendimento, perché in tal caso si potrebbe alterare la valutazione. Durante la fase di valutazione o testing il parser prende in input dal treebank solo le frasi $S^{(i)}$, ignorando il dependency tree $T^{(i)}$. Per ciascuna frase $S^{(i)}$ restituisce il dependency tree $\hat{T}^{(i)}$ predetto che sarà quindi confrontato con il dependency tree corretto $T^{(i)}$, usato come riferimento, dello stesso treebank.

Solitamente, i dependency tree predetti sono codificati nello stesso formato con il quale il parser legge i treebank. In Figura 1.12 è mostrata la codifica CoNLL-X del dependency tree predetto da un generico parser per la frase dell'esempio di Figura 1.4.

Il confronto tra il dependency tree predetto $\hat{T}^{(i)}$ e il dependency tree corretto $T^{(i)}$, che funge da gold standard, avviene tramite l'uso di diverse metriche, le più usate sono:

- **Exact match (EM)**: È la percentuale delle frasi correttamente predette. Vale a dire la percentuale di dependency tree tale che $\hat{T}^{(i)} = T^{(i)}$.

- **Label accuracy score (LS)**: È la percentuale delle parole predette che hanno la stessa etichetta della relazione di dipendenza di riferimento.

- **Unlabled attachment score (UAS)**: È la percentuale delle parole predette che hanno la stessa testa della relazione di dipendenza di riferimento.

- **Labeled attachment score (LAS)**: È la percentuale delle parole predette che hanno la stessa etichetta e testa della relazione di dipendenza di riferimento.

system.out

```
1   Il       _   DET_RD   DET_RD   _   2   det        _   _
2   Governo  _   NOUN_S   NOUN_S   _   0   root       _   _
3   deve     _   AUX_VM   AUX_VM   _   4   mark       _   _
4   avere    _   VERB_V   VERB_V   _   2   acl:relcl  _   _
5   la       _   DET_RD   DET_RD   _   6   det        _   _
6   fiducia  _   NOUN_S   NOUN_S   _   4   obj        _   _
7   di       _   ADP_E    ADP_E    _   10  case       _   _
8   le       _   DET_RD   DET_RD   _   10  det        _   _
9   due      _   NUM_N    NUM_N    _   10  amod       _   _
10  Camere   _   NOUN_S   NOUN_S   _   6   nmod       _   _
11  .        _   PUNCT_FS PUNCT_FS _   2   punct      _   _
```

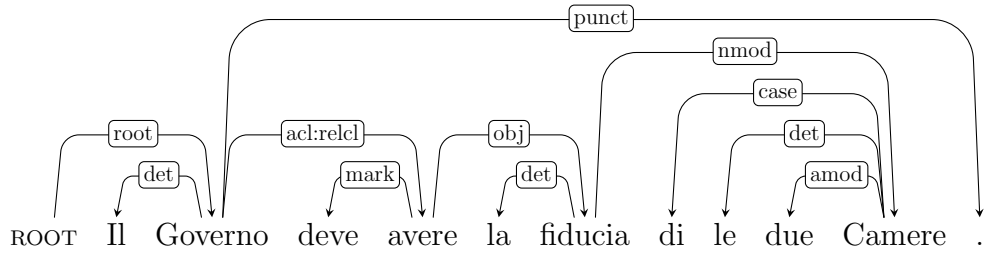


Figura 1.12: Codifica in formato CoNLL-X (sopra) del dependency tree predetto (sotto) da un generico parser per la frase "*Il Governo deve avere la fiducia di le due Camere.*".

- **Precision**: È la percentuale delle dipendenze correttamente predette dal sistema:

$$\text{Precision} = \frac{\#\text{corrette}}{\#\text{system}}$$

- **Recall**: È la percentuale delle dipendenze predette corrette rispetto al treebank di riferimento:

$$\text{Recall} = \frac{\#\text{corrette}}{\#\text{gold}}$$

- **F1 score**: È la media armonica di Precision e Recall:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Utilizzando il dependency tree di Figura 1.4 come gold standard calcoliamo le metriche di UAS, LAS e LS rispetto al dependency tree predetto di Figura 1.12. Nella Tabella 1.5 sono messe a confronto, per ciascuna parola, tutte le relazioni dei due dependency tree. Per ciascuna parola, sulle colonne relative alle metriche, la spunta indica che la relazione contribuisce al conteggio della specifica metrica.

gold			**system**			**metriche**		
dipendente	*deprel*	*testa*	*dipendente*	*deprel*	*testa*	**UAS**	**LAS**	**LS**
Il	det	Governo	Il	det	Governo	✓	✓	✓
Governo	nsubj	avere	Governo	root	ROOT			
deve	aux	avere	deve	mark	avere	✓		
avere	root	ROOT	avere	acl:relcl	Governo			
la	det	fiducia	la	det	fiducia	✓	✓	✓
fiducia	obj	avere	fiducia	obj	avere	✓	✓	✓
di	case	Camere	di	case	Camere	✓	✓	✓
le	det	Camere	le	det	Camere	✓	✓	✓
due	nummod	Camere	due	amod	Camere	✓		
Camere	nmod	fiducia	Camere	nmod	fiducia	✓	✓	✓
.	punct	avere	.	punct	Governo			✓
						8/11	6/11	7/11

Tabella 1.5: Confronto delle relazioni di dipendenza tra due dependency tree, uno funge da gold standard l'altro è l'albero predetto dal sistema. Sulle colonne a destra sono mostrate le metriche di UAS, LAS e LS. Una spunta sulla colonna di una metrica indica che la relativa relazione di dipendenza contribuisce al conteggio per la specifica metrica.

Per calcolare l'UAS si verifica semplicemente se la testa della relazione tra il gold standard e il dependency tree predetto dal sistema è uguale. Consideriamo la relazione *deve* $\xleftarrow{mark}$ *avere* dell'albero predetto con la corrispondente relazione del gold standard *deve* $\xleftarrow{aux}$ *avere*. Il tipo della relazione del depen-

dency tree predetto (*mark*) è diversa da quella del gold standard (*aux*), per cui non sarà conteggiata nelle metriche di LAS e LS. Invece, dato che le due relazioni hanno la stessa testa (*avere*) le considereremo valide per il conteggio della metrica UAS. Per come è definita, la metrica LAS non può mai essere maggiore dello UAS. Infatti, affinché una relazione sia valida nel conteggio del LAS, deve valere che sia il tipo di dipendenza sia la testa delle due relazioni a confronto siano uguali, mentre nello UAS è sufficiente che siano uguali solo le teste. La metrica LS, invece, verifica solamente l'uguaglianza del tipo delle relazioni di dipendenza, come possiamo notare nell'ultima relazione relativa alla punteggiatura dell'esempio di Tabella 1.5.

A volte è possibile trovare valutazioni di parser in cui la punteggiatura viene esclusa dalla valutazione, questa scelta ha come conseguenza l'innalzamento dei valori delle metriche. Se escludiamo la punteggiatura dall'esempio di Tabella 1.5, il valore di UAS salirebbe a 8/10 e quello del LAS a 6/10.

Capitolo 2

Neural Network Models

In questo capitolo saranno introdotti alcuni modelli di reti neurali contestualizzandoli nel settore dell'**intelligenza artificiale**, nello specifico del *machine learning*. Saranno trattati solo alcuni dei modelli più noti in letteratura, concentrandosi prevalentemente su quelli usati dai parser moderni.

Molta della conoscenza umana per svolgere un particolare compito (o task) è spesso intuitiva e difficile da definire formalmente. La sfida dell'intelligenza artificiale è, appunto, costruire algoritmi in grado di svolgere compiti che sono semplici per l'essere umano (riconoscimento del parlato o degli oggetti), ma difficili da descrivere formalmente. Il **machine learning** (o *apprendimento automatico*) è un ramo di ricerca dell'intelligenza artificiale che si pone l'obiettivo di sviluppare tecniche e metodologie in grado di acquisire automaticamente conoscenza e modelli computazionali dai dati.

Le prestazioni degli algoritmi dipendono molto dalla rappresentazione dei dati su cui l'algoritmo lavora, ogni informazione relativa ai dati è detta **feature**. Di tutte le features a nostra disposizione se ne selezionano manualmente alcune che possono essere significative per il task che vogliamo svolgere e in seguito passate agli algoritmi di machine learning che si occupano di acquisire conoscenza da queste. Per compiti più complessi è difficile stabilire o conoscere quali features possano essere utili. Immaginiamo di voler riconoscere se in una foto è presente o meno un volto. Potremmo usare la presenza o

l'assenza di naso, bocca, orecchie e quant'altro per stabilire se nell'immagine è presente o meno un volto. Purtroppo questo è difficile da descrivere algoritmicamente poiché il computer ha a disposizione solo la rappresentazione in pixel della foto, in altre parole ogni pixel dell'immagine è una feature.

Una specifica applicazione del machine learning, nota come *representation learning*, usa algoritmi che, in maniera automatica, permettono di ricavare quelle features che sono più idonee a rappresentare i dati, piuttosto che selezionarle manualmente. Che sia manuale o automatica, la scelta delle feature, consiste nel separare quei fattori di variazione che ci permettono di spiegare i dati osservati. Ad esempio, quando si analizza il parlato, i fattori di variazioni possono includere l'età, il sesso e l'accento. Estrarre features così astratte potrebbe essere molto complesso poiché la rappresentazione dei dati è costituita, nel caso del riconoscimento vocale, essenzialmente dal valore di frequenza del suono registrato. Grazie alle tecniche di **deep learning** è possibile risolvere questo problema rappresentando features complesse (accento, sesso) in termini di features più semplici (frequenza del suono).

In Figura 2.1 è riportata la relazione gerarchica tra i tre tipi di approcci all'apprendimento automatico appena descritti. In Figura 2.2 sono rappresentati dei diagrammi di flusso che schematizzano l'approccio dei diversi tipi di apprendimento automatico, con particolare enfasi sui blocchi che sono portati a termine in maniera completamente automatica.

Nel seguito trattiamo più in dettaglio cosa vuol dire apprendimento e come può essere costruito un algoritmo che apprenda dai dati in maniera automatica, relativamente ai task di classificazione supervisionata su cui si fonda il dependency parsing.

2.1 Machine Learning

In Mitchell (1997) si dice che un algoritmo **apprende** dall'esperienza E rispetto a un task T e una misura delle prestazioni P, se le performance del task T, misurate con P, migliorano con l'esperienza E. Nel caso del depen-

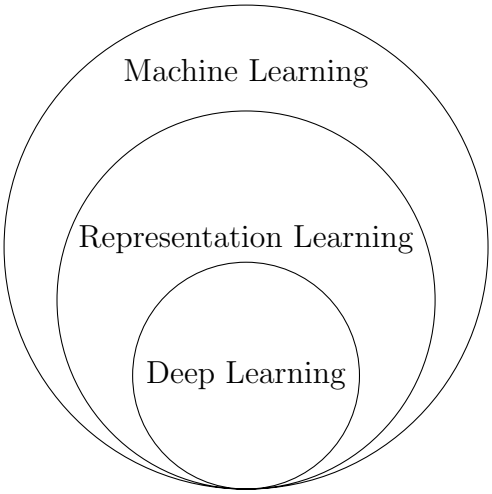


Figura 2.1: Relazione gerarchica tra i tre tipi di approccio all'apprendimento automatico, così come illustrata in Goodfellow et al. (2016). Il deep learning è una particolare applicazione del representation learning che a sua volta è una particolare applicazione di machine learning.

dency parsing transition-based, l'esperienza E è rappresentata dal treebank, le misure delle performance P sono le metriche di valutazione come UAS o LAS e il task T consiste nell'approssimare tramite un classificatore la funzione ORACLE che predice la transizione successiva. In altre parole l'apprendimento è il processo che permette di raggiungere l'abilità nello svolgere un particolare compito. Per tutto il resto del capitolo tratteremo esclusivamente il task di *classificazione supervisionata*.

Gli algoritmi di machine learning sono descritti in modo da elaborare istanze di dati rappresentati come tuple $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n$ prese da un insieme di m istanze $\boldsymbol{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\} \in \mathbb{R}^{m \times n}$, dove $x_i^{(j)} \in \mathbb{R}$ è la i-esima feature della j-esima istanza $\boldsymbol{x}^{(j)} \in \boldsymbol{X}$.

La **classificazione** è un particolare task in cui si cerca di costruire un **classificatore** che sia in grado di restituire in output la classe di appartenenza, tra k classi possibili, di una generica istanza di input $\boldsymbol{x}$. Ad esempio, nel caso del dependency parsing transition-based un classificatore ha il compito di predire una tra tutte le possibili transizioni, data una configurazione in

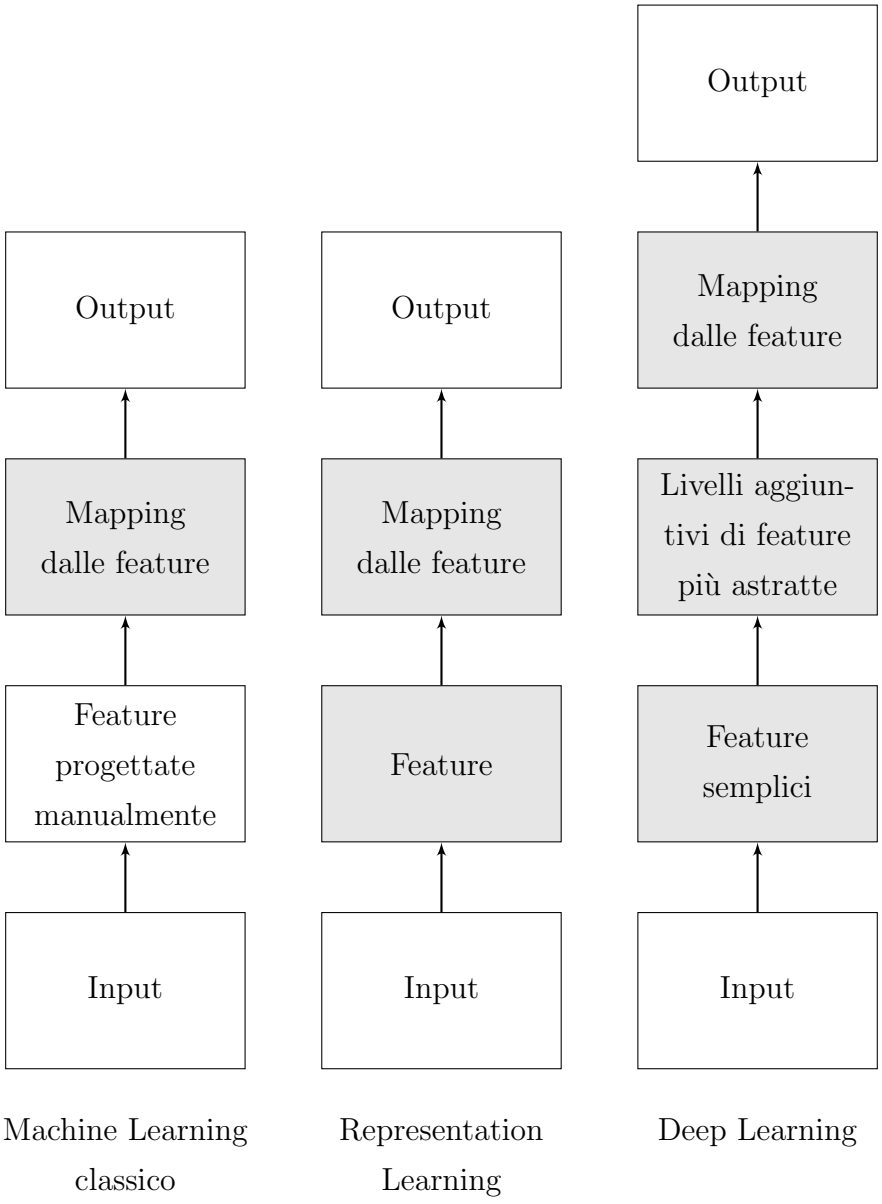


Figura 2.2: Diagrammi di flusso che mostrano le differenze tra i tre tipi di apprendimento automatico così come riportati in Goodfellow et al. (2016). I blocchi in grigio indicano le componenti che sono in grado di apprendere dai dati in maniera automatica.

input.

Nell'apprendimento di tipo **supervisionato** l'algoritmo ha a disposizione

per ciascuna istanza $\boldsymbol{x}^{(i)}$ la relativa classe di appartenenza $\boldsymbol{y}^{(i)}$. L'insieme delle classi corrette $\boldsymbol{Y} = \{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(m)}\} \in \mathbb{R}^{m \times k}$ è costituito dai vettori target $\boldsymbol{y}^{(i)}$ costruiti in modo tale che l'elemento j-esimo del vettore valga 1 se l'istanza $x^{(i)}$ appartiene alla classe j:

$$y_j^{(i)} = \begin{cases} 1 & \text{se } \boldsymbol{x}^{(i)} \text{ è etichettato con la classe } j \\ 0 & \text{altrimenti} \end{cases} \tag{2.1}$$

Ad esempio, nel caso di un compito di classificazione con quattro classi possibili, se $\boldsymbol{x}^{(i)}$ è etichettato come classe 2 allora $\boldsymbol{y}^{(i)} = [0, 1, 0, 0]$.

In un task di classificazione supervisionata l'obiettivo è quello di apprendere una funzione f che assegni ad un nuovo input $\boldsymbol{x} \notin \boldsymbol{X}$ una classe $\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ tale che $\hat{\boldsymbol{y}} = \boldsymbol{y}$.

La misura adottata per valutare le prestazioni di un classificatore è l'accuratezza, ovvero la percentuale di esempi che sono stati etichettati con la classe corretta (come avviene per UAS e LAS). Per valutare correttamente un classificatore è necessario usare esempi di input che non siano stati usati nella fase di apprendimento del modello, per capire quanto l'algoritmo sia riuscito a generalizzare il problema. La valutazione, per questo motivo, è effettuata su un insieme di dati, detto **test set**, disgiunto dall'insieme dei dati usati per apprendere il modello, detto **training set**.

Gli algoritmi di machine learning sono una forma di statistica applicata che fa assunzioni sui dati osservati per costruire un modello. Nell'approccio **parametrico** si assume che i dati osservati siano stati generati a partire da un certa distribuzione con parametri Θ, questi ultimi devono essere stimati a partire dagli esempi nel training set $\mathcal{D}^{(\text{train})} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{m}$. Nei modelli lineari si stabiliscono i parametri $\Theta = \{\boldsymbol{W}, \boldsymbol{b}\}$ e su questi si definisce la funzione f_Θ da apprendere:

$$\hat{\boldsymbol{y}} = f_\Theta(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \tag{2.2}$$

dove $\boldsymbol{W} \in \mathbb{R}^{k \times n}$, $\boldsymbol{b} \in \mathbb{R}^k$. I parametri $w_{ij} \in \boldsymbol{W}$ e $b_i \in \boldsymbol{b}$ sono anche chiamati *pesi* e il vettore $\boldsymbol{b}$ è detto *bias*. Supponendo di voler apprendere

un modello lineare con $\boldsymbol{W} \in \mathbb{R}^{4\times 10}$ e $\boldsymbol{b} \in \mathbb{R}^4$ dobbiamo stimare esattamente $4 \times 10 + 4 = 44$ parametri (o pesi).

In un task di classificazione solitamente si usa normalizzare il risultato della Equazione (2.2) in modo che il vettore restituito $\hat{\boldsymbol{y}}$ abbia valori compresi nell'intervallo $[0, 1]$ e la somma totale degli elementi sia uguale a 1. In questo modo possiamo interpretare il vettore di output $\hat{\boldsymbol{y}}$ normalizzato come una distribuzione di probabilità in cui più un elemento $\hat{y}_c \in \hat{\boldsymbol{y}}$ è vicino a 1 più è probabile che l'esempio $\boldsymbol{x}$ appartenga alla classe c. Per normalizzare il risultato si applica la funzione **softmax**.

Definizione 18. Sia $\boldsymbol{z} = (z_1, \ldots, z_k) \in \mathbb{R}^k$ un vettore reale di dimensione k la funzione softmax : $\mathbb{R}^k \to \mathbb{R}^k$ è definita nel seguente modo:

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \qquad \text{con } 1 \leq i \leq k. \tag{2.3}$$

Per ottenere un vettore $\hat{\boldsymbol{y}}$ normalizzato applichiamo la funzione softmax all'Equazione (2.2):

$$\hat{\boldsymbol{y}} = f_\Theta(\boldsymbol{x}) = \text{softmax}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.4}$$

Ad esempio, dato un generico input $\boldsymbol{x}$ calcoliamo il vettore

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} = [0.3, 0.6, 1.2, -0.3] \tag{2.5}$$

che normalizziamo in $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{z}) = [0.19, 0.26, 0.47, 0.08]$. Dunque, la classe più probabile a cui appartiene l'input $\boldsymbol{x}$ è la 3.

Per poter stimare i parametri Θ si utilizza una funzione $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$, detta **loss function**, che quantifica l'errore commesso data la predizione $\hat{\boldsymbol{y}}$ (classe predetta) e il relativo valore atteso $\boldsymbol{y}$ (classe reale). Sia $\mathbb{C} = \{1, \ldots, k\}$ l'insieme delle k possibili classi tra cui scegliere, descriviamo alcune loss function per la classificazione:

- La *cross-entropy loss*, anche nota come *negative log-likelihood*, misura la dissomiglianza tra il vettore target reale $\boldsymbol{y}$ e la predizione $\hat{\boldsymbol{y}}$ usando l'entropia:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i\in\mathbb{C}} y_i \log(\hat{y}_i) \tag{2.6}$$

Quando si usa questa loss è necessario che l'output sia normalizzato tramite la funzione softmax. Nel caso in cui la classe corretta sia solo una, indicata con t, la cross-entropy si riduce al calcolo di

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\log \hat{y}_t \tag{2.7}$$

poiché solo l'elemento t-esimo del vettore target $\boldsymbol{y}$ vale 1, mentre gli altri 0. La cross-entropy cerca di impostare la funzione di probabilità assegnata alla classe corretta t a 1.

- Siano $t = \arg\max_{i \in \mathbb{C}} y_i$ la classe corretta con il più alto valore e $k = \arg\max_{i \in \mathbb{C}} \hat{y}_i$ la classe predetta con il più alto valore tale che $i \neq t$, la *hinge loss* è definita come:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \max(0, 1 - (\hat{y}_t - \hat{y}_k)) \tag{2.8}$$

La hinge loss cerca di assegnare alla classe corretta un margine di almeno 1 rispetto a tutte le altre classi.

Dato un training set etichettato $\mathcal{D}^{(\text{train})} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}$, una loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ e il risultato del modello parametrico $f_\Theta(\boldsymbol{x}^{(i)}) = \hat{\boldsymbol{y}}^{(i)}$, definiamo la **train loss** $\mathcal{L}$ rispetto ai parametri Θ come la media del valore della loss function su ciascun esempio di training:

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)}) \tag{2.9}$$

L'obiettivo dell'algoritmo di training consiste nel trovare il valore dei parametri $\hat{\Theta}$ che minimizzano la train loss $\mathcal{L}$:

$$\hat{\Theta} = \arg\min_{\Theta} \mathcal{L}(\Theta) \tag{2.10}$$

Per trovare il punto minimo di una generica funzione $y = f(x)$ è possibile usare metodi numerici basati sulla *discesa del gradiente* in cui si fa uso della derivata prima $f'(x)$, come indicatore della direzione del cambiamento della funzione. Assegnando un valore iniziale a x_0 si valuta la funzione nel punto

$g = f'(x_i)$ con $i \geq 0$. Se $g = 0$ allora x_i è un punto di minimo e l'algoritmo termina, in caso contrario ci muoviamo nella direzione opposta del gradiente g, aggiornando il valore $x_{i+1} \leftarrow x_i - \eta g$, dove η è detto **learning rate**. Per poter applicare questi metodi è necessario che le function loss siano differenziabili, per poter calcolare la derivata, e desiderabilmente convesse, in modo che il punto di minimo trovato dall'algoritmo sia anche un punto di minimo globale.

Uno degli algoritmi più noti di discesa del gradiente utile per risolvere il problema di ottimizzazione dei parametri Θ dell'Equazione (2.14) è l'algoritmo **Stochastic Gradient Descent** (**SGD**) riportato in Figura 2.3.

function SGD($\mathcal{D}^{(\text{train})}, L, f_\Theta, \Theta, \eta, b, \text{epochs}$)
  $\hat{\Theta} \leftarrow$ INITIALIZE(Θ)
  for $e = 1$ **to** epochs **do**
    $\mathcal{B} \leftarrow$ SAMPLE($\mathcal{D}^{(\text{train})}, b$)
    $g \leftarrow 0$
    for each $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \in \mathcal{B}$ **do**
      $\hat{\boldsymbol{y}}^{(i)} \leftarrow f_{\hat{\Theta}}(\boldsymbol{x}^{(i)})$
      $g \leftarrow g + [\nabla_{\hat{\Theta}} L(\hat{\boldsymbol{y}}^{(i)}, \boldsymbol{y}^{(i)})]/b$
    $\hat{\Theta} \leftarrow \hat{\Theta} - \eta g$
  return $\hat{\Theta}$

Figura 2.3: Algoritmo di ottimizzazione SGD basato sulla discesa del gradiente. La funzione INITIALIZE inizializza i parametri in base ad una distribuzione scelta. La funzione SAMPLE restituisce un mini-batch $\mathcal{B}$ di dimensione b a partire dal training set. La funzione restituisce i parametri $\hat{\Theta}$ aggiornati.

L'algoritmo comincia inizializzando i parametri Θ. Un inizializzazione spesso usata consiste nell'assegnare ai parametri un valore, scelto in maniera uniforme, compreso nell'intervallo $(-0.01, 0.01)$. Un'altra inizializzazione, detta di *Glorot* o *Xavier*, prevede che i valori dei parametri siano scelti, in

maniera uniforme, nell'intervallo $\pm\sqrt{6/(r+c)}$, dove r e c sono il numero di righe e colonne della struttura dello specifico parametro in Θ.

Il numero di **epochs** stabilisce quante iterazioni l'algoritmo effettua sull'intero training set prima di restituire il valore dei parametri aggiornati. Può anche capitare che l'algoritmo non converga o si blocchi in un punto di minimo locale. Per questi motivi, oltre alle epochs, possono essere associati diversi criteri di stop per l'algoritmo. A volte, invece di usare le epochs, si può specificare il numero di iterazioni massime che indicano la quantità di mini-batch sui quali iterare. Ad ogni epoch l'algoritmo estrae un insieme di esempi $\mathcal{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(b)}\}$ di dimensione b dal training set, detto **mini-batch**, su cui stima la train loss e aggiorna i parametri. La dimensione dei mini-batch può variare da 1, caso in cui la train loss è calcolata su ogni singola istanza di train, all'intera dimensione del training set (solitamente è nell'ordine delle centinaia).

In Ruder (2017) sono riportati altri algoritmi di ottimizzazione basati sulla discesa del gradiente che apportano alcune variazioni all'algoritmo SGD, ne riportiamo brevemente alcuni tra i più utilizzati:

- *Momentum.* L'algoritmo SGD ha forti oscillazioni in prossimità dei punti di ottimo locale che gli impediscono la convergenza. Tramite il momentum l'algoritmo SGD riesce a ridurre le oscillazioni considerando il gradiente del passo precedente g_{t-1} pesato con un fattore γ (solitamente 0.9):

$$\hat{\Theta} \leftarrow \hat{\Theta} - \gamma g_{t-1} - \eta g_t \tag{2.11}$$

  Il risultato è una convergenza più rapida e la riduzione delle oscillazioni.

- *Adaptive Gradient (AdaGrad).* Adatta il learning rate η rispetto ai parametri:

$$\hat{\Theta}_{t+1} \leftarrow \hat{\Theta}_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{2.12}$$

  dove $G_t \in \mathbb{R}^{d \times d}$ è una matrice diagonale in cui ciascun elemento $g_{ii} \in G_t$ è la somma dei quadrati del gradiente calcolato rispetto a Θ_i fino al

passo t e ϵ è il termine di smoothing che evita le divisioni per zero (solitamente dell'ordine di 10^{-8}). L'operatore $\odot$ indica il prodotto matrice-vettore tra G_t e g_t. Il principale beneficio di AdaGrad sta nell'eliminazione della necessità di ottimizzare manualmente il learning rate η.

- *Adaptive Moment Estimation (Adam).* È un metodo che adatta il learning rate per ogni parametro calcolando l'exponential smoothing della serie dei gradienti passati e del loro quadrato:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\ \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\ \hat{\Theta}_{t+1} &\leftarrow \Theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \end{aligned} \tag{2.13}$$

  Solitamente i valori sono $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\epsilon = 10^{-8}$.

Una volta appresi i parametri Θ, per capire se l'algoritmo di apprendimento ha prodotto un buon modello si valuta l'errore, definito come $1 -$ accuratezza, calcolato sul training set (*train error*) e sul test set (*test error*). Un buon modello deve essere in grado di tenere basso sia il train error che la differenza tra train e test error. In caso contrario, possono verificarsi due eventi non desiderabili, i quali indicano che il modello non ha la giusta capacità di generalizzare:

- Il train error è basso ma la differenza tra train e test error è alta. Ciò vuol dire che il nostro modello ha buone prestazioni sui dati osservati ma non su quelli non osservati, in questo caso si parla di **overfitting**.

- La differenza tra train e test error è bassa ma il train error è alto. Ciò vuol dire che l'algoritmo non ha buone prestazioni sui dati osservati, in questo caso si parla di **underfitting**.

In Figura 2.4 è riportato un grafico che mostra la relazione tra train e test error e le aree in cui si può avere underfitting e overfitting.

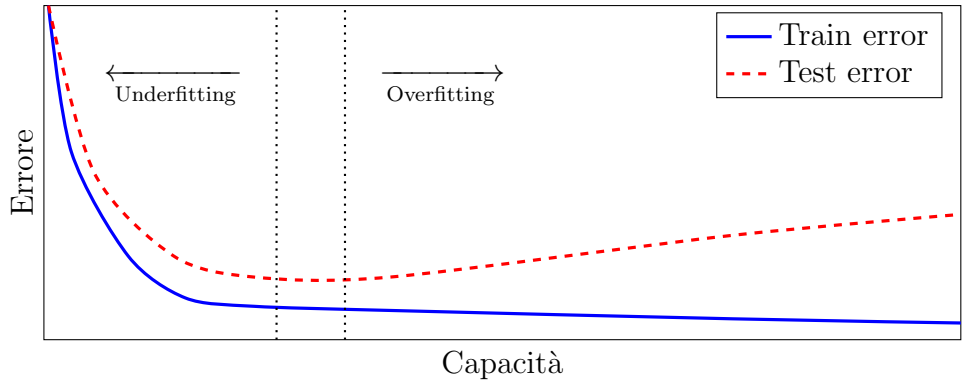


Figura 2.4: Relazione tra train error e test error. Nella zona di underfitting sia il train error che il test error sono alti. Nella zona di overfitting il train error è basso, mentre la differenza tra train e test error diventa sempre più alta. Una buona capacità di generalizzazione è compresa nell'intervallo delineato dalle linee tratteggiate.

Il problema di ottimizzazione dell'Equazione (2.10) può condurre il modello in overfitting poiché l'obiettivo è quello di minimizzare la loss function unicamente sui dati osservati presi dal training set. Per evitare l'overfitting è opportuno introdurre un termine $R(\Theta)$, detto di **regolarizzazione**, che cerca di guidare l'algoritmo di apprendimento verso un modello con una migliore capacità di generalizzazione senza produrne uno troppo specifico per i dati di training. Il problema di ottimizzazione dell'Equazione (2.10) può essere riscritto sommando il termine di regolarizzazione alla loss $\mathcal{L}$:

$$\hat{\Theta} = \arg\min_{\Theta} \mathcal{L}(\Theta) + \lambda R(\Theta) \tag{2.14}$$

Il valore di λ nella Equazione (2.14) ci permette di stabilire quanto peso dare al termine di regolarizzazione. Se $\lambda = 0$ non stiamo assegnando alcuna regolarizzazione e questo potrebbe comportare overfitting. Un valore eccessivo, invece, può generare underfitting in quanto non stiamo dando il giusto peso al train loss. Il termine di regolarizzazione più usato è il quadrato della

norma 2 calcolato sui parametri, $R(\Theta) = \|\Theta\|_2^2$, noto come regolarizzazione l_2 o L_2 o **weight decay**.

Un algoritmo di ottimizzazione è composto da parametri che non possono essere appresi dall'algoritmo stesso: il valore λ dell'Equazione 2.14, il learning rate η, la dimensione b del mini-batch, il numero di epochs e altri parametri relativi allo specifico algoritmo di ottimizzazione. Tutti i parametri che non possono essere direttamente appresi dal processo di ottimizzazione sono detti **iperparametri** e devono essere impostati manualmente in modo da cercare i valori che producano un modello con una buona capacità di generalizzazione.

Se vogliamo osservare l'andamento degli errori durante l'apprendimento per verificare che l'algoritmo stia apprendendo un buon modello con determinati valori degli iperparametri non possiamo utilizzare il test set perché la valutazione finale del modello risulterebbe alterata. Per risolvere questo problema si utilizza un **validation set** composto da un insieme di dati che non sono presenti né nel training set né nel test set. Usando il validation set possiamo ottimizzare i valori degli iperparametri con i quali apprendiamo il modello che sarà, infine, valutato sul test set.

2.2 Feedforward Neural Network

L'obiettivo dei modelli lineari, come nell'Equazione (2.2), è quello di cercare un iperpiano che separi i dati osservati nello spazio in modo che tutte le istanze di una classe siano divise da tutte quelle delle altre classi. Questo è possibile solo quando i dati sono distribuiti in modo tale da permettere una divisione lineare delle classi. Se consideriamo un task di classificazione binaria lo scopo di un modello lineare è cercare di tracciare una retta che separi le istanze di una classe da quelle dell'altra (Figura 2.5).

Nel caso in cui non sia possibile separare linearmente i dati, si può utilizzare una funzione ϕ ausiliaria che mappa i dati in una nuova rappresentazione che sia linearmente separabile. La funzione ϕ può essere definita a sua volta come un modello lineare a cui viene applicata una funzione g non lineare,

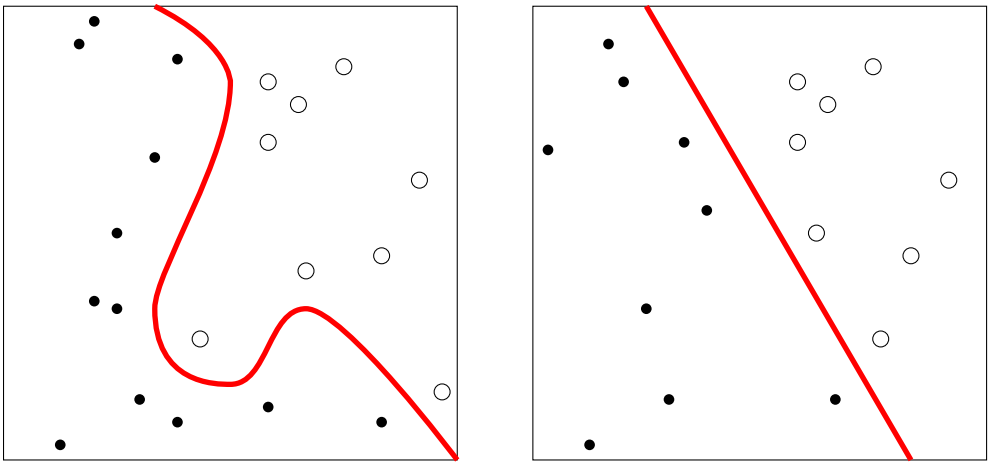


Figura 2.5: Nelle due figure sono mostrati dei dati appartenenti a due classi distinte, distribuiti sul piano, identificati da cerchi pieni e vuoti. Nella figura a destra i dati sono distribuiti in modo tale da poter essere divisi in due aree delimitate da una linea retta. A sinistra i dati non possono essere divisi da una retta ma da una funzione non lineare.

detta funzione di **attivazione**:

$$\begin{aligned} \hat{\boldsymbol{y}} &= \boldsymbol{W}\phi(\boldsymbol{x}) + \boldsymbol{b} \\ \phi(\boldsymbol{x}) &= g(\boldsymbol{W'}\boldsymbol{x} + \boldsymbol{b'}) \end{aligned} \tag{2.15}$$

L'intera espressione dell'Equazione (2.15) è differenziabile, perciò è ancora possibile applicare metodi di ottimizzazione basati sulla discesa del gradiente.

Tramite questo procedimento cerchiamo di apprendere automaticamente una nuova rappresentazione dei dati $\phi(\boldsymbol{x})$. Questo approccio è alla base delle tecniche di deep learning e dei modelli noti come **feedforward networks** o **multilayer perceptron** (MLP).

Definendo più composizioni di modelli lineari sui dati di input a cui sono applicate funzioni di attivazione, è possibile costruire architetture sempre più profonde. Ad esempio, possiamo definire la seguente architettura MLP per

un task di classificazione:

$$\begin{aligned} \boldsymbol{h}^{(1)} &= g^{(1)}(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}) \\ \boldsymbol{h}^{(2)} &= g^{(2)}(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}) \\ \boldsymbol{h}^{(3)} &= g^{(3)}(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}) \\ \boldsymbol{o} &= \boldsymbol{W}^{(4)}\boldsymbol{h}^{(3)} + \boldsymbol{b}^{(4)} \\ \hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{o}) \end{aligned} \tag{2.16}$$

dove $\Theta = \{\boldsymbol{W}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{W}^{(3)}, \boldsymbol{W}^{(4)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}, \boldsymbol{b}^{(3)}, \boldsymbol{b}^{(4)}\}$.

Ogni composizione di un MLP è definita *livello* (o *layer*), nell'esempio dell'Equazione (2.16) la rete è composta da cinque livelli. Il vettore di input $\boldsymbol{x}$ è detto *input* layer e il livello finale $\boldsymbol{o}$ è detto *output* layer. Tutti i livelli compresi tra l'input layer e l'output layer sono detti *hidden* layer. Il numero di livelli del modello stabilisce la sua profondità, ragione per cui questi modelli sono chiamati **deep**.

L'uso di funzioni di attivazione g non lineari ha un importante ruolo nell'abilità di una rete di rappresentare funzioni complesse. Senza la non linearità delle funzioni di attivazione la rete neurale sarebbe in grado di rappresentare solo trasformazioni lineari dell'input rendendo impossibile la separazione di dati non linearmente separabili. Inoltre, le funzioni di attivazioni devono essere differenziabili in modo da permettere l'uso di algoritmi di ottimizzazione basati sulla discesa del gradiente. Alcune delle funzioni di attivazione più usate sono:

- La tangente iperbolica *tanh* che mappa i valori di x nell'intervallo $[-1, 1]$:
$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.17}$$
- La funzione *sigmoide* che mappa i valori di x nell'intervallo $[0, 1]$:
$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.18}$$
- La funzione *Rectified Linear Unit (ReLU)* che filtra i valori di $x < 0$:
$$\text{ReLU}(x) = \max(0, x) \tag{2.19}$$

- La funzione *Leaky ReLU (LReLU)* che, rispetto alla ReLU, consente di avere un gradiente positivo basso quando l'unità non è attiva:

$$\text{LReLU}(x) = \max(x, 0.01x) \tag{2.20}$$

Come si evince dall'Equazione (2.16) le reti feedforward non sono altro che uno stack di modelli lineari separati da funzioni di attivazione g non lineari. L'architettura di una rete feedforward offre alcuni gradi di libertà, tra cui il numero e l'ampiezza degli hidden layer e le funzioni di attivazione da usare che, insieme, si aggiungono alla lista degli iperparametri. Le dimensioni del livello di input e output, invece, sono fisse e dipendono dal numero di features delle istanze $\boldsymbol{x}$ e del numero di classi k da predire. In Figura 2.6 è riportata un'istanza d'esempio dell'architettura descritta nell'Equazione (2.16) rappresentata come rete in cui ciascun elemento è connesso all'altro.

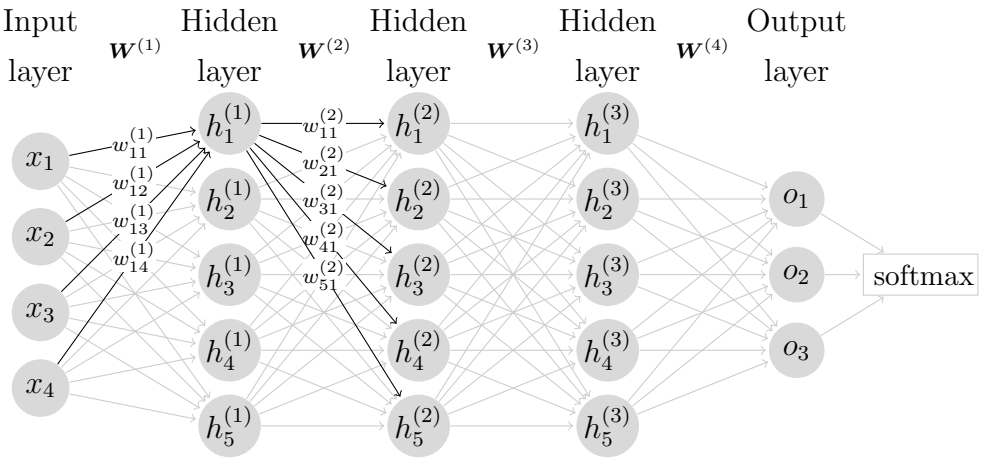


Figura 2.6: Rete MLP a cinque livelli: un livello di input che legge l'input $\boldsymbol{x}$ composto da quattro features, tre livelli nascosti di ampiezza 5 e il livello di output che stabilisce a quale tra le tre classi l'istanza di input $\boldsymbol{x}$ appartiene dopo aver applicato la funzione softmax. Ogni connessione tra gli elementi di un livello e del successivo rappresenta un peso da apprendere.

Nell'esempio, il primo elemento del primo livello nascosto si calcola come:

$$h_1^{(1)} = g^{(1)}(w_{11}^{(1)} \cdot x_1 + w_{12}^{(1)} \cdot x_2 + w_{13}^{(1)} \cdot x_3 + w_{14}^{(1)} \cdot x_4 + b_1^{(1)}) \tag{2.21}$$

Questo elemento sarà a sua volta usato nel calcolo di tutti gli elementi del secondo livello nascosto $h_1^{(2)} = g^{(2)}(w_{11}^{(2)} \cdot h_1^{(1)} + \cdots)$, $h_2^{(2)} = g^{(2)}(w_{21}^{(2)} \cdot h_1^{(1)} + \cdots)$ e così via. I livelli di un MLP sono anche chiamati *fully-connected* o *affine* perché tutti gli elementi sono connessi tra loro.

Nelle reti deep esiste un'ulteriore tecnica di regolarizzazione per ridurre l'overfitting nota come **dropout**. La tecnica consiste nel selezionare casualmente, con probabilità p, una parte dei pesi nella rete e non considerarli nella fase di aggiornamento dei parametri, facendo in modo tale che la rete non si stabilizzi su specifici pesi. Ad esempio, impostando il dropout a 0.2, un quinto dei pesi, scelti casualmente, non è aggiornato.

Nell'ambito delle reti feedforward esiste un **teorema di approssimazione universale**, quest'ultimo dimostra che data una funzione è possibile approssimarla tramite una rete feedforward con un solo livello nascosto abbastanza ampio. Non esiste, però, una procedura universale che dato un training set restituisce la funzione che generalizza tutti i possibili dati osservati e non, perciò il teorema resta solo un interessante risultato teorico. Da ciò segue che una rete feedforward con un singolo livello è teoricamente sufficiente per rappresentare qualunque funzione, ma questo livello deve essere indefinitamente grande e potrebbe anche generalizzare una funzione sbagliata a causa dell'overfitting. L'uso di modelli deep permette di ridurre l'ampiezza dei livelli nascosti e l'errore di generalizzazione.

Data l'enorme quantità di pesi e operazioni su di essi per il loro continuo aggiornamento, il gradiente può in alcuni casi diventare quasi nullo, avvicinandosi a 0, o diventare eccessivamente alto. Nel primo caso è difficile migliorare la loss function, mentre nel secondo caso l'apprendimento diventa instabile. Questo problema legato all'apprendimento dei parametri nelle reti deep è noto come **vanishing/exploding gradient problem**.

Per evitare problemi legati al gradiente è utile usare termini di regolarizzazione, come l_2, strategie di *batch-normalization* in cui, per ogni mini-batch, sono normalizzati gli input di ogni livello della rete in modo da avere media 0 e varianza 1 e usare particolari funzioni di attivazione, come ReLU.

2.3 Embedding Layer

Finora abbiamo sempre considerato i parametri Θ di una rete come valori numerici reali. Nel caso in cui dovessimo trattare elementi simbolici, come parole di un vocabolario, è utile associare ad ogni possibile feature simbolica un vettore reale di dimensione d in modo da poterlo usare all'interno della rete neurale e apprenderlo congiuntamente agli altri parametri Θ.

La trasformazione da feature simbolica a vettore d-dimensionale è ottenuta da un apposito livello nella rete, noto come **embedding layer**, una matrice $\boldsymbol{E} \in \mathbb{R}^{|V| \times d}$ dove ogni riga corrisponde alla codifica d-dimensionale di ciascun vocabolo distinto presente nel vocabolario V.

Esistono due modalità per rappresentare una feature simbolica in formato vettoriale:

- **one-hot encoding**. Ogni vettore ha la stessa grandezza del vocabolario, $d = |V|$, e ogni elemento del vettore corrisponde a una feature unica. Il vettore con codifica one-hot ha gli elementi tutti a 0 tranne la feature che corrisponde alla parola da rappresentare, che ha valore 1. Ad esempio il vettore per rappresentare la parola *casa* con una codifica one-hot, su un vocabolario di dimensione 10000, avrà un solo elemento a 1, quello corrispondente alla parola *casa* ed i restanti 9999 elementi a 0. Questo tipo di rappresentazione è sparsa e impiega molta memoria.

- **dense encoding**. La dimensione del vettore è di molto inferiore a quella del vocabolario V, tipicamente d non è più grande di 300. Per ogni feature simbolica $w_i \in V$ con $1 \leq i \leq |V|$ ci si calcola il rispettivo vettore $\vec{v}(w_i) \in \mathbb{R}^d$. Il beneficio di una rappresentazione *dense* è che features simili condividono vettori simili che possono essere combinati tra loro.

La codifica dense si può ottenere durante l'apprendimento stesso simultaneamente agli altri parametri. È anche possibile calcolare la codifica dense in maniera indipendente, attraverso appositi algoritmi, in modo da riutilizzarla in diverse reti. In questo caso parleremo di *feature embedding preaddestrate*.

Uno degli algoritmi più noti per ottenere una codifica dense, descritto in Mikolov et al. (2013a,b), è WORD2VEC che comprende due modelli: CBOW e Skip-gram. Entrambi sono fondati sull'ipotesi di semantica distribuzionale in cui si assume che il significato delle parole si possa derivare dal contesto in cui queste compaiono e che le parole che occorrono in contesti simili abbiano significati simili. CBOW predice una parola dato il contesto, mentre Skip-gram predice il contesto data la parola. Il contesto, in questo caso, non è altro che l'insieme delle parole che compaiono a sinistra e a destra della parola da predire.

Data una sequenza di T parole di training $w_1, \ldots, w_T$ e un contesto di dimensione c l'obiettivo di Skip-gram è di massimizzare la probabilità media:

$$\hat{\Theta} = \arg\max_{\Theta} \frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c} \log p_{\Theta}(w_{t+j}|w_t) \tag{2.22}$$

dove w_t è la parola centrale e il contesto è rappresentato dalle parole che compaiono a sinistra e destra. I parametri Θ sono i vettori di dimensione d da apprendere che rappresentano la codifica dense di ciascuna parola nel vocabolario V.

2.4 Recurrent Neural Network

Le **recurrent neural networks** (RNN) sono una particolare famiglia di reti neurali utilizzate per elaborare dati sequenziali la cui architettura è ottenuta aggiungendo connessioni cicliche a partire dal modello feedforward. Il motivo di introdurre un'architettura ciclica che segua l'ordine sequenziale dei dati (come la struttura di una frase) risiede nel fatto che i modelli feedforward non sono in grado di distinguere l'ordine dei dati e non hanno memoria dei precedenti input.

L'idea alla base delle RNN è quella di condividere i parametri Θ attraverso i vari stati in cui la rete si trova analizzando una sequenza di input. Il vettore di output $\hat{\boldsymbol{y}}^{(t)}$ di una RNN dipende dal t-esimo esempio di training $\boldsymbol{x}^{(t)}$, dallo

stato precedente $\boldsymbol{s}^{(t-1)}$ della rete e dai parametri condivisi Θ:

$$\hat{\boldsymbol{y}}^{(t)} = \text{RNN}_{\Theta}(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}) \tag{2.23}$$

In Figura 2.7 è rappresentata una RNN a partire dalla definizione ricorsiva appena data.

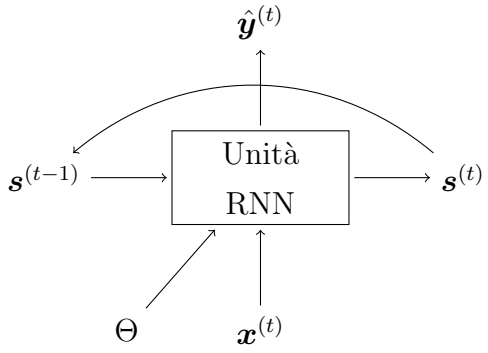


Figura 2.7: Rappresentazione ricorsiva di una rete RNN.

Poiché una RNN è definita ricorsivamente, per conoscere il t-esimo output della sequenza di input dobbiamo analizzare l'intera sequenza. Prendiamo ad esempio una sequenza di lunghezza $t = 3$ e assumiamo che $\hat{\boldsymbol{y}}^{(t)} = \boldsymbol{s}^{(t)}$, l'output della RNN sull'intera sequenza è dato da:

$$\begin{aligned} \hat{\boldsymbol{y}}^{(3)} &= \text{RNN}(\boldsymbol{s}^{(2)}, \boldsymbol{x}^{(3)}) \\ &= \text{RNN}(\text{RNN}(\boldsymbol{s}^{(1)}, \boldsymbol{x}^{(2)}), \boldsymbol{x}^{(3)}) \\ &= \text{RNN}(\text{RNN}(\text{RNN}(\boldsymbol{s}^{(0)}, \boldsymbol{x}^{(1)}), \boldsymbol{x}^{(2)}), \boldsymbol{x}^{(3)}) \end{aligned} \tag{2.24}$$

dove $\boldsymbol{s}^{(0)}$ è il vettore dello stato iniziale. In Figura 2.8 è rappresentata una rete RNN *unfolded* su una sequenza di tre input.

Esiste una formulazione equivalente al teorema di approssimazione universale per le reti RNN che dimostra la Turing-completezza: ogni funzione calcolabile da una macchina di Turing può essere calcolata da una RNN di grandezza finita.

L'applicazione di una sola RNN su una sequenza di input $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$ lunga n (scritta in maniera più compatta come $\boldsymbol{x}^{(1:n)}$) cattura solo le informazioni nell'ordine di lettura dell'input. Se volessimo catturare più informazioni

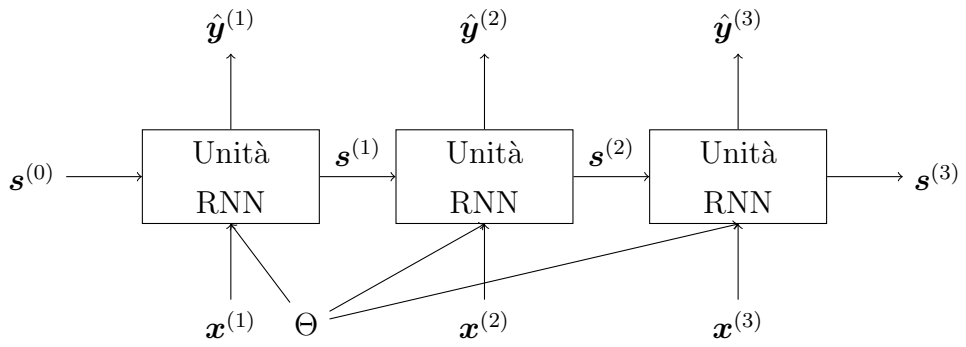


Figura 2.8: RNN unfolded applicata ad una sequenza di tre input. L'output finale $\hat{\boldsymbol{y}}^{(3)}$ dipende dagli stati e input precedenti della rete e dai parametri condivisi Θ.

sulla sequenza, a prescindere dalla direzione di lettura, possiamo utilizzare una seconda RNN che legge la sequenza di input nel verso opposto della prima. Le **RNN bidirezionali** combinano l'output di due RNN, una RNN_f e RNN_b che leggono la sequenza di input in direzione opposta. La RNN_f legge la sequenza nell'ordine $\boldsymbol{x}^{(1:t)}$, mentre la RNN_b nell'ordine $\boldsymbol{x}^{(t:1)}$:

$$\begin{aligned}
\hat{\boldsymbol{y}}_f^{(t)} &= \text{RNN}_f(\boldsymbol{s}_f^{(t-1)}, \boldsymbol{x}^{(t)}) \\
\hat{\boldsymbol{y}}_b^{(t)} &= \text{RNN}_b(\boldsymbol{s}_b^{(t-1)}, \boldsymbol{x}^{(n-t+1)}) \\
\hat{\boldsymbol{y}}^{(t)} &= \text{BiRNN}(\boldsymbol{x}^{(t)}) = [\hat{\boldsymbol{y}}_f^{(t)}; \hat{\boldsymbol{y}}_b^{(t)}]
\end{aligned} \tag{2.25}$$

dove $1 \leq t \leq n$ e n è la lunghezza della sequenza. L'output della RNN bidirezionale $\hat{\boldsymbol{y}}^{(t)}$ è la combinazione delle due predizioni $\hat{\boldsymbol{y}}_f^{(t)}$ e $\hat{\boldsymbol{y}}_b^{(t)}$.

Come accade per le reti feedforward, possiamo comporre diversi livelli di RNN in modo da formare RNN deep. Per costruire una RNN deep di profondità k si procede assegnando l'output di una RNN come input della RNN al livello superiore:

$$\begin{aligned}
\hat{\boldsymbol{y}}_1^{(t)} &= \text{RNN}^{(1)}(\boldsymbol{s}_1^{(t-1)}, \boldsymbol{x}^{(t)}) \\
\hat{\boldsymbol{y}}_2^{(t)} &= \text{RNN}^{(2)}(\boldsymbol{s}_2^{(t-1)}, \hat{\boldsymbol{y}}_1^{(t)}) \\
&\vdots \\
\hat{\boldsymbol{y}}^{(t)} = \hat{\boldsymbol{y}}_k^{(t)} &= \text{RNN}^{(k)}(\boldsymbol{s}_k^{(t-1)}, \hat{\boldsymbol{y}}_{k-1}^{(t)})
\end{aligned} \tag{2.26}$$

Dalle definizioni date finora, le RNN sono in grado di apprendere come mappare una sequenza di input $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$ in una sequenza etichettata $\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(n)}$, dove la lunghezza è fissata a n per entrambe le sequenze. In alcuni casi può essere utile riuscire ad apprendere come mappare tra loro due sequenze di dimensione diversa, l'architettura che ci permette di fare ciò è nota col nome di **encoder-decoder** o **sequence-to-sequence**.

Prima di parlare di questa architettura è necessario introdurre il concetto di generazione condizionata. Un generatore RNN condizionato è una rete RNN che, data una sequenza, impara a generare il successivo elemento $\boldsymbol{t}^{(j+1)}$ a partire dagli elementi precedentemente generati $\hat{\boldsymbol{t}}^{(1)}, \ldots, \hat{\boldsymbol{t}}^{(j)}$ come probabilità condizionata:

$$\hat{\boldsymbol{t}}^{(j+1)} \sim p(\boldsymbol{t}^{(j+1)} \mid \hat{\boldsymbol{t}}^{(1:j)}) \tag{2.27}$$

A partire da una sequenza di input $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(n)}$, i modelli sequence-to-sequence, ricavano un *contesto* $\boldsymbol{c}$ di dimensione fissa tramite una RNN chiamata *encoder*. Il contesto $\boldsymbol{c}$, solitamente rappresentato dallo stato finale $\boldsymbol{s}^{(n)}$ della RNN encoder, è usato come stato iniziale di una RNN generatore, chiamata *decoder*, che impara a generare l'elemento $\boldsymbol{y}^{(j)}$ dati i precedenti elementi generati dal decoder $\hat{\boldsymbol{y}}^{(1)}, \ldots, \hat{\boldsymbol{y}}^{(j-1)}$, condizionati dalla sequenza di input dell'encoder attraverso il contesto $\boldsymbol{c}$:

$$\begin{aligned} \boldsymbol{c} &= \mathrm{RNN}_{\mathrm{Enc}}(\boldsymbol{s}^{(n-1)}, \boldsymbol{x}^{(n)}) \\ \hat{\boldsymbol{y}}^{(j+1)} &\sim p(\boldsymbol{y}^{(j+1)} \mid \hat{\boldsymbol{y}}^{(1:j)}, \boldsymbol{c}) \end{aligned} \tag{2.28}$$

con $1 \leq j \leq m$, dove m è la lunghezza della sequenza da predire. In questo modo siamo in grado di generare sequenze di lunghezza arbitraria m a partire da sequenze di lunghezza n attraverso l'uso del contesto $\boldsymbol{c}$.

Quando il contesto $\boldsymbol{c}$ è piccolo potremmo non riuscire a catturare correttamente sequenze troppo lunghe nonostante non ci siano limitazioni circa la dimensione degli stati per l'encoder e il decoder. Una possibile soluzione è quella di usare una sequenza di contesti $\boldsymbol{c}^{(1)}, \ldots, \boldsymbol{c}^{(n)}$ prodotti da una RNN encoder bidirezionale:

$$\boldsymbol{c}^{(i)} = \mathrm{BiRNN}_{\mathrm{Enc}}(\boldsymbol{x}^{(i)}) \tag{2.29}$$

Il decoder, quindi, usa la sequenza di contesti come memoria e ad ogni passo j del processo di generazione sceglie su quale degli n contesti concentrarsi, questo meccanismo è noto come **attention**. L'obiettivo è di imparare a scegliere quale tra gli n contesti è utile per generare la sequenza di output desiderata. Solitamente il contesto usato al passo j di generazione si ottiene come somma pesata di tutti contesti $\boldsymbol{c}^{(1)}, \ldots, \boldsymbol{c}^{(n)}$ prodotti dall'encoder:

$$\tilde{\boldsymbol{c}}^{(j)} = \sum_{i=1}^{n} \alpha_i^{(j)} \cdot \boldsymbol{c}^{(i)} \tag{2.30}$$

dove $\boldsymbol{\alpha}^{(j)} \in \mathbb{R}^n$ è il vettore di attention al j-esimo passo di generazione. Il processo di generazione, dunque, risulta condizionato dal contesto $\tilde{\boldsymbol{c}}^{(j)}$:

$$\hat{\boldsymbol{y}}^{(j+1)} \sim p(\boldsymbol{y}^{(j+1)} \mid \hat{\boldsymbol{y}}^{(1:j)}, \tilde{\boldsymbol{c}}^{(j)}) \tag{2.31}$$

Nel caso in cui $\tilde{\boldsymbol{c}}^{(j)} = \boldsymbol{c}^{(n)}$ stiamo considerando un semplice modello sequence-to-sequence, senza il meccanismo di attenzione, poiché prendiamo solo l'ultimo contesto prodotto dall'encoder.

Le RNN, se applicate a sequenze molto lunghe, possono avere difficoltà nel catturare dipendenze tra elementi troppo distanti tra loro. Il motivo è che durante l'apprendimento il gradiente diminuisce rapidamente e non riesce a conservare informazioni per un lungo periodo. Per poter far fronte a questo problema sono state introdotte le cosiddette **RNN gated**.

Le RNN gated controllano gli accessi alla memoria per ogni singola unità RNN in modo da ridurre quanto più possibile la sovrascrittura di informazioni che possono essere rilevanti. L'idea è quella di introdurre un vettore *gate* $\boldsymbol{g}$, i cui valori sono compresi in $[0, 1]$, che determina l'accesso solo ad alcuni elementi dello stato $\boldsymbol{s}$. Un'unità RNN gated riceve in input uno stato $\boldsymbol{s}^{(t-1)}$ e un elemento della sequenza di training $\boldsymbol{x}^{(t)}$ e calcola il nuovo stato come[1]:

$$\boldsymbol{s}^{(t)} = \boldsymbol{g} \odot \boldsymbol{x}^{(t)} + (\mathbf{1} - \boldsymbol{g}) \odot \boldsymbol{s}^{(t-1)} \tag{2.32}$$

Più l'elemento $g_i \in \boldsymbol{g}$ è vicino a 1 più il corrispettivo elemento contribuisce al calcolo dello stato successivo, più g_i si avvicina a 0 più l'elemento

[1] L'operatore $\odot$ è il prodotto di hadamard. L'operazione $\boldsymbol{x} = \boldsymbol{u} \odot \boldsymbol{v}$ si calcola $x_i = u_i \cdot v_i$

corrispettivo non influenza lo stato successivo. I valori del gate sono appresi simultaneamente agli altri parametri in fase di apprendimento. Il meccanismo di controllo con gate che decide quali parti dell'input saranno ricordate ($g_i \to 1$) e quali dimenticate ($g_i \to 0$) è alla base delle architetture **long short-term memory** (**LSTM**) e **gated recurrent unit** (**GRU**).

Le reti RNN basate su unità LSTM calcolano lo stato successivo $\boldsymbol{s}^{(t)}$ e l'output $\hat{\boldsymbol{y}}^{(t)}$ come segue:

$$\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma(\boldsymbol{U}^f \boldsymbol{x}^{(t)} + \boldsymbol{W}^f \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^f) \\
\boldsymbol{i}^{(t)} &= \sigma(\boldsymbol{U}^i \boldsymbol{x}^{(t)} + \boldsymbol{W}^i \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^i) \\
\boldsymbol{o}^{(t)} &= \sigma(\boldsymbol{U}^o \boldsymbol{x}^{(t)} + \boldsymbol{W}^o \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^o) \\
\boldsymbol{z}^{(t)} &= \tanh(\boldsymbol{U}^z \boldsymbol{x}^{(t)} + \boldsymbol{W}^z \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^z) \\
\boldsymbol{s}^{(t)} &= \boldsymbol{f}^{(t)} \odot \boldsymbol{s}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \boldsymbol{z}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \boldsymbol{o}^{(t)} \odot \tanh(\boldsymbol{s}^{(t)})
\end{aligned} \tag{2.33}$$

Il controllo delle parti da dimenticare dello stato è a carico della *forget gate* $\boldsymbol{f}$ con i relativi parametri $\boldsymbol{b}^f$, $\boldsymbol{U}^f$, $\boldsymbol{W}^f$. L'*input gate* $\boldsymbol{i}$, con parametri $\boldsymbol{b}^i$, $\boldsymbol{U}^i$, $\boldsymbol{W}^i$, si occupa delle parti dello stato che l'unità deve ricordare. L'*output gate*, con i relativi parametri $\boldsymbol{b}^o$, $\boldsymbol{U}^o$, $\boldsymbol{W}^o$, decide quali parti del nuovo stato $s^{(t)}$ considerare per restituire l'output della cella. Tutte le tipologie di gate precedenti sono una combinazione lineare dell'input $x^{(t)}$ e lo stato precedente $s^{(t-1)}$ a cui è applicata una funzione di attivazione sigmoide σ. Il vettore $\boldsymbol{z}^{(t)}$ è costituito dagli elementi candidati che potrebbero essere considerati per il calcolo del nuovo stato, ottenuto dall'operazione tra forget gate e il precedente stato, per decidere quali parti dimenticare e l'operazione tra input gate e elementi candidati, per decidere quali parti ricordare.

L'altra unità RNN gated è GRU, la quale è simile alla LSTM ma fa uso di meno gate combinando il *forget gate* e l'*input gate* in un singolo *update*

gate $\boldsymbol{u}$:

$$\begin{aligned}
\boldsymbol{u}^{(t)} &= \sigma(\boldsymbol{U}^u \boldsymbol{x}^{(t)} + \boldsymbol{W}^u \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^u) \\
\boldsymbol{r}^{(t)} &= \sigma(\boldsymbol{U}^r \boldsymbol{x}^{(t)} + \boldsymbol{W}^r \boldsymbol{s}^{(t-1)} + \boldsymbol{b}^r) \\
\tilde{\boldsymbol{s}}^{(t)} &= \tanh(\boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}(\boldsymbol{r}^{(t)} \odot s^{(t-1)})) \\
\boldsymbol{s}^{(t)} &= \boldsymbol{u}^{(t)} \odot \tilde{\boldsymbol{s}}^{(t)} + (\boldsymbol{1} - \boldsymbol{u}^{(t)}) \odot \boldsymbol{s}^{(t-1)} \\
\hat{\boldsymbol{y}}^{(t)} &= \boldsymbol{s}^{(t)}
\end{aligned} \tag{2.34}$$

In contrasto all'update gate c'è il *reset gate* $\boldsymbol{r}$ che controlla quali parti dello stato usare per calcolare il successivo stato.

Capitolo 3

Neural Dependency Parser

In questo capitolo saranno introdotti alcuni dependency parser basati su reti neurali deep le cui prestazioni rappresentano, o sono molto vicine, allo stato dell'arte. I parser che utilizzano modelli di reti neurali sono anche noti in letteratura come **neural dependency parser**. Dopo aver discusso brevemente la loro architettura si procede alla descrizione degli esperimenti. Si analizzano prima i treebank in lingua italiana disponibili nelle Universal Dependencies, in seguito si descrivono le configurazioni usate per apprendere i modelli di parsing. Infine sono riportati, discussi e confrontati i risultati delle valutazioni.

3.1 Modelli di Neural Dependency Parsing

In **Chen and Manning (2014)** è proposto, per la prima volta, di rappresentare le parole, POS tag e tipi di dipendenza attraverso una codifica densa usando vettori di embedding. In questo modo è possibile automatizzare il processo di apprendimento delle features evitando di estrarle seguendo template progettati manualmente.

Il parser è sviluppato sull'approccio transition-based e apprende un classificatore basato su una rete neurale che si occupa di scegliere la transizione

corretta usando il sistema arc-standard. L'algoritmo di parsing usato dal parser è di tipo greedy deterministico.

Ciascuna parola w_i, POS tag t_j e tipo di dipendenza l_k è rappresentata dal rispettivo vettore di embedding $\boldsymbol{e}_i^w, \boldsymbol{e}_j^t, \boldsymbol{e}_k^l$. La rete neurale scelta è di tipo feedforward (MLP) con un solo hidden layer, dunque con tre livelli totali in cui l'input layer è rappresentato dal vettore di features $[\boldsymbol{x}^w, \boldsymbol{x}^t, \boldsymbol{x}^l]$. Il vettore $\boldsymbol{x}^w$ è composto da 18 vettori di embedding relativi alle parole presenti in una determinata posizione dello stack o del buffer di una configurazione, dunque $\boldsymbol{x}^w = [\boldsymbol{e}_1^w; \ldots; \boldsymbol{e}_{18}^w]$. Lo stesso vale per il vettore $\boldsymbol{x}^t = [\boldsymbol{e}_1^t; \ldots; \boldsymbol{e}_{18}^t]$, che comprende 18 POS embedding, e il vettore delle $\boldsymbol{x}^l = [\boldsymbol{e}_1^l; \ldots; \boldsymbol{e}_{12}^l]$, costituito da 12 vettori di embedding del tipo di dipendenza. Il livello nascosto $\boldsymbol{h}$ è una combinazione lineare dell'input layer a cui viene applicata una funzione di attivazione cubica $g(x) = x^3$ ed è fatto confluire in un livello softmax che predice la transizione con probabilità più alta:

$$\begin{aligned} \boldsymbol{h} &= (\boldsymbol{W}_1^w \boldsymbol{x}^w + \boldsymbol{W}_1^t \boldsymbol{x}^t + \boldsymbol{W}_1^l \boldsymbol{x}^l + \boldsymbol{b}_1)^3 \\ \boldsymbol{p} &= \text{softmax}(\boldsymbol{W}_2 \boldsymbol{h}) \end{aligned} \tag{3.1}$$

La funzione di attivazione cubica, al posto di attivazioni classiche come tanh e sigmoid, è adatta a catturare le interazioni tra i tre elementi considerati nell'apprendimento del modello: parole, POS tags e tipi di dipendenza.

Il training set $\mathcal{D}^{(\text{train})} = \{(c_i, t_i)\}$ è composto dalle coppie configurazione-transizione e non fa uso di template per trasformare la configurazione. I parametri della rete sono appresi minimizzando la cross-entropy loss, usando regolarizzazione l_2, tramite l'algoritmo di ottimizzazione AdaGrad con mini-batch.

In **Dyer et al. (2015)** viene proposto un parser transition-based basato sul sistema arc-standard. In questo modello si cerca di apprendere la rappresentazione dell'intero stato del parser, ottenuta a partire dalla rappresentazione del buffer, dello stack e della storia delle azioni intraprese dal parser. Per rappresentare l'intero stato si utilizza una tecnica che gli autori chiamano *stack LSTM*, basata su RNN di tipo LSTM e supporta azioni di

push e pop proprio come gli elementi della configurazione del parser (stack e buffer).

L'idea è quella di rappresentare lo stack $S = (w_1, \ldots, w_n)$ attraverso lo stato finale della RNN applicata alla sequenza di parole $w_1, \ldots, w_n$ contenuta nello stack. Il modello utilizza tre strutture *stack LSTM* per ogni elemento della configurazione: una per lo stack S, una per il buffer B e l'altra per la storia delle azioni A. Ad ogni passo t il parser usa le rappresentazioni $\boldsymbol{s}_t, \boldsymbol{b}_t, \boldsymbol{a}_t$ degli elementi S, B e A per determinare la transizione da applicare. La rappresentazione dello stato del parser al passo t, denominata $\boldsymbol{p}_t$, è definita come

$$\boldsymbol{p}_t = \max\{0, \boldsymbol{W}[\boldsymbol{s}_t; \boldsymbol{b}_t; \boldsymbol{a}_t] + \boldsymbol{d}\} \tag{3.2}$$

Ogni token $\boldsymbol{x}$ relativo a un qualunque elemento di una configurazione, dato in input alla RNN, è rappresentato concatenando il vettore di embedding $\boldsymbol{w}$ della parola ed il vettore di embedding $\boldsymbol{t}$ relativo al POS tag della stessa:

$$\boldsymbol{x} = \max\{0, \boldsymbol{V}[\boldsymbol{w}; \boldsymbol{t}] + \boldsymbol{b}\} \tag{3.3}$$

Per rappresentare gli archi dei dependency tree parziali in A si usa un vettore $\boldsymbol{c}$ ottenuto dalla composizione dei vettori di embedding $\boldsymbol{h}, \boldsymbol{d}, \boldsymbol{r}$, che indicano rispettivamente la testa, la dipendente e il tipo di relazione:

$$\boldsymbol{c} = \tanh(\boldsymbol{U}[\boldsymbol{h}; \boldsymbol{d}; \boldsymbol{r}] + \boldsymbol{e}] \tag{3.4}$$

Data la rappresentazione $\boldsymbol{z}$ della sequenza corretta di transizioni e della frase di input $\boldsymbol{s}$, il parser cerca di minimizzare la negative conditional log-likelihood:

$$p(\boldsymbol{z}|\boldsymbol{s}) = -\sum_{t=1}^{|\boldsymbol{z}|} \log p(z_t|\boldsymbol{p}_t) \tag{3.5}$$

dove $p(z_t|\boldsymbol{p}_t)$ è la probabilità di compiere una specifica transizione z_t data la rappresentazione dello stato $\boldsymbol{p}_t$. I parametri della rete sono appresi tramite l'algoritmo di ottimizzazione SGD senza mini-batch e con un fattore di regolarizzazione l_2. L'algoritmo di parsing adottato può utilizzare una beam search.

In **Ballesteros et al. (2015)** viene proposta una modifica alla rappresentazione dell'Equazione (3.3), il vettore di embedding $\boldsymbol{w}$ della parola è sostituito da una rappresentazione basata sui singoli caratteri da cui è composta la parola, tramite l'applicazione di una LSTM bidirezionale. Data una parola, denominiamo $\overrightarrow{\boldsymbol{w}}$ il vettore dello stato finale della RNN che legge i caratteri da sinistra a destra e $\overleftarrow{\boldsymbol{w}}$ lo stato finale della RNN che legge i caratteri della stessa in senso opposto. La rappresentazione di un token $\boldsymbol{x}$ in (3.3) è ridefinita come:

$$\boldsymbol{x} = \max\{0, \boldsymbol{V}[\overrightarrow{\boldsymbol{w}}; \overleftarrow{\boldsymbol{w}}; \boldsymbol{t}] + \boldsymbol{b}\} \tag{3.6}$$

Nella modifica, inoltre, è introdotta una transizione aggiuntiva al sistema arc-standard che permette di produrre dependency tree che siano anche non-projective.

In **Kiperwasser and Goldberg (2016)** si segue l'idea precedente di rappresentare l'intero stato del parser usando, però, una LSTM bidirezionale deep con k livelli al posto dello stack LSTM. Il parser è sviluppato in due versioni: la prima basata sull'approccio transition-based con sistema arc-hybrid e oracolo dinamico, la seconda basata sull'approccio graph-based con modello arc-factored.

Data una frase di n parole $w_1, \ldots, w_n$ con i relativi POS tags $t_1, \ldots, t_n$, ad ogni parola w_i e POS tag t_i sono associati i vettori di embedding $\boldsymbol{e}_i^w$ e $\boldsymbol{e}_i^t$, usati per rappresentare la sequenza di input $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ e calcolare il contesto $\boldsymbol{v}_i$ come:

$$\begin{aligned} \boldsymbol{x}_i &= [\boldsymbol{e}_i^w; \boldsymbol{e}_i^t] \\ \boldsymbol{v}_i &= \text{BiLSTM}(\boldsymbol{x}_i) \end{aligned} \tag{3.7}$$

Le features $\boldsymbol{v}_i$ sono usate per ottenere una rappresentazione ϕ che sarà usata come input per un classificatore MLP, con un solo livello nascosto, che ne calcolerà lo score:

$$\text{MLP}(\phi(x)) = \boldsymbol{W}_2[\tanh(\boldsymbol{W}_2\phi(x) + \boldsymbol{b}_1)] + \boldsymbol{b}_2 \tag{3.8}$$

Nel caso del modello transition-based la rete MLP apprende lo score di una transizione data la rappresentazione $\phi(c)$ di una configurazione c, otte-

nuta combinando i vettori di embedding $\boldsymbol{v}_i$ associati alle prime tre parole in cima allo stack e alla prima sul buffer:

$$\phi(c) = [\boldsymbol{v}_{\sigma_3}; \boldsymbol{v}_{\sigma_2}; \boldsymbol{v}_{\sigma_1}; \boldsymbol{v}_{\beta_1}] \tag{3.9}$$

Nel caso del modello graph-based il classificatore MLP apprende lo score di ogni singolo arco (h, d) combinando il vettore di embedding $\boldsymbol{v}_i$ della testa h e della dipendente d:

$$\phi(h, d) = [\boldsymbol{v}_h; \boldsymbol{v}_d] \tag{3.10}$$

L'algoritmo di parsing usato nel caso transition-based è di tipo greedy deterministico, mentre nel caso graph-based è usato Eisner. I parametri sono appresi minimizzando la hinge loss tramite l'algoritmo di ottimizzazione Adam.

In **Andor et al. (2016)** viene proposto un parser transition-based con sistema arc-standard basato su una rete con architettura feed-forward. I sistemi di transizione sono caratterizzati da una sequenza di configurazioni $c_1, \ldots c_j$ con le relative transizioni $t_1, \ldots, t_j$. L'idea alla base del modello è di assumere che ci sia una relazione univoca tra la sequenza di transizioni $t_1, \ldots, t_{j-1}$ e lo stato c_j. In altre parole si assume che uno stato codifichi l'intera storia delle transizioni. L'obiettivo è quello di apprendere, tramite una rete feed-forward, la funzione $\mathrm{s}(t_{1:j-1}, t_j)$ che calcoli lo score della transizione successiva t_j valida per c, data la sequenza di transizioni precedenti $t_{1:j-1}$ (che si assume codifichi la configurazione c).

Sia $\mathcal{T}_n$ l'insieme delle transizioni valide lunghe n, il modello cerca trovare la soluzione al problema di ottimizzazione

$$\arg\max_{t_{1:n} \in \mathcal{T}_n} p_G(t_{1:n}) = \arg\max_{t_{1:n} \in \mathcal{T}_n} \sum_{j=1}^{n} \mathrm{s}(t_{1:j-1}, t_j) \tag{3.11}$$

dove p_G definisce una distribuzione di probabilità CRF (Conditional Random Field) della sequenza di transizioni $t_{1:n}$:

$$p_G(t_{1:n}) = \frac{\exp \sum_{j=1}^{n} \mathrm{s}(t_{1:j-1}, t_j)}{\sum_{t'_{1:n} \in \mathcal{T}_n} \exp \sum_{j=1}^{n} \mathrm{s}(t'_{1:j-1}, t'_j)} \tag{3.12}$$

Per approssimare l'arg max viene utilizzata una beam search in modo da rendere trattabile l'apprendimento. I parametri sono appresi minimizzando la CRF loss e usando come algoritmo di ottimizzazione SGD con la variante Momentum.

In **Cheng et al. (2016)** è proposto un parser graph-based di tipo arc-factored che fa uso di una RNN bidirezionale con meccanismo di attenzione per analizzare la frase.

La rappresentazione di ogni parola w_i è ottenuta a partire dalla combinazione dei vettori one-hot $\boldsymbol{e}_i$ degli attributi della parola a cui è applicata una funzione di attivazione LReLU con peso 0.1:

$$\boldsymbol{x}_i = \text{LReLU}[\boldsymbol{P}(\boldsymbol{E}^{\text{pos}}\boldsymbol{e}_i^{\text{pos}} + \boldsymbol{E}^{\text{form}}\boldsymbol{e}_i^{\text{form}} + \boldsymbol{E}^{\text{lemma}}\boldsymbol{e}_i^{\text{lemma}} + \ldots)] \tag{3.13}$$

Lo stato della RNN è dato da $\boldsymbol{h}_j = \text{GRU}(\boldsymbol{h}_{j-1}, \boldsymbol{x}_j)$, applicato in entrambe le direzioni della frase in modo da ottenere $\boldsymbol{h}_j^l$ e $\boldsymbol{h}_j^r$. Il tipo di cella RNN usata per le componenti è la GRU con funzione di attivazione LReLU invece della tanh. Nel modello si preferisce la codifica one-hot piuttosto che quella densa perché, in questo modo, si evita di stabilire la dimensione di embedding.

Il parser è composto da tre componenti: memoria, query destra-sinistra e query sinistra-destra. Data una frase $S = w_0 w_1 \ldots w_n$ il parser costruisce gli elementi $\boldsymbol{m}_0, \boldsymbol{m}_1, \ldots, \boldsymbol{m}_n$ della componente memoria combinando gli stati ottenuti dall'applicazione della RNN sulla frase in entrambi i versi, dunque $\boldsymbol{m}_j = [\boldsymbol{h}_j^l, \boldsymbol{h}_j^r]$.

Ogni elemento $\boldsymbol{q}_t$ delle componenti di query è usato per interrogare gli elementi $\boldsymbol{m}_j$ della componente memoria e restituire uno score $a_{t,j}$. Lo score indica il peso dell'attenzione relativo all'arco composto dalla parola in posizione t (dipendente) e da quella in posizione j (testa) per $t = 1, \ldots, n$ e $j = 0, \ldots, n$, ed è calcolato come:

$$a_{t,j} = \text{softmax}(\boldsymbol{V} \tanh(\boldsymbol{C}\boldsymbol{m}_j + \boldsymbol{D}\boldsymbol{q}_t)) \tag{3.14}$$

Si definisce il *soft* embedding degli elementi della componente memoria, similmente al meccanismo di attenzione, come $\tilde{\boldsymbol{m}}_t = \sum_{j=1}^{n} a_{t,j}\boldsymbol{m}_j$ e si calcolano

gli elementi delle componenti di query come $\boldsymbol{q}_t = \text{GRU}(\boldsymbol{q}_{t-1}, [\tilde{\boldsymbol{m}}_t; \boldsymbol{x}_t])$. Le rappresentazioni delle due componenti query, in entrambi i versi, sono passate a una rete MLP con un solo hidden layer che restituisce la probabilità delle relazioni testa-dipendente come:

$$\boldsymbol{y}_t = \text{softmax}(\boldsymbol{U}[\tilde{\boldsymbol{m}}_t^l; \tilde{\boldsymbol{m}}_t^r] + \boldsymbol{W}[\boldsymbol{q}_t^l; \boldsymbol{q}_t^r]) \tag{3.15}$$

dove $y_{t,1}, \ldots, y_{t,m}$ sono le probabilità di tutte le m possibili relazioni. Analizzando tutte le combinazioni testa-dipendente si possono catturare implicitamente informazioni graph-based di ordine superiore al primo, pur usando un modello di tipo arc-factored. L'algoritmo di parsing utilizzato è Chu-Liu/Edmonds e i parametri della rete sono appresi minimizzando la cross-entropy loss tramite l'algoritmo di ottimizzazione Adam.

In **Dozat and Manning (2017); Dozat et al. (2017)** è proposto un parser graph-based di tipo arc-factored e basato su quello di Kiperwasser and Goldberg (2016). La differenza è che al posto di utilizzare una rete MLP affine ne usa una *biaffine*.

Nell'approccio affine lo stato $\boldsymbol{v}_i \in \mathbb{R}^d$ dell'Equazione (3.7) è usato per calcolare lo score $\boldsymbol{s}_i$ di un arco tramite una trasformazione lineare $\boldsymbol{s}_i = \boldsymbol{W}\boldsymbol{v}_i + \boldsymbol{b}$ dove $\boldsymbol{W} \in \mathbb{R}^{n \times d}$ e $\boldsymbol{b} \in \mathbb{R}^n$. Nell'architettura biaffine la trasformazione si ottiene introducendo il prodotto di due matrici $\boldsymbol{HW} \in \mathbb{R}^{d \times d}$ al posto della singola matrice $\boldsymbol{W}$ e il prodotto $\boldsymbol{Hb} \in \mathbb{R}^d$ al posto del bias $\boldsymbol{b}$.

Prima di utilizzare la trasformazione biaffine, lo stato $\boldsymbol{v}_i$ viene dato in input a diverse MLP con tre livelli al fine di eliminare informazioni irrilevanti. Allo stato generato dall'applicazione delle MLP si applica la trasformazione biaffine per predire la testa $\hat{y}_i^{(\text{arc})}$ della parola i come:

$$\begin{aligned}
\boldsymbol{h}_i^{(\text{arc-dep})} &= \text{MLP}^{(\text{arc-dep})}(\boldsymbol{v}_i) \\
\boldsymbol{h}_i^{(\text{arc-head})} &= \text{MLP}^{(\text{arc-head})}(\boldsymbol{v}_i) \\
\boldsymbol{s}_i^{(\text{arc})} &= \boldsymbol{H}^{(\text{arc-head})}\boldsymbol{W}^{(\text{arc})}\boldsymbol{h}_i^{(\text{arc-dep})} \\
&\quad + \boldsymbol{H}^{(\text{arc-head})}\boldsymbol{b}^{(\text{arc})} \\
\hat{y}_i^{(\text{arc})} &= \arg\max_j s_{ij}^{(\text{arc})}
\end{aligned} \tag{3.16}$$

Dopo aver predetto la testa si sceglie il tipo di dipendenza $\hat{y}_i^{(\text{rel})}$ come segue:

$$\begin{aligned}
\boldsymbol{h}_i^{(\text{rel-dep})} &= \text{MLP}^{(\text{rel-dep})}(\boldsymbol{v}_i) \\
\boldsymbol{h}_i^{(\text{rel-head})} &= \text{MLP}^{(\text{rel-head})}(\boldsymbol{v}_i) \\
\boldsymbol{s}_i^{(\text{rel})} &= \boldsymbol{h}_{\hat{y}_i^{(\text{arc})}}^{\top(\text{rel-head})} \boldsymbol{U}^{(\text{rel})} \boldsymbol{h}_i^{(\text{rel-dep})} \\
&\quad + \boldsymbol{W}^{(\text{rel})}(\boldsymbol{h}_i^{(\text{rel-dep})} \oplus \boldsymbol{h}_{\hat{y}_i^{(\text{arc})}}^{(\text{rel-head})}) + \boldsymbol{b}^{(\text{rel})} \\
\hat{y}_i^{(\text{rel})} &= \arg\max_j s_{ij}^{(\text{rel})}
\end{aligned} \tag{3.17}$$

L'algoritmo MST di parsing scelto è Chu-Liu/Edmonds. I parametri sono appresi addestrando congiuntamente i due classificatori biaffine e minimizzando la somma della cross-entropy loss.

In **Shi et al. (2017b,a)** è proposto un parser che combina una rappresentazione compatta delle features di tre paradigmi di parsing diversi: i due sistemi arc-hybrid e arc-eager dell'approccio transition-based e il modello arc-factored per l'approccio graph-based. Per ogni frase sono ricavati, attraverso l'uso di una rete LSTM bidirezionale, i vettori di embedding di ciascuna parola a partire dalla rappresentazione a livello di caratteri, già citata in Ballesteros et al. (2015).

Il modello graph-based apprende lo score degli archi seguendo l'architettura deep biaffine, precedentemente discussa e proposta in Dozat and Manning (2017). I due modelli transition-based condividono le stesse configurazioni e lo score è calcolato utilizzando un sistema di deduzione che apprende il modello congiunto, con assioma iniziale $[0, 1]$ e stato finale $[0, n + 1]$. La prova con il più alto score del sistema di deduzione che porta alla configurazione $[0, n + 1]$ costituisce la sequenza di transizioni predetta. I parametri sono appresi minimizzando la hinge loss tramite l'algoritmo di ottimizzazione Adam.

In **Nguyen et al. (2017)**, è proposto un modello di rete neurale in grado di apprendere congiuntamente sia il POS tagging che il dependency parsing graph-based di tipo arc-factored. L'idea alla base è che più il POS tags è accurato più le performance del parsing migliorano e, viceversa, la struttura dei dependency tree può risolvere alcune ambiguità di POS tagging.

Data una frase in input $w_1, \ldots, w_n$ di n parole, l'embedding di ciascuna parola w_i è costruito concatenando il vettore di word embedding $\boldsymbol{e}_{w_i}$ al vettore $\boldsymbol{e}^c_{w_i}$, ottenuto dall'embedding della sua rappresentazione in caratteri, come in Ballesteros et al. (2015):

$$\boldsymbol{e}_i = [\boldsymbol{e}_{w_i}, \boldsymbol{e}^c_{w_i}] \tag{3.18}$$

La i-esima parola w_i è rappresentata dal vettore $\boldsymbol{v}_i$ ottenuto come $\boldsymbol{v}_i = \text{BiLSTM}(\boldsymbol{e}_i)$. Lo score dell'arco con testa w_i e dipendente w_j è calcolato come:

$$\text{score}(w_i, w_j) = \text{MLP}([\boldsymbol{v}_{w_i}; \boldsymbol{v}_{w_j}]) \tag{3.19}$$

La funzione di loss L_{arc} usata per apprendere i parametri del task di parsing è la hinge loss. Per il POS tagging la sequenza di tag è rappresentata dallo stato ottenuto applicando una LSTM bidirezionale sulla sequenza dei tag e la funzione di loss $L_{\text{pos}}(\hat{\boldsymbol{t}}, \boldsymbol{t})$ da minimizzare è la cross-entropy, dove $\hat{\boldsymbol{t}}$ è la sequenza dei POS tags predetti e $\boldsymbol{t}$ quella reale.

L'algoritmo di parsing MST è Eisner e i parametri del modello sono appresi minimizzando la somma delle due loss function L_{pos} e L_{arc} tramite l'algoritmo di ottimizzazione Adam.

In Tabella 3.1 sono riportate le abbreviazioni (prima colonna) con cui si farà riferimento in seguito, per questioni di impaginazione, agli articoli dei parser descritti finora. Nella tabella sono anche riportati i link alle repository che contengono le implementazioni dei parser sviluppate dagli stessi autori. Per quanto riguarda il parser Kiperwasser and Goldberg (2016), si distinguerà la versione transition-based, abbreviandola con KG16:T, da quella graph-based, abbreviandola con KG16:G.

In Tabella 3.2 sono riassunte le caratteristiche dei vari parser relativamente all'approccio al parsing e l'algoritmo usato. In Tabella 3.3 sono riassunte le caratteristiche dei parser per quanto riguarda l'architettura delle reti neurali usata, l'algoritmo di ottimizzazione e la funzione di loss.

Parser	Riferimento	Repo.	Implementazione
CM14	Chen and Manning (2014)	CoreNLP	Java
BA15	Dyer et al. (2015) Ballesteros et al. (2015)	LSTM	DyNet C++
KG16	Kiperwasser and Goldberg (2016)	BiST	DyNet Python
AN16	Andor et al. (2016)	SyntaxNet	TensorFlow Python
CH16	Cheng et al. (2016)	BiAtt-DP	C++
DM17	Dozat and Manning (2017) Dozat et al. (2017)	Parser-v1	TensorFlow Python
SH17	Shi et al. (2017b,a)	C2L2	DyNet Python
NG17	Nguyen et al. (2017)	jPTDP	DyNet Python

Tabella 3.1: Neural dependency parser usati negli esperimenti. La prima colonna definisce la sigla abbreviata con cui si potrà farà riferimento ad uno specifico parser descritto negli articoli associati. Nella colonna Repository sono disponibili i link alle implementazioni dei parser sviluppate dagli autori.

Parser	Approccio	Parsing
CM14	Transition-based: arc-standard	Greedy
BA15	Transition-based: arc-standard	Beam-search
KG16:T	Transition-based: arc-hybrid dinamico	Greedy
KG16:G	Graph-based: arc-factored	Eisner
AN16	Transition-based: arc-standard	Beam-search
CH16	Graph-based: arc-factored	Chu-Liu/Edmonds
DM17	Graph-based: arc-factored	Chu-Liu/Edmonds
SH17	Transition-based: arc-hybrid e arc-eager Graph-based: arc-factored	Greedy Eisner
NG17	Graph-based: arc-factored	Eisner

Tabella 3.2: Caratteristiche dei parser analizzati relativamente all'approccio al parsing. Nell'ultima colonna sono riportati gli algoritmi di parsing usati dal parser.

Parser	Architettura	Ottim.	Loss
CM14	MLP	AdaGrad	Cross-entropy
BA15	Stack LSTM	SGD	Neg. conditional log-likelihood
KG16	Deep BiLSTM con MLP	Adam	Hinge
AN16	MLP	Momentum	Conditional Random Field
CH16	BiGRU attention con MLP	AdaGrad	Cross-entropy
DM17	Deep Biaffine attention con MLP	Adam	Cross-entropy
SH17	Deep Biaffine attention con MLP e programmazione dinamica	Adam	Hinge
NG17	Deep BiLSTM con MLP	Adam	Cross-entropy + Hinge

Tabella 3.3: Caratteristiche dei parser analizzati relativamente al tipo di architettura adottata, algoritmo di ottimizzazione e funzione di loss.

3.2 Esperimenti e Risultati

L'obiettivo degli esperimenti consiste nel valutare i parser precedentemente descritti utilizzando treebank in lingua italiana presi dalla collezione delle Universal Dependencies.

Dato il largo uso di applicazioni basate sull'analisi del testo dei social media, uno degli aspetti fondamentali è capire come migliorare il trattamento del testo proveniente da queste nuove fonti. Per questo motivo negli esperimenti saranno utilizzati due treebank differenti, uno costituito da testi con dominio generico e l'altro da testi presi esclusivamente da dominio social media. Si vuole sperimentare se l'uso di risorse specifiche al dominio dei social media sia in grado di migliorare le prestazioni dei dependency parser nell'analisi del testo degli stessi. Gli esperimenti sono organizzati in tre diversi setup:

1. Nel primo setup (Sezione 3.2.3) si apprendono i modelli di parsing su un treebank in lingua italiana di dominio generico. Questo setup sarà utile per confrontare le prestazioni dei parser addestrati su testi generici con quelle ottenute a partire da testi provenienti dai social media;

2. Nel secondo setup (Sezione 3.2.4) si apprendono i modelli di parsing su un treebank in lingua italiana costituito unicamente da testi di dominio social media;

3. Nell'ultimo setup (Sezione 3.2.5) si apprendono i modelli di parsing sulla fusione dei corpora usati nei setup precedenti.

I risultati delle valutazioni sono ottenuti utilizzando l'*Evaluation tool* del software DependAble[1], descritto in Choi et al. (2015), basato sullo script standard di valutazione `eval.pl` del CoNLL-X Shared Task. Per ogni parser sono state valutate cinque diverse istanze, le valutazioni includono la punteggiatura e riportano media e deviazione standard dei valori ottenuti nelle istanze per ogni setup. Su tutti i modelli è stato calcolato, con lo stesso

[1]Disponibile online: `https://github.com/emorynlp/dependable`

tool, il livello di significatività statistica usando il test McNemar. Il sistema sul quale sono stati eseguiti gli esperimenti è un Intel Quad-Core i7-3770K 3.50GHz con 16GB di memoria RAM.

3.2.1 Treebank

I treebank usati negli esperimenti sono annotati seguendo il formato Universal Dependencies v2. Prima di procedere con gli esperimenti tutti i treebank sono stati convertiti dal formato CoNLL-U al formato CoNLL-X, tramite lo script di conversione `conllu_to_conllx.pl` messo a disposizione tra i tool ufficiali UD[2]. Questa conversione è stata necessaria perché alcuni parser sono stati sviluppati precedentemente all'uscita del formato CoNLL-U e accettano in input solo il formato CoNLL-X.

UD Italian 2.1

Il corpus considerato per rappresentare testi di dominio generico è **UD Italian** nella versione 2.1[3]. Il treebank UD Italian è stato ricavato convertendo il corpus ISDT (Italian Stanford Dependency Treebank), con schema Stanford Dependencies, nello schema Universal Dependencies, come descritto in Attardi et al. (2015). L'ISDT è stato rilasciato per la prima volta durante il task di dependency parsing in Evalita 2014 ed è stato ricavato dalla conversione del corpus MIDT (Merged Italian Dependency Treebank) in Bosco et al. (2013). Il MIDT è il risultato della fusione e conversione di due dependency treebank precedentemente esistenti per la lingua italiana:

- Il treebank TUT (Turin University Treebank), in Bosco et al. (2000);
- Il treebank ISST-TANL, rilasciato inizialmente come ISST-CoNLL per il CoNLL 2007 shared task e sviluppato a partire dall'ISST (Italian Syntactic-Semantic Treebank), in Montemagni et al. (2000).

[2] `https://github.com/UniversalDependencies/tools`

[3] `https://lindat.mff.cuni.cz/repository/xmlui/handle/11234/1-2515`

Nell'intero corpus sono presenti 13.884 frasi uniche, quelle già presenti nel treebank ISDT più altre nuove frasi aggiunte dopo la conversione nel formato UD. La suddivisione dell'intero corpus in train, development e test è riportata nella Tabella 3.4.

		Frasi	Parole	Token
train file:	`it-ud-train.conllu`	12.838	270.703	252.631
dev file:	`it-ud-dev.conllu`	564	11.908	11.133
test file:	`it-ud-test.conllu`	482	10.417	9.680

Tabella 3.4: Suddivisione in train, development e test set del corpus UD Italian 2.1

UD Italian PoSTWITA 2.2

Il corpus considerato per rappresentare testi di dominio social media è **UD Italian PoSTWITA** nella versione 2.2[4]. Il corpus, descritto in Sanguinetti et al. (2017), è costituito da testi presi dalla piattaforma Twitter in italiano. UD Italian PoSTWITA è stato creato a partire da un dataset usato per il part-of-speech tagging di social media in Evalita 2016. La suddivisione del corpus è riportata in Tabella 3.5.

		Frasi	Parole	Token
train file:	`it_postwita-ud-train.conllu`	5.368	99.441	95.308
dev file:	`it_postwita-ud-dev.conllu`	671	12.335	11.850
test file:	`it_postwita-ud-test.conllu`	674	12.668	12.114

Tabella 3.5: Suddivisione in train, development e test set del corpus UD Italian PoSTWITA 2.2

[4] `https://github.com/UniversalDependencies/UD_Italian-PoSTWITA`

3.2.2 Iperparametri e configurazione

Per inizializzare i valori di word embedding dei parser sono stati pre-addestrati dei vettori di dimensione $d = 100$, utilizzando l'implementazione di WORD2VEC presente nel pacchetto Python Gensim[5] a partire dal corpus itWaC, descritto in Baroni et al. (2009), che comprende circa 2 miliardi di token in italiano. Riportiamo, per completezza, gli iperparametri di tutte le configurazioni, utilizzando quelli suggeriti dagli autori dei parser nei rispettivi articoli.

Per Chen and Manning (2014) i parametri delle rete sono inizializzati casualmente in $[-0.01, 0.01]$. L'algoritmo di ottimizzazione AdaGrad ha learning rate $\eta = 0.01$ con $\epsilon = 10^{-6}$, dimensione di mini-batch $b = 10.000$ e termina dopo aver raggiunto 20.000 iterazioni. Il peso di regolarizzazione l_2 è impostato a $\lambda = 10^{-8}$ e la probabilità di dropout è $p = 0.5$. L'ampiezza del livello nascosto della MLP è $h = 200$.

Per Dyer et al. (2015) si è utilizzata l'implementazione proposta in Ballesteros et al. (2015). I parametri sono inizializzati seguendo lo schema Xavier, il peso di regolarizzazione l_2 è $\lambda = 10^{-6}$ e, ad ogni iterazione con probabilità $p = 0.2$, ogni parola singoletto nel training set è sostituita con un token UNK per evitare overfitting. L'algoritmo di ottimizzazione usato è SGD senza mini-batch con learning rate iniziale fissato a $\eta_0 = 0.1$ che si aggiorna progressivamente durante l'apprendimento come $\eta_t = \eta_0/(1 + \rho t)$, con $\rho = 0.1$ e t numero di epoch già complete, cioè come un momentum senza fattore γ. L'algoritmo termina dopo aver raggiunto le 5.500 iterazioni. L'architettura stack LSTM ha $l = 2$ livelli e ampiezza del livello nascosto fissata a $l = 100$. La beam width è impostata a 1, dunque il parsing è di tipo greedy deterministico.

Per Kiperwasser and Goldberg (2016) durante l'apprendimento si usa una variante della tecnica *word dropout* in cui una parola w, che ha frequenza $\#(w)$, può essere sostituita da un simbolo UNK con una probabilità $p = \alpha/(\#(w) + \alpha)$ con $\alpha = 0.25$. Se viene effettuata la sostituzione il vettore di

[5] `https://radimrehurek.com/gensim/`

embedding associato alla parola è soggetto a dropout con probabilità $p = 0.5$. Tutti i parametri sono inizializzati tramite lo schema Xavier. L'algoritmo di ottimizzazione Adam è inizializzato con learning rate $\eta = 0.1$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ e $\beta_2 = 0.999$ e termina dopo 30 epochs. La rete LSTM è composta da $k = 2$ livelli con una dimensione di $l = 125$ e ampiezza del livello nascosto della rete MLP $h = 100$.

Per Andor et al. (2016) la rete feed-forward è composta da due livelli nascosti h_1 e h_2 entrambi di dimensione 512. L'algoritmo di ottimizzazione ha un fattore di momentum $\gamma = 0.9$, learning rate iniziale $\eta = 0.02$, dimensione dei mini-batch $b = 8$ e termina dopo 10 epochs. L'ampiezza della beam search è impostata a 16.

Per Cheng et al. (2016) i parametri delle rete sono inizializzati seguendo una distribuzione gaussiana $\mathcal{N}(0, 0.1)$ e la dimensione del livello nascosto della rete è $h = 368$. L'algoritmo di discesa del gradiente è AdaGrad con dimensione dei mini-batch $b = 1$ e learning rate $\eta = 0.0004$, che dimezza ad ogni iterazione solo se la cross-entropy loss sul development set aumenta. L'algoritmo termina quando il valore di loss aumenta per due volte di fila, quindi non è fissato a priori un limite massimo di iterazioni. Per il training sono state usate le codifiche one-hot degli attributi FORM, LEMMA, CPOSTAG, POSTAG, FEATS e DEPREL.

Per Dozat and Manning (2017) la cella RNN usata è di tipo LSTM. L'algoritmo di ottimizzazione Adam ha valore iniziale di learning rate $\eta = 0.002$ con $\beta_1 = \beta_2 = 0.9$, $\epsilon = 10^{-12}$, dimensione del mini-batch $b = 5.000$ e termina dopo 25.000 iterazioni. Per regolarizzare l'apprendimento si usa la tecnica dropout con probabilità $p = 0.33$. Ogni rete LSTM ha $k = 3$ livelli di dimensione $l = 300$, mentre le MLP hanno un solo livello nascosto di ampiezza $h = 100$.

Per Shi et al. (2017b,a) i parametri sono inizializzati usando lo schema Xavier. L'algoritmo di ottimizzazione Adam ha learning rate iniziale $\eta = 0.001$ con $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, mini-batch di dimensione $b = 50$ e termina dopo 20 epochs. La rete LSTM è composta da $k = 2$ livelli di

dimensione $l = 256$ e la MLP ha ampiezza $h = 128$.

Per Nguyen et al. (2017) i parametri sono inizializzati seguendo lo schema Xavier. L'algoritmo di ottimizzazione è Adam senza mini-batch con learning rate iniziale $\eta = 0.1$, $\epsilon = 10^{-8}$, $\beta_1 = 0.9$ e $\beta_2 = 0.999$ e termina dopo 30 epochs. Si usa come tecnica di regolarizzazione la variante di word dropout di Kiperwasser and Goldberg (2016) con $\alpha = 0.25$. La LSTM bidirezionale è composta da $k = 2$ livelli di dimensione $l = 128$, mentre la rete MLP ha un livello nascosto di ampiezza $h = 100$.

In Tabella 3.6 sono riassunti i valori degli iperparametri dei parser.

Parser	Iperparametri
CM14	$\eta = 0.01,\ \epsilon = 10^{-6},\ b = 10.000,\ p = 0.5,\ h = 200,$ iter $= 20.000,\ \lambda = 10^{-8}$
BA15	$\eta = 0.1,\ k = 2,\ l = 100,$ iter $= 5.500,\ \lambda = 10^{-6}$
KG16	$\eta = 0.1,\ \epsilon = 10^{-8},\ \beta_1 = 0.9,\ \beta_2 = 0.999,\ k = 2,\ l = 125,$ $h = 100,$ epochs $= 30,\ p = 0.5$
AN16	$\eta = 0.02,\ \gamma = 0.9,\ h_1 = h_2 = 512,\ b = 8,$ beam $= 16,$ epochs $= 10$
CH16	$\eta = 0.0004,\ h = 368,\ b = 1$
DM17	$\eta = 0.002,\ \beta_1 = \beta_2 = 0.9,\ \epsilon = 10^{-12},\ k = 3,\ l = 300,\ h = 100,$ $b = 5.000,$ iter $= 25.000,\ p = 0.33$
SH17	$\eta = 0.001,\ \beta_1 = 0.9,\ \beta_2 = 0.9999,\ \epsilon = 10^{-8},\ k = 2,\ l = 256,$ $h = 128,\ b = 50,$ epochs $= 20$
NG17	$\eta = 0.1,\ \epsilon = 10^{-8},\ \beta_1 = 0.9,\ \beta_2 = 0.999,\ k = 2,\ l = 128,$ $h = 100,$ epochs $= 30$

Tabella 3.6: Valori degli iperparametri dei parser: η è il learning rate; γ è il fattore di momentum; $\epsilon, \beta_1, \beta_2$ sono i parametri per Adam; b è la dimensione del mini-batch; p è la probabilità di dropout; h è l'ampiezza dei livelli nascosti MLP; k è il numero di livelli delle RNN; l la dimensione dello stato delle RNN; λ è il peso del termine di regolarizzazione.

3.2.3 Setup 0

In questo esperimento sono stati utilizzati il train e il dev file del corpus UD Italian 2.1 per apprendere i modelli di parsing. I modelli appresi sono stati in seguito valutati su due diversi test set: il primo è il test file del corpus UD Italian 2.1 (Tabella 3.4), il secondo è il test file del corpus UD Italian PoSTWITA 2.2 (Tabella 3.5). I risultati delle valutazioni sono riportati rispettivamente in Tabella 3.7 e Tabella 3.8.

	Dev Ita				**Test Ita**			
	UAS		LAS		UAS		LAS	
	μ	σ	μ	σ	μ	σ	μ	σ
CM14	88.20%	0.18	85.46%	0.14	89.33%	0.17	86.85%	0.22
BA15	91.15%	0.11	88.55%	0.23	91.57%	0.38	89.15%	0.33
KG16:T	91.17%	0.29	88.42%	0.24	91.21%	0.33	88.72%	0.24
KG16:G	91.85%	0.27	89.23%	0.31	92.04%	0.18	89.65%	0.10
AN16	85.52%	0.34	77.67%	0.30	87.70%	0.31	79.48%	0.24
CH16	92.42%	0	89.60%	0	92.82%	0	90.26%	0
DM17	**93.37**%	0.27	**91.37**%	0.24	**93.72**%	0.14	**91.84**%	0.18
SH17	89.67%	0.24	85.05%	0.24	89.89%	0.29	84.55%	0.30
NG17	90.37%	0.12	87.19%	0.21	90.67%	0.15	87.58%	0.11

Tabella 3.7: Media e deviazione standard delle valutazioni su cinque istanze di training dei modelli appresi sul train e dev file del corpus UD Italian 2.1. Tutti i risultati sono statisticamente significativi ($p < 0.05$) ed i migliori valori sono evidenziati in grassetto.

	Test PoSTW			
	UAS		LAS	
	μ	σ	μ	σ
Chen and Manning (2014)	72.13%	0.31	63.96%	0.25
Ballesteros et al. (2015)	73.86%	0.30	65.92%	0.19
Kiperwasser and Goldberg (2016):T	77.04%	0.24	67.99%	0.23
Kiperwasser and Goldberg (2016):G	75.86%	0.26	67.81%	0.24
Andor et al. (2016)	65.56%	0.15	52.53%	0.16
Cheng et al. (2016)	76.94%	0	67.54%	0
Dozat and Manning (2017)	**77.80**%	0.24	**68.54**%	0.26
Shi et al. (2017b,a)	71.67%	0.23	67.07%	0.19
Nguyen et al. (2017)	70.23%	0.93	61.11%	0.50

Tabella 3.8: Media e deviazione standard delle valutazioni su cinque istanze di training dei modelli appresi sul train e dev file del corpus UD Italian 2.1. Il test file utilizzato è quello del corpus UD Italian PoSTWITA 2.2. Tutti i risultati sono statisticamente significativi ($p < 0.05$) ed i migliori valori sono evidenziati in grassetto.

3.2.4 Setup 1

In questo esperimento sono stati utilizzati il train e il dev file del corpus UD Italian PoSTWITA 2.2 (Tabella 3.5) per apprendere i modelli di parsing. I modelli appresi sono stati in seguito valutati sul dev e test file dello stesso corpus. I risultati della valutazione sono riportati in Tabella 3.9.

	Dev PoSTW				**Test PoSTW**			
	UAS		LAS		UAS		LAS	
	μ	σ	μ	σ	μ	σ	μ	σ
CM14	81.03%	0.17	75.24%	0.30	81.50%	0.28	76.07%	0.17
BA15	83.44%	0.20	77.70%	0.25	84.06%	0.38	78.64%	0.44
KG16:T	77.38%	0.14	68.81%	0.25	77.41%	0.43	69.13%	0.43
KG16:G	78.81%	0.23	70.14%	0.33	78.78%	0.44	70.52%	0.51
AN16	77.74%	0.25	66.63%	0.16	77.78%	0.33	67.21%	0.30
CH16	84.78%	0	78.51%	0	86.12%	0	79.89%	0
DM17	**85.01**%	0.16	**78.80**%	0.09	**86.26**%	0.16	**80.40**%	0.19
SH17	80.52%	0.18	73.71%	0.14	81.11%	0.29	74.53%	0.26
NG17	82.02%	0.11	75.20%	0.24	82.74%	0.39	76.22%	0.41

Tabella 3.9: Media e deviazione standard delle valutazioni su cinque istanze di training dei modelli appresi sul train e dev file del corpus UD Italian PoSTWITA 2.2. Tutti i risultati sono statisticamente significativi ($p < 0.05$) ed i migliori valori sono evidenziati in grassetto.

3.2.5 Setup 2

Nell'ultimo esperimento i file di train e dev del corpus UD Italian 2.1 sono stati concatenati ai file di train e dev del corpus UD Italian PoSTWITA 2.2 in modo da ottenere i file `it+postwita-ud-train.conllu` con 18.206 frasi e 347.939 token e `it+postwita-ud-dev.conllu` con 1.235 frasi e 22.983 token. I modelli sono stati appresi a partire da questi due nuovi file e valutati sul file di test del corpus UD Italian PoSTWITA 2.2. I risultati della valutazione sono riportati in Tabella 3.10.

	Dev Ita+PoSTW				**Test PoSTW**			
	UAS		LAS		UAS		LAS	
	μ	σ	μ	σ	μ	σ	μ	σ
CM14	85.52%	0.13	81.51%	0.05	82.62%	0.24	77.45%	0.23
BA15	87.85%	0.13	83.80%	0.12	85.15%	0.29	80.12%	0.27
KG16:T	83.89%	0.23	77.77%	0.26	80.47%	0.36	72.92%	0.46
KG16:G	84.70%	0.14	78.41%	0.14	81.41%	0.37	73.49%	0.19
AN16	82.95%	0.33	73.46%	0.37	79.81%	0.27	69.19%	0.19
CH16	89.16%	0	84.56%	0	86.85%	0	80.93%	0
DM17	**89.72**%	0.10	**85.85**%	0.13	**87.22**%	0.24	**81.65**%	0.21
SH17	85.85%	0.36	80.00%	0.39	83.12%	0.50	76.38%	0.38
NG17	86.81%	0.04	82.13%	0.09	84.09%	0.07	78.02%	0.11

Tabella 3.10: Media e deviazione standard delle valutazioni su cinque istanze di training dei modelli appresi sul train e dev file ottenuti dalla concatenazione del corpus UD Italian 2.1 e UD Italian PoSTWITA 2.2. Il test file usato è quello del corpus UD Italian PoSTWITA 2.2. Tutti i risultati sono statisticamente significativi ($p < 0.05$) ed i migliori valori sono evidenziati in grassetto.

3.2.6 Confronto

Nella Tabella 3.11 sono riportate le medie delle valutazioni sul test file del corpus UD Italian PoSTWITA 2.2 ottenute nei precedenti setup a partire dai diversi corpora di apprendimento: usando solo il train e dev file del corpus UD Italian 2.1 (setup 0), usando solo il train e dev file del corpus UD Italian PoSTWITA 2.2 (setup 1), usando i train e dev file ottenuti dalla fusione dei due corpora precedenti (setup 2). Gli stessi risultati sono riprodotti graficamente in Figura 3.1.

	Ita		**PoSTW**		**Ita+PoSTW**	
	UAS	LAS	UAS	LAS	UAS	LAS
CM14	72.13%	63.96%	81.50%	76.07%	82.62%	77.45%
BA15	73.86%	65.92%	84.06%	78.64%	85.15%	80.12%
KG16:T	77.04%	67.99%	77.41%	69.13%	80.47%	72.92%
KG16:G	75.86%	67.81%	78.78%	70.52%	81.41%	73.49%
AN16	65.56%	52.53%	77.78%	67.21%	79.81%	69.19%
CH16	76.94%	67.54%	86.12%	79.89%	86.85%	80.93%
DM17	77.80%	68.54%	86.26%	80.40%	87.22%	81.65%
SH17	71.67%	67.07%	81.11%	74.53%	83.12%	76.38%
NG17	70.23%	61.11%	82.74%	76.22%	84.09%	78.02%
Media	**73.46%**	**64.72%**	**81.75%**	**74.73%**	**83.42%**	**76.68%**

Tabella 3.11: Risultati delle valutazioni sul test file del corpus UD Italian PoSTWITA 2.2 ottenuti nei tre setup dai modelli appresi sui treebank UD Italian 2.1 (Ita), UD Italian PoSTWITA 2.2 (PoSTW) e la loro fusione (Ita+PoSTW).

Dai risultati emerge che i modelli appresi dal corpus UD Italian PoSTWITA 2.2 sono migliori, in media, di ∼8.3% in UAS e ∼10% in LAS rispetto ai modelli appresi dal solo corpus UD Italian 2.1. Se si considerano, invece,

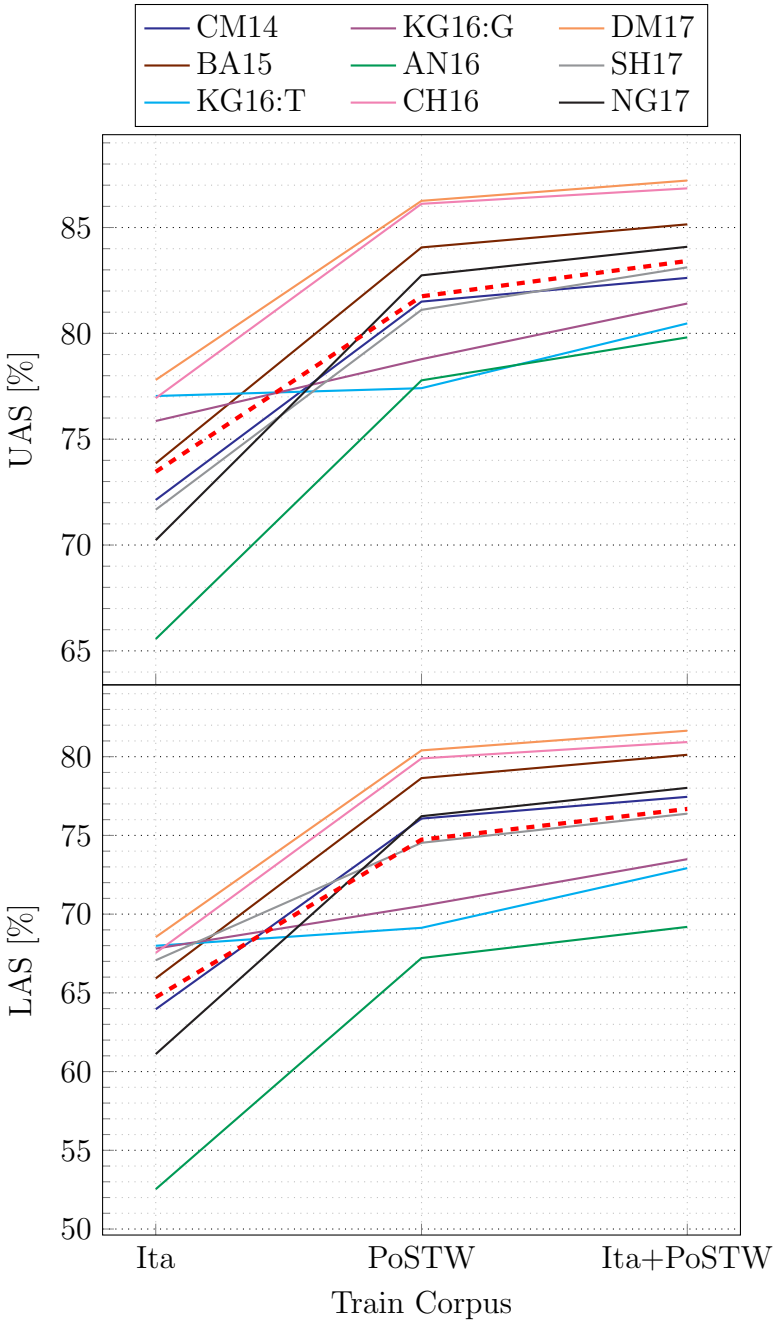


Figura 3.1: Risultati della Tabella 3.11 organizzati per corpus di apprendimento sulle ascisse e per UAS e LAS sulle ordinate. La linea rossa tratteggiata rappresenta la media dei risultati delle valutazioni dei parser.

i modelli appresi dall'unione dei due treebank si può notare che c'è ancora un margine di miglioramento con un incremento di ∼1.7% in UAS e ∼2% in LAS, rispetto ai modelli appresi utilizzando solo il corpus UD Italian PoSTWITA 2.2. Il miglior modello in tutti gli esperimenti risulta essere quello di Dozat and Manning (2017).

Da ciò possiamo dedurre che per avere buone prestazioni all'interno di un dominio, nello specifico quello dei social media, è fondamentale utilizzare più dati possibili, includendo dati di dominio, per apprendere i modelli di parsing. A maggior supporto di questa ipotesi si può notare come la fusione dei due treebank, UD Italian 2.1 e UD Italian PoSTWITA 2.2, possa generare un buon margine di miglioramento.

In Evalita 2014 è riportata la migliore valutazione che stabilisce lo stato dell'arte per il dependency parsing in italiano: LAS 88.76% e UAS 93.55%[6]. Il corpus usato per l'apprendimento nel task è proprio l'ISDT (Italian Stanford Dependency Treebank), lo stesso corpus a partire dal quale è stato creato UD Italian. Il valore UAS per il miglior parser degli esperimenti su UD Italian 2.1 (Tabella 3.7) supera di poco lo stato dell'arte stabilito in Evalita 2014, mentre il LAS è 91.84%, superiore a quello di Evalita 2014 del 3.08%. Nel Capitolo 4 si riescono a migliorare ulteriormente le prestazioni portando l'UAS alla soglia del 94%. Tuttavia, va precisato che pur essendo costruito sullo stesso corpus, l'UD Italian non coincide esattamente con l'ISDT, dunque il confronto non è completo dato che i due treebank hanno delle differenze nella loro costruzione come schema di annotazione e frasi aggiunte e corrette nell'UD Italian che non compaiono in ISDT.

Infatti in Attardi et al. (2015) viene mostrato come il cambio da ISDT a UD Italian 1.2 generi migliori prestazioni su esperimenti condotti con parser di tipo statistico. I risultati in quest'ultimo articolo sono le uniche valutazioni pubblicate in letteratura sul treebank UD Italian, nella versione 1.2.

Tra gli esperimenti condotti con i parser di tipo statistico il miglior risultato su UD Italian 1.2 è del parser Mate con valutazione UAS 92.47% e

[6] `http://www.evalita.it/2014/tasks/dep_par4IE`

LAS 90.22%. Anche in questo caso non possiamo effettuare un confronto completo con i risultati ottenuti usando UD Italian 2.1, pur essendo molto più vicini ad avere due treebank simili[7]. Fermo restando che UD Italian 1.2 e UD Italian 2.1 sono simili ma non coincidono, le valutazioni ottenute del parser Dozat and Manning (2017) su UD Italian 2.1 sono superiori a quelle di Mate su UD Italian 1.2 sia in UAS che LAS.

[7]Per le modifiche tra le versioni è possibile vedere il Changelog del corpus UD Italian: `https://github.com/UniversalDependencies/UD_Italian-ISDT`

Capitolo 4

Ensemble Neural Dependency Parser

Le tecniche utilizzabili e studiate per creare un ensemble nell'ambito del dependency parsing sono distinguibili in due tipologie: **voting** e **stacking**. Nel primo caso la costruzione dell'ensemble avviene a parsing-time, invece nel secondo caso a learning-time. Come riportato in Surdeanu and Manning (2010) la tecnica di voting è da preferire poiché produce risultati simili allo stacking ma è più semplice da realizzare dato che si riescono ad evitare, in tal modo, i lunghi tempi di apprendimento. Considerando l'approccio del voting, un ensemble di m parser per una frase $S = w_0 w_1 \dots w_n$ di lunghezza n consiste nel generare un dependency tree a partire dagli m alberi predetti dai singoli parser usati nell'ensemble, detti *votanti*. Gli esperimenti riportati di seguito sono realizzati prendendo in considerazione solo i migliori modelli sul development set, appresi nel setup 0 con test set relativo al corpus UD Italian 2.1, e quelli nel setup 2, con test set relativo al corpus UD Italian PoSTWITA 2.2.

Prima di procedere alla sperimentazione delle tecniche di voting cerchiamo di capire se, almeno in via teorica, esiste la possibilità di migliorare le prestazioni dei parser di base tramite lo sviluppo di un ensemble. Immaginiamo di avere due oracoli:

- Macro, che a partire dagli m parser coinvolti nell'ensemble è in grado di identificare e selezionare correttamente quale tra gli m alberi è quello valido per la frase S;

- Micro, che per ogni arco degli alberi predetti dagli m parser è in grado di identificare e selezionare il migliore.

Questi due oracoli possono essere ottenuti semplicemente a partire dal gold treebank, che ci indica quale albero o arco è quello corretto rispetto ad uno predetto. Tramite il software DependAble descritto in Choi et al. (2015) sono state valutate le prestazioni di Micro e Macro in riferimento ai gold treebank dei relativi corpora e, considerando tutti i parser, i risultati sono riportati in Tabella 4.1. Questi ultimi forniscono un limite superiore teorico per la costruzione di un ensemble ideale e, da quanto emerge, sembra esserci un buon margine di miglioramento.

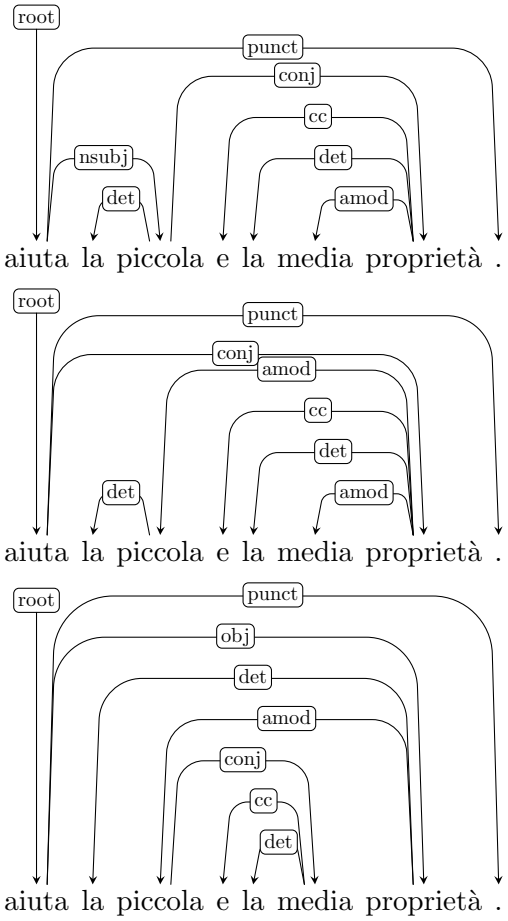
	Development			**Test**		
	UAS	LAS	LS	UAS	LAS	LS
	UD Italian					
Micro	98.30%	97.82%	99.92%	98.08%	97.72%	99.93%
Macro	96.62%	95.10%	98.83%	96.31%	94.82%	98.74%
	UD Italian+PoSTWITA					
Micro	97.08%	96.02%	99.66%	96.32%	94.73%	99.46%
Macro	94.62%	91.29%	97.37%	93.27%	88.50%	96.07%

Tabella 4.1: Risultati della valutazione utilizzando gli oracoli Micro e Macro a partire dai gold treebank del setup 0 (UD Italian) e setup 2 (UD Italian+PoSTWITA), utilizzando tutti i parser.

4.1 Voting

La tecnica di voting è stata proposta per la prima volta in Zeman and Žabokrtský (2005). Data la frase $S = w_1 w_2 \dots w_n$, ognuno degli m parser di base considerato nell'ensemble contribuisce assegnando uno score ad ogni relazione di dipendenza candidata (w_i, r, w_j), con $0 \leq i, j \leq n$ e $i \neq j$, dove w_i è la dipendente, w_j è la testa e r il tipo della relazione di dipendenza. Le relazioni di dipendenza candidate sono prese tra tutte quelle disponibili a partire dagli m alberi predetti dai parser. Nell'esempio di Figura 4.1 consideriamo tre dependency tree (mostrati in alto a sinistra) ottenuti da tre diversi parser. A partire da questi, ogni singolo arco distinto diventa un arco candidato inserito nella lista degli archi candidati. Tutti gli archi candidati, dunque, compaiono in almeno uno dei dependency tree di partenza. Lo score $\mathbf{s_c}$ è il numero dei voti di ogni singolo arco candidato, calcolato come il numero di volte che questo compare nei singoli dependency tree ottenuti dai parser di base. Dopo aver ottenuto la lista dei candidati con i relativi score, per ogni parola contenuta nella frase si sceglie l'arco che ha score massimo e, in caso di pareggio, si sceglie il primo arco che figura nella lista per quella parola. Nell'esempio di Figura 4.1 si ha un pareggio per la parola *proprietà* i cui tre archi candidati hanno tutti $\mathbf{s_c} = 1$. Poiché la costruzione della lista dei candidati avviene in ordine di comparsa degli archi nei dependency tree a partire dal primo, in caso di parità l'arco che sarà scelto è quello relativo al dependency tree del primo parser. La strategia di voting che costruisce l'albero finale componendo gli archi con score massimo, o nel caso di parità scegliendo il primo arco, è nota col nome di *majority*.

Poiché nella composizione dell'albero non si pone alcun vincolo sulla struttura dello stesso, l'albero generato tramite strategia majority non ha garanzia di essere ben formato. Come è possibile vedere in Figura 4.1, a partire da tre dependency tree ben formati è stato composto un dependency tree che non è ben formato. Per evitare questo problema si può utilizzare la strategia di *switching* che consiste nel controllare se l'albero composto tramite strategia majority sia o meno ben formato e, nel caso in cui non fosse ben formato,



candidati	s_c
aiuta $\xleftarrow{\text{root}}$ ROOT	**3**
la $\xleftarrow{\text{det}}$ piccola	**2**
la $\xleftarrow{\text{det}}$ proprietà	1
piccola $\xleftarrow{\text{nsubj}}$ aiuta	1
piccola $\xleftarrow{\text{amod}}$ proprietà	**2**
e $\xleftarrow{\text{cc}}$ proprietà	**2**
e $\xleftarrow{\text{cc}}$ media	1
la $\xleftarrow{\text{det}}$ proprietà	**2**
la $\xleftarrow{\text{det}}$ media	1
media $\xleftarrow{\text{amod}}$ proprietà	**2**
media $\xleftarrow{\text{conj}}$ piccola	1
proprietà $\xleftarrow{\text{conj}}$ piccola	**1**
proprietà $\xleftarrow{\text{conj}}$ aiuta	1
proprietà $\xleftarrow{\text{obj}}$ aiuta	1
. $\xleftarrow{\text{punct}}$ aiuta	**3**

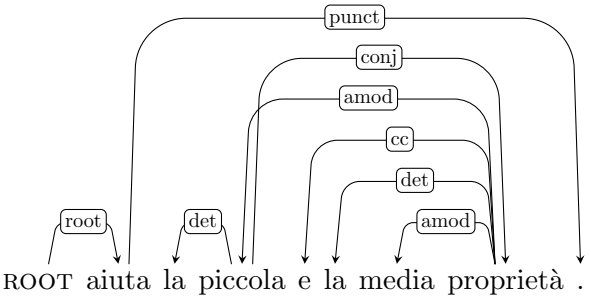


Figura 4.1: Dai dependency tree ottenuti da tre parser di base (sinistra) si stabiliscono gli archi candidati (destra) dai quali si può generare il dependency finale (in basso). L'albero non è ben formato poiché non è connesso e presenta il ciclo piccola $\xrightarrow{\text{conj}}$ proprietà $\xrightarrow{\text{amod}}$ piccola.

sostituirlo interamente con il dependency tree predetto dal primo parser. Segue intuitivamente che un ensemble per avere senso deve considerare almeno tre diversi dependency tree.

Prima di cominciare a sperimentare queste due strategie è opportuno fare qualche considerazione sull'ensemble di classificatori in generale. Immaginiamo di avere tre classificatori binari C_1, C_2, C_3 che predicono se un oggetto appartiene o meno ad una classe, restituendo rispettivamente un 1 o uno 0. Volendo classificare una sequenza di dieci oggetti, la cui corretta classificazione sappiamo essere 1111111111, supponiamo di ottenere le seguenti predizioni, dove tra parentesi è indicata l'accuratezza:

$$\begin{aligned} C_1 &= 1111111100 \quad (80\%) \\ C_2 &= 1111111100 \quad (80\%) \\ C_3 &= 1011111100 \quad (70\%) \end{aligned} \tag{4.1}$$

Le sequenze predette commettono tutte lo stesso errore nel classificare gli ultimi due oggetti come classe 0. Un ensemble di questi tre classificatori, tramite la strategia di majority, costruisce la sequenza 1111111100, che ha un'accuratezza dell'80%. In altre parole, l'ensemble non è in grado di migliorare l'accuratezza della predizione. Consideriamo altri tre classificatori che generano le seguenti predizioni con le rispettive accuratezze:

$$\begin{aligned} C'_1 &= 1111111100 \quad (80\%) \\ C'_2 &= 0111011101 \quad (70\%) \\ C'_3 &= 1000101111 \quad (60\%) \end{aligned} \tag{4.2}$$

Mediamente l'accuratezza è inferiore a quella dell'esempio (4.1) ma, se provassimo a costruire un ensemble a partire da questi ultimi tre classificatori, otterremmo la sequenza 1111111101 che ha un'accuratezza del 90%, superiore a quella del miglior classificatore. Questo avviene perché nel caso (4.2), gli errori sono distribuiti differentemente nei classificatori rispetto al caso (4.1), in cui commettevano lo stesso tipo di errore (sbagliando la classificazione degli ultimi due oggetti). Questa diversificazione dell'errore, permette

ai modelli utilizzati in un ensemble di compensare gli uni gli errori degli altri, così da migliorare l'accuratezza complessiva.

Come analisi preliminare cerchiamo di catturare e misurare la diversità, o equivalentemente la similarità, dei parser a nostra disposizione. L'idea alla base è quella di capire quante volte i parser sono d'accordo sul predire una certa relazione, questa misura è l'agreement.

Consideriamo due dependency tree $T = (V, A)$ e $\tilde{T} = (V, \tilde{A})$ per una stessa frase $S = w_0 w_1 w_2 \dots w_n$, dove l'insieme dei nodi V (ovvero le parole della frase) sono comuni ad entrambi gli alberi e l'insieme di archi A e $\tilde{A}$ possono essere diversi. Definiamo l'**agreement** tra due dependency tree come:

$$\text{agreement}(T, \tilde{T}) = \frac{1}{|A|} \sum_{\substack{(w,r,h)\in A \\ (w,\tilde{r},\tilde{h})\in \tilde{A}}} \begin{cases} 1 & \text{se } h = \tilde{h} \text{ e } r = \tilde{r} \\ 0 & \text{altrimenti} \end{cases} \tag{4.3}$$

dove (w, r, h) e $(w, \tilde{r}, \tilde{h})$ sono le relazioni di dipendenza rispettivamente nell'albero T e $\tilde{T}$ per ogni parola $w \in \{w_1, \dots w_n\}$. Dalla definizione segue che l'agreement è una misura simmetrica, per cui si ha che $\text{agreement}(T, \tilde{T}) = \text{agreement}(\tilde{T}, T)$. In altre parole l'agreement calcola la percentuale delle relazioni di dipendenza tra due alberi che sono state etichettate nello stesso modo. Estendendo la definizione di agreement su un insieme di alberi come la media dell'agreement sull'intero treebank, possiamo calcolarci l'agreement tra i parser usando le predizioni. In Tabella 4.2 è riportato l'agreement per ciascuna coppia di parser utilizzando i modelli appresi nel setup 0, mentre in Tabella 4.3 quelli nel setup 2.

L'agreement dipende in buona parte anche dalle prestazioni di un parser poiché è legato alla metrica del LAS, che è definibile come $\text{agreement}(T, G)$, dove G è il dependency tree gold standard.

Anche se la strategia majority potrebbe generare parser malformati, sarà considerata per fornire un confronto con la strategia switching. In alcuni casi anche avere un albero malformato, ma con una buona accuratezza, potrebbe tornare utile in alcuni processi dove è prevista una correzione manuale dei dependency tree, come avviene ad esempio nell'annotazione automatica di un

agreement	BA15	KG16:T	KG16:G	AN16	CH16	DM17	SH17	NG17
CM14	87.59%	87.88%	87.81%	81.19%	88.11%	88.30%	80.67%	85.74%
BA15		88.97%	90.26%	82.11%	89.85%	90.59%	82.85%	87.64%
KG16:T			91.25%	82.27%	90.25%	91.02%	82.61%	88.01%
KG16:G				82.85%	90.89%	91.92%	83.30%	88.81%
AN16					82.48%	83.56%	78.83%	84.32%
CH16						92.38%	83.98%	88.66%
DM17							85.38%	90.27%
SH17								83.70%

Tabella 4.2: Agreement calcolato sul development set a partire dalle predizioni dei modelli appresi nel setup 0 (UD Italian 2.1).

agreement	BA15	KG16:T	KG16:G	AN16	CH16	DM17	SH17	NG17
CM14	83.79%	77.13%	76.95%	77.74%	83.73%	83.60%	74.33%	80.56%
BA15		78.34%	78.10%	78.65%	84.93%	85.58%	76.18%	82.98%
KG16:T			80.15%	77.46%	79.02%	79.85%	72.95%	80.22%
KG16:G				77.54%	79.02%	80.68%	72.90%	80.79%
AN16					78.76%	79.50%	72.34%	80.16%
CH16						87.19%	76.70%	83.66%
DM17							77.45%	84.60%
SH17								76.31%

Tabella 4.3: Agreement calcolato sul development set a partire dalle predizioni dei modelli appresi nel setup 2 (UD Italian 2.1+PoSTWITA 2.2).

treebank. Per questo motivo procediamo allo sviluppo di alcuni esperimenti considerando sia la strategia majority che quella switching.

Per gli esperimenti sono state analizzate le seguenti configurazioni di votanti:

- (dm+ch+kgg), considera i tre migliori parser, che nell'ordine sono Dozat and Manning (2017), Cheng et al. (2016) e Ballesteros et al. (2015);
- (an+cm+shi), considera i peggiori tre parser, che nell'ordine sono An-

dor et al. (2016), Chen and Manning (2014) e Shi et al. (2017b);

- (dm+cm+shi), considera il miglior parser, Dozat and Manning (2017), combinandolo con quelli che hanno agreement minore, ovvero Chen and Manning (2014) e Shi et al. (2017b);
- (an+tutti), considera tutti i parser con primo il peggiore, cioè Andor et al. (2016);
- (dm+tutti), considera tutti i parser con primo il migliore, cioè Dozat and Manning (2017).

I risultati degli esperimenti per il setup 0 sono riportati in Figura 4.4.

		Development			**Test**		
votanti	**strategia**	UAS	LAS	LS	UAS	LAS	LS
dm+ch+ba	*majority*	94.20%	92.27%	96.13%	93.77%	92.13%	96.32%
dm+ch+ba	*switching*	94.11%	92.16%	96.06%	93.79%	92.14%	96.32%
an+cm+shi	*majority*	90.43%	87.96%	93.89%	91.03%	88.47%	93.69%
an+cm+shi	*switching*	89.44%	86.77%	93.08%	90.17%	87.43%	93.17%
dm+cm+shi	*majority*	93.84%	92.03%	96.36%	93.82%	92.27%	**96.52**%
dm+cm+shi	*switching*	93.76%	91.94%	96.31%	93.82%	92.25%	96.50%
an+tutti	*majority*	94.37%	92.65%	96.52%	93.83%	92.27%	96.34%
an+tutti	*switching*	93.99%	92.15%	96.17%	93.43%	91.73%	95.93%
dm+tutti	*majority*	**94.42**%	**92.67**%	**96.53**%	**93.94**%	**92.41**%	96.43%
dm+tutti	*switching*	94.38%	92.60%	96.46%	93.91%	92.37%	96.40%
DM17 (baseline)		93.74%	91.66%	95.78%	93.75%	92.03%	96.19%

Tabella 4.4: Ensemble ottenuti applicando le strategie di switching e majority sul development e test set del corpus UD Italian 2.1, a partire dai modelli appresi nel setup 0. La baseline è data dal miglior parser di base, ovvero Dozat and Manning (2017).

Gli esperimenti mostrano che tra la strategia di majority e quella di switching non c'è un grande differenza in termini di prestazioni, fatta eccezione

per la combinazione dei peggiori tre parser (an+cm+shi) in cui la strategia majority permette di ottenere risultati migliori rispetto a quella switching. Escludendo questo caso, in generale sarebbe preferibile usare la strategia switching, utilizzando come primo parser quello con prestazioni migliori, poiché fornisce alberi ben formati. Il miglior risultato è ottenuto a partire dalla combinazione (dm+tutti), che supera di poco la combinazione dei soli tre migliori parser (dm+ch+ba). Gli stessi risultati, grossomodo, si possono ottenere usando la combinazione (an+tutti), dove sono considerati tutti i parser e come primo è scelto il peggiore. Nei risultati relativi al test set i valori di UAS hanno un incremento molto più basso rispetto a quello di LAS e LS, questo potrebbe indicare che l'ensemble riesce a catturare con maggiore precisione le etichette delle relazioni di dipendenza piuttosto che la relazione testa-dipendente. Ciò potrebbe essere legato al fatto che i candidati sono scelti all'interno di alberi già predetti e non generati ex-novo. In linea di massima, la migliore combinazione ci permette di avere un incremento per l'UAS di $\sim 0.2\%$ e il LAS e LS di $\sim 0.4\%$

I risultati degli esperimenti per il setup 2 sono riportati in Tabella 4.5. Le stesse considerazioni fatte per i risultati ottenuti per il setup 0 valgono anche per il setup 2. L'unica differenza sostanziale è che, nel caso del setup 0, l'incremento delle prestazioni rispetto alla baseline non supera la soglia dell'1%, mentre nel caso del setup 2 si supera la soglia del 2% in LAS. In particolare, nella configurazione dm+tutti si riesce ad avere un miglioramento di $\sim 0.9\%$ in UAS, ~ 2.4 in LAS e LS.

Un'ulteriore osservazione da fare circa i risultati su entrambi i setup è che la configurazione di tutti i parser (dm+tutti) permette un piccolo incremento di alcuni decimi rispetto alla configurazione che usa solo i migliori tre (dm+ch+bm). Questo può voler indicare che il numero di parser usato nell'ensemble è importante ai fini del miglioramento delle prestazioni. Altrettanto importante è la qualità del primo parser utilizzato nella combinazione di tutti i parser. Infatti, la configurazione (an+tutti) che usa come primo parser il peggiore ha una valutazione inferiore rispetto alla configurazione

		Development			Test		
votanti	**strategia**	UAS	LAS	LS	UAS	LAS	LS
dm+ch+ba	*majority*	90.57%	87.16%	92.72%	88.21%	83.64%	90.58%
dm+ch+ba	*switching*	90.51%	87.10%	92.72%	88.13%	83.51%	90.43%
an+cm+shi	*majority*	86.90%	83.60%	90.98%	84.09%	79.78%	88.25%
an+cm+shi	*switching*	86.01%	82.50%	90.29%	82.58%	77.94%	87.13%
dm+cm+shi	*majority*	90.35%	87.21%	93.13%	88.07%	83.64%	90.79%
dm+cm+shi	*switching*	90.27%	87.11%	93.03%	87.99%	83.52%	90.59%
an+tutti	*majority*	90.30%	87.26%	93.25%	88.36%	84.13%	91.05%
an+tutti	*switching*	89.70%	86.45%	92.67%	87.46%	83.06%	90.32%
dm+tutti	*majority*	90.64%	87.60%	**93.46**%	**88.51**%	**84.42**%	**91.29**%
dm+tutti	*switching*	**90.65**%	**87.62**%	93.38%	88.50%	84.20%	91.04%
DM17 (baseline)		89.82%	85.96%	91.77%	87.59%	81.95%	88.77%

Tabella 4.5: Ensemble ottenuti applicando le strategie di switching e majority sul development e test set a partire dai modelli appresi nel setup 2. La baseline è data dal miglior parser di base, ovvero Dozat and Manning (2017).

(dm+tutti) che utilizza come primo parser il migliore. Per capire quanto è sostenibile la strategia majority riportiamo il numero di alberi che non è ben formato in Tabella 4.6.

Nel caso della combinazione dei peggiori parser (an+cm+shi) la media degli alberi malformati può arrivare anche al 7.9%, quasi un albero su dieci non è ben formato. Se consideriamo, invece, la combinazione dei migliori parser (dm+ch+ba), la media scende al 2.5%. Inoltre, dai dati emerge che aumentare il numero dei parser considerati nel voto fa aumentare anche il numero di alberi malformati.

	Ita		Ita+PoSTWITA		
votanti	**Dev**	**Test**	**Dev**	**Test**	**media**
dm+ch+ba	9/564	7/482	31/1235	31/674	2.5%
an+cm+shi	45/564	25/482	88/1235	77/674	7.9%
dm+cm+shi	6/564	6/482	19/1235	23/674	1.8%
an+tutti	18/564	17/482	73/1235	63/674	5.5%
dm+tutti	17/564	11/482	75/1235	57/674	5.0%

Tabella 4.6: Numero di alberi malformati ottenuti usando la strategia majority con la relativa media calcolata sul development e test set di entrambi i corpora.

4.2 Reparsing

Un altro approccio al voting, proposto in Sagae and Lavie (2006), è quello del *reparsing*. In questo approccio si prova a superare la limitazione della strategia majority, relativa alla possibilità di generare alberi non ben formati, sfruttando gli algoritmi di parsing. Questa tecnica prevede di costruire un grafo diretto con tutti gli archi candidati distinti, dove il peso è ottenuto a partire dai voti. Infine, sul grafo si applicano le classiche tecniche di parsing usando un algoritmo MST che genera un dependency tree ben formato.

Considereremo per gli esperimenti due tipi di algoritmi MST. Il primo è l'algoritmo Chu-Liu/Edmonds, che effettua una ricerca sull'intero spazio dei dependency tree non-projective, quindi l'albero generato può essere anche non-projective. Per questo tipo di approccio possiamo usare anche dei metodi per pesare il voto, che influenzerà il peso attribuito all'arco nel grafo. Utilizzeremo tre diversi metodi di peso del voto proposti in Hall et al. (2007):

- *w2*, tutti i parser hanno lo stesso peso;
- *w3*, i parser sono pesati in base alla labeled accuracy;
- *w4*, i parser sono pesati in base alla labeled accuracy relativamente al POS tag.

Come osservato in McDonald et al. (2005), effettuare una ricerca sull'intero spazio dei dependency tree non-projective può essere a volte controproducente. Nonostante alcune lingue permettano relazioni non-projective, sono comunque per la maggior parte di tipo projective. Quindi, cercando tra tutti gli alberi non-projective, si corre il rischio di trovare dependency tree non desiderabili anche se ben formati. Ad esempio il training set del corpus UD Italian contiene 564 dependency tree non-projective sull'intero corpus di 12838 dependency tree, ovvero circa il 4.4%. Per questo motivo consideriamo un secondo algoritmo MST che effettua una ricerca solo sui dependency tree projective, ovvero l'algoritmo di Eisner.

Per gli esperimenti di reparsing considereremo solamente i migliori tre parser (dm+ch+ba) e tutti i parser (l'ordine in questo caso non è importante). In Tabella 4.7 sono riportati i risultati relativi al setup 0.

		Development			**Test**		
votanti	**strategia**	UAS	LAS	LS	UAS	LAS	LS
dm+ch+ba	*cle-w2*	93.82%	91.85%	95.87%	93.54%	91.83%	96.23%
dm+ch+ba	*cle-w3*	93.89%	91.82%	95.78%	93.78%	92.06%	96.20%
dm+ch+ba	*cle-w4*	94.20%	92.28%	96.13%	93.72%	92.04%	96.31%
dm+ch+ba	*eisner*	94.05%	92.05%	95.99%	93.46%	91.78%	96.16%
tutti	*cle-w2*	**94.31**%	92.53%	96.48%	93.85%	92.23%	96.41%
tutti	*cle-w3*	94.16%	92.41%	96.47%	**94.00**%	**92.48**%	96.50%
tutti	*cle-w4*	94.29%	**92.58**%	**96.54**%	93.95%	92.38%	**96.52**%
tutti	*eisner*	**94.31**%	92.53%	96.41%	93.95%	92.35%	96.33%
DM17 (baseline)		93.74%	91.66%	95.78%	93.75%	92.03%	96.19%

Tabella 4.7: Ensemble ottenuti applicando gli algoritmi di parsing Chu-Liu/Edmonds (cle) pesato e Eisner sul development e test set, a partire dai modelli appresi nel setup 0. La baseline è data dal miglior parser di base, ovvero Dozat and Manning (2017).

Nel caso della combinazione dei tre migliori parser (dm+ch+ba), dai risul-

tati emerge che l'algoritmo Chu-Liu/Edmonds con metodo di pesi *w4* riporta risultati simili a *w3* e migliori rispetto a *w2*. Quest'ultimo metodo di peso del voto mostra prestazioni peggiori rispetto al *w2*. L'algoritmo di Eisner ha, grossomodo, gli stessi risultati di Chu-Liu/Edmonds nel caso del voto senza peso, cioè *w2*. La differenza tra i vari schemi di peso e i due algoritmi si riduce quando li consideriamo tutti, e non solo i migliori tre parser. Ciò indica che la quantità di parser usati nell'ensemble, nel caso del reparsing, è importante e rende la scelta del metodo di peso quasi ininfluente sul risultato finale.

In Tabella 4.8 sono riportati i risultati del reparsing sul setup 2.

		Development			**Test**		
votanti	**strategia**	UAS	LAS	LS	UAS	LAS	LS
dm+ch+ba	*cle-w2*	90.33%	86.95%	92.72%	87.69%	83.31%	90.62%
dm+ch+ba	*cle-w3*	89.82%	85.96%	91.77%	87.59%	81.95%	88.77%
dm+ch+ba	*cle-w4*	90.41%	86.99%	92.81%	87.94%	83.32%	90.53%
dm+ch+ba	*eisner*	90.50%	87.05%	92.82%	88.04%	83.51%	90.72%
tutti	*cle-w2*	**90.52**%	**87.53**%	**93.51**%	**88.36**%	**84.25**%	**91.29**%
tutti	*cle-w3*	89.90%	86.75%	93.03%	87.79%	83.54%	91.06%
tutti	*cle-w4*	90.42%	87.46%	93.46%	88.19%	84.11%	**91.29**%
tutti	*eisner*	90.45%	87.41%	93.35%	88.31%	84.08%	91.02%
DM17 (baseline)		89.82%	85.96%	91.77%	87.59%	81.95%	88.77%

Tabella 4.8: Ensemble ottenuti applicando gli algoritmi di parsing Chu-Liu/Edmonds (cle) pesato e Eisner sul development e test set a partire dai modelli appresi nel setup 2. La baseline è data dal miglior parser di base, ovvero Dozat and Manning (2017).

In questo caso l'algoritmo di Chu-Liu/Edmonds con schema di peso *w2* restituisce i risultati migliori. Nel caso in cui consideriamo solo i migliori tre parser (dm+ch+ba), lo schema migliore è *w4*, mentre se consideriamo tutti i parser il migliore è *w2*. In Tabella 4.9 sono riportate le percentuali degli

alberi non-projective, utilizzando l'algoritmo Chu-Liu/Edmonds confrontate con i treebank development e test di ciascun corpus.

		Ita		Ita+PoSTWITA		
votanti	**strategia**	**Dev**	**Test**	**Dev**	**Test**	**media**
dm+ch+ba	*cle-w2*	66/564	50/482	145/1235	109/674	12.5%
dm+ch+ba	*cle-w3*	67/564	51/482	69/1235	37/674	8.4%
dm+ch+ba	*cle-w4*	61/564	46/482	109/1235	77/674	10.2%
tutti	*cle-w2*	35/564	28/482	116/1235	58/674	7.5%
tutti	*cle-w3*	45/564	28/482	107/1235	62/674	7.9%
tutti	*cle-w4*	36/564	24/482	108/1235	51/674	6.9%
gold standard		31/564	21/482	46/1235	18/674	4.1%

Tabella 4.9: Numeri di dependency tree non-projective nell'approccio reparsing usando l'algoritmo Chu-Liu/Edmonds reparsing con diversi pesi.

Dai dati emerge che maggiore è il numero di votanti, minore è la percentuale dei dependency tree non-projective. Nonostante ciò la media degli alberi non-projective generati supera di molto quella dei gold treebank, che è di circa il 4.1%. Questo potrebbe portare a preferire l'algoritmo di Eisner, anche se in questo caso si avrebbero solo alberi projective.

4.3 Distilling

Recentemente, in Kuncoro et al. (2016), è stato proposto un approccio per la costruzione di un ensemble basato su voting in seguito definito *distilling*, che permette il training di un singolo modello a partire da diversi parser addestrati indipendentemente. La tecnica consiste nell'apprendere un singolo modello di parsing a partire da una matrice di costi basata sui voti degli m parser, usando una particolare funzione di costo che considera l'incertezza della predizione. L'idea alla base è che il disagreement, ovvero quando due

parser non concordano sullo stesso arco, può essere un segnale che la relazione in causa è ambigua. Allenare un modello distilling richiede parecchio tempo, ma ha il vantaggio di creare un modello finale che non ha bisogno di interrogare i singoli parser da cui è stato costruito. Negli ensemble considerati finora il tempo di composizione del dependency tree finale dipende dalla somma dei tempi di parsing degli m parser tutte le volte che dobbiamo costruire l'ensemble. Nel caso del distilling il modello viene creato una sola volta a partire dagli m parser che non saranno più usati.

4.4 Confronto

I risultati sono riportati in Tabella 4.10 per il setup 0 e in Tabella 4.11 per il setup 2, insieme ai migliori risultati degli approcci visti finora relativamente al test set.

strategia	UAS	LAS	LS
majority	93.94% (+0.19%)	92.41% (+0.38%)	96.52% (+0.33%)
switching	93.91% (+0.16%)	92.37% (+0.34%)	96.50% (+0.31%)
cle	94.00% (+0.25%)	92.48% (+0.45%)	96.50% (+0.31%)
eisner	93.95% (+0.20%)	92.35% (+0.32%)	96.33% (+0.14%)
distilling	92.50% (–1.25%)	89.93% (–2.10%)	94.75% (–1.44%)

Tabella 4.10: In tabella sono riportati i risultati migliori ottenuti per le diverse strategie di ensemble sul test set del setup 0. Tra parentesi sono indicati gli incrementi relativamente alla baseline.

L'approccio distilling mostra risultati più bassi rispetto alla baseline e, in generale, più bassi rispetto agli altri metodi. I migliori risultati si ottengono tramite l'approccio majority che porta con sé il difetto di non garantire se l'albero generato sia ben formato. La differenza tra i due setup è che nel setup 0 i miglioramenti non superano la soglia del 0.5%, mentre nel setup 2 la superano e arrivano al 2.5% di guadagno in LAS. Per quanto riguarda i metodi

strategia	UAS	LAS	LS
majority	88.51% (+0.92%)	84.42% (+2.47%)	91.29% (+2.52%)
switching	88.50% (+0.91%)	84.20% (+2.25%)	91.04% (+2.27%)
cle	88.36% (+0.77%)	84.25% (+2.30%)	91.29% (+2.52%)
eisner	88.31% (+0.72%)	84.08% (+2.13%)	91.02% (+2.25%)
distilling	86.73% (–0.86%)	81.39% (–0.56%)	89.19% (+0.42%)

Tabella 4.11: In tabella sono riportati i risultati migliori ottenuti per le diverse strategie di ensemble sul test set del setup 2. Tra parentesi sono indicati gli incrementi relativamente alla baseline.

di reparsing, sia nel setup 2 che nel setup 0 l'algoritmo di Chu-Liu/Edmonds risulta essere leggermente migliore di quello di Eisner. I metodi, in linea di massima, restituiscono tutti lo stesso miglioramento, con oscillazioni di pochi punti decimali.

Pertanto, la scelta della strategia è dovuta, in parte, alle proprietà che desideriamo nell'albero finale. Se non siamo interessati alla correttezza della forma dell'albero, perché corretto manualmente, possiamo usare tranquillamente il majority. Se invece volessimo un albero corretto e avessimo un parser con buone prestazioni, potremmo usare lo switching usando questo come primo parser e fornendo se possibile altri modelli, anche più di tre. Se la lingua è prevalentemente strutturata in modo da contenere molte forme non-projective, potremmo voler indirizzare la nostra scelta sul reparsing con l'algoritmo Chu-Liu/Edmonds o, se non ci interessa, su Eisner. Per quanto riguarda il distilling, non è risultato essere un buon metodo da usare per modelli ottenuti da diversi parser ma, come suggerisce l'autore, potrebbe essere utile quando consideriamo più modelli ottenuti a partire da diverse istanze di uno stesso parser.

Conclusioni

All'interno di questo lavoro si è mostrato come apprendere modelli di parsing a partire da corpora di dominio social media (UD Italian PoSTWITA 2.2) produca prestazioni significativamente migliori nel parsing su testi dello stesso dominio rispetto a modelli appresi su corpora di dominio generico (UD Italian 2.1). L'apprendimento dei modelli su UD Italian PoSTWITA 2.2 produce un incremento medio di circa 8.3% in UAS e 10% in LAS rispetto a UD Italian 2.1. Utilizzando come corpus di apprendimento la fusione dei due corpora precedenti si ottengono ulteriori miglioramenti, circa 1.7% in UAS e 2% in LAS, rispetto ai modelli appresi sul solo corpus UD Italian PoSTWITA 2.2 di dominio social media. Ciò indica che maggiore è il numero di dati presenti nel corpus, inclusi i dati di dominio, migliori saranno le prestazioni dei parser.

Tra gli otto parser presi in considerazione per gli esperimenti il migliore in tutti i setup è risultato essere quello di Dozat and Manning (2017), che ha una valutazione molto vicina allo stato dell'arte per il parsing della lingua italiana. I risultati ottenuti nei precedenti esperimenti sono stati pubblicati in Sanguinetti et al. (2018).

Inoltre, si è mostrato come tramite tecniche di ensemble si possono combinare i modelli dei singoli parser in modo da ottenere risultati migliori del miglior modello singolo. L'incremento sul corpus, ottenuto dalla fusione dei due treebank, è vicino al 2.5% in LAS e 1% in UAS, mentre per il corpus UD Italian 2.1 poco meno dello 0.5% in LAS e 0.25% in UAS. Il maggior guadagno in LAS rispetto allo UAS fa emergere come negli alberi predetti

dei modelli ensemble sia più facile individuare il tipo corretto della relazione di dipendenza piuttosto che la testa.

Avendo a disposizione differenti parser, è stato possibile condurre esperimenti effettuando diverse combinazioni tra loro. Dagli esperimenti è emerso che l'accuratezza dei singoli parser è tanto importante quanto il loro numero nel modello ensemble per ottenere buone prestazioni, infatti la combinazione dei migliori tre parser ha prestazioni inferiori rispetto alla combinazione che li utilizza tutti, sia per l'approccio voting che quello di reparsing. Per quanto riguarda la tecnica di distilling, questa non riesce ad ottenere risultati migliori rispetto al miglior parser singolo. La tecnica di voting e quella di reparsing mostrano risultati simili tra loro rendendo, così, la scelta di una tecnica piuttosto che un'altra legata a specifiche proprietà che si vogliono considerare nel dependency tree finale.

Appendice A

Sviluppo container per training e testing

Per rendere più rapida l'esecuzione di apprendimento e test dei modelli di parsing si è costruito un ambiente virtuale ad hoc in modo da automatizzare i due processi per ciascun parser a partire da diversi treebank.

Per costruire l'ambiente si è utilizzata la piattaforma *Docker* attraverso la quale si possono sviluppare i cosiddetti *Docker container*, i quali permettono la rapida installazione, distribuzione ed esecuzione di applicativi su qualunque sistema sfruttando la virtualizzazione. Un *container* è un'unità software stand-alone che include tutte le componenti di cui ha bisogno per eseguire il codice al suo interno in un ambiente completamente isolato ma coesistente con il sistema che lo ospita. Il container pur sfruttando la virtualizzazione non è una virtual machine, ma può essere visto come un unico pacchetto software che svolge particolari compiti per cui è stato costruito, permettendo di risolvere problemi legati all'installazione di requisiti software sui diversi sistemi che lo ospitano.

L'immagine del container, costruita per avviare gli esperimenti può essere installata ed eseguita utilizzando la Community Edition[1] della piattaforma Docker, scaricabile gratuitamente e disponibile per i sistemi operativi più po-

[1]Disponibile su `https://www.docker.com/community-edition`

polari: Windows, Linux (per le maggiori distribuzioni) e Mac. Per installare la Community Edition su Ubuntu si procede eseguendo i comandi

```
$ sudo apt-get update
$ sudo apt-get install \
       apt-transport-https \
       ca-certificates \
       curl \
       software-properties-common
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
   ↪ sudo apt-key add -
$ sudo apt-key fingerprint 0EBFCD88
$ sudo add-apt-repository \
       "deb [arch=amd64] https://download.docker.com/linux/
          ↪ ubuntu \
       $(lsb_release -cs) \
       stable"
$ sudo apt-get update
$ sudo apt-get install docker-ce
```

La versione usata per eseguire gli esperimenti è la 17.09.1-ce. Per utilizzare la piattaforma Docker correttamente è necessario eseguire tutti i comandi `docker` come amministratore del sistema.

Il container è un'istanza runtime dell'immagine di un ambiente virtuale definito tramite un file di testo, di nome *Dockerfile*, che contiene tutte le informazioni per la sua creazione[2]. Il *Dockerfile* che definisce l'immagine per l'ambiente che permette di eseguire le procedure di training e testing è contenuto nella cartella `docker` e mostrato in Figura A.1.

L'installazione dell'immagine, che chiameremo **dockerparser**, sul sistema locale si effettua tramite l'esecuzione del comando `build` nella cartella

[2]Per informazioni dettagliate sui comandi usati per costruire il container tramite *Dockerfile* visitare `https://docs.docker.com/engine/reference/builder/`

docker/Dockerfile

```
1  FROM ubuntu:16.04
2
3  ENV PARSERDIR=/root/parser
4
5  RUN apt-get clean && apt-get update && apt-get install -y locales
6  RUN locale-gen it_IT.UTF-8
7  ENV LANG it_IT.UTF-8
8  ENV LANGUAGE it
9  ENV LC_ALL it_IT.UTF-8
10
11 COPY build.sh build.sh
12 RUN bash build.sh && rm build.sh
13
14 ENV LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib:/opt/OpenBLAS/lib
15
16 COPY launcher.sh /root/launcher.sh
17 COPY config /root/config
18 COPY utils /root/utils
19
20 WORKDIR /root
21
22 ENTRYPOINT ["/root/launcher.sh"]
```

Figura A.1: Dockerfile che definisce l'immagine del container usata negli esperimenti.

che contiene il *Dockerfile*[3]

```
~$ cd docker
~/docker$ sudo docker build -t dockerparser .
```

In alternativa, è possibile caricare direttamente l'archivio dell'immagine usando il comando:

```
~$ sudo docker load < dockerparser.tar
```

[3]L'installazione dell'immagine può durare diverse ore e necessita che il sistema sia connesso stabilmente ad internet.

La prima riga del *Dockerfile* ci permette di definire come immagine di base dell'ambiente virtuale il sistema operativo Ubuntu 16.04. Dalla riga 5 alla 9 sono installati alcuni pacchetti software per il supporto ai file in formato UTF-8 utili per leggere i treebank. La procedura che installa tutti i parser e i requisti software nell'ambiente virtuale Ubuntu è contenuta nello script `build.sh` che viene copiato alla riga 11 dalla cartella del sistema locale a quella dell'ambiente virtuale, eseguito e infine rimosso. Dalla riga 16 alla 18 vengono copiati un file e due cartelle sull'immagine virtuale che serviranno per l'esecuzione degli esperimenti:

- Lo script `launcher.sh` si occupa di preparare e avviare le procedure di training e testing per ciascun parser facendo uso degli script presenti nelle cartelle `config` e `utils`. I modelli generati sono salvati nella cartella `models`, mentre le predizioni nella cartella `preds` del container.

- La cartella `config` contiene tutti gli script, suddivisi per parser, che definiscono la configurazione di avvio del parser per le procedure di training e testing. Modificando opportunamente gli script forniti di default si possono modificare le opzioni per training (tra cui gli iperparametri) e testing.

- La cartella `utils` contiene alcuni script utili per lavorare sui treebank.

Alla riga 22 lo script `launcher.sh` è impostato come punto d'accesso di default dell'ambiente, in questo modo lo script si avvia ogni volta che si esegue un container. Questo ci permette di creare un ponte che permetta di eseguire le procedure di training e testing senza abbandonare l'ambiente del sistema locale.

Per organizzare meglio un esperimento si consiglia di creare un'apposita cartella sul sistema locale in cui inserire i treebank in formato CoNLL da cui apprendere i modelli, nella divisione train/dev/test, e un file per le word embedding di dimensione 100 di default (Figura A.2a).

```
setup1
├── UD_Italian-PoSTWITA-2.2
│   ├── it_postwita-ud-train.conll
│   ├── it_postwita-ud-dev.conll
│   └── it_postwita-ud-test.conll
├── embed
│   └── postwita-2.2.txt
└── start.sh
```

(a) Cartella sul sistema locale prima di avviare gli esperimenti del Setup 1.

```
root
├── corpus
│   ├── train.conll
│   ├── dev.conll
│   └── test.conll
├── embed
│   └── embed.txt
├── config
│   ├── ChenManning2014
│   └── ...
├── parser
│   ├── ChenManning2014
│   └── ...
└── launcher.sh
```

(b) Contenuto dell'immagine del container dockerparser con la cartella `corpus` e `embed` montate.

```
setup1
├── UD_Italian-PoSTWITA-2.2
├── embed
├── models
│   ├── ChenManning2014
│   │   └── nndep.model.txt.gz
│   └── ...
├── preds-test
│   ├── ChenManning2014.out
│   └── ...
├── preds-dev
│   ├── ChenManning2014.out
│   └── ...
└── start.sh
```

(c) Cartella del Setup 1 dopo l'apprendimento dei modelli e del testing.

Figura A.2: Contenuto delle cartelle nel sistema locale prima (a) e dopo l'apprendimento (c) avviato tramite lo script `start.sh` (Figura A.3). In (b) i file nelle cartelle `corpus` e `embed` del container sono gli stessi di (a) ma montanti nell'ambiente virtuale con i nomi che lo script `launcher.sh` si aspetta.

Per avviare il container senza errori è necessario montare correttamente i treebank e il file di embedding all'interno dell'ambiente virtuale in modo che

possano essere usati correttamente dallo script `launcher.sh`.

Per montare dei file o cartelle dal sistema locale al container si usa il parametro `-v` composto da due campi separati dal carattere : (due punti). Il primo campo è il nome del file o cartella sul sistema locale, mentre il secondo campo è il percorso in cui il file o cartella sarà montato nel sistema virtuale del container.

Per avviare correttamente la procedura di apprendimento, è necessario eseguire il comando `run` e montare correttamente i file all'interno del container:

```
$ sudo docker run \
       -v <train-file>:/root/corpus/train.conll \
       -v <dev-file>:/root/corpus/dev.conll \
       -v <test-file>:/root/corpus/test.conll \
       -v <embed-file>:/root/embed/embed.txt \
       -v <models-folder>:/root/models \
dockerparser train [cm,ba,kgt,kgg,an,ch,dm,shi,ng]
```

I parametri a sinistra e destra del carattere : fanno riferimento rispettivamente ai percorsi assoluti sul sistema locale e sul container.

I file che lo script `launcher.sh` si aspetta hanno una posizione e un nome fisso, per questo motivo i file e le cartelle a destra del carattere : sono fissi per qualunque esperimento, mentre quelli a sinistra possono variare. Lo script `launcher.sh` ha bisogno che i treebank abbiano nome `train.conll`, `dev.conll` e `test.conll` e siano posizionati nella cartella `/root/corpus`; mentre il file delle word embedding ha nome `embed.txt` e deve trovarsi nella cartella `/root/embed` (Figura A.2b).

Se si vuole fare a meno delle word embedding preaddestrate è sufficiente che il file di testo `embed.txt` sia vuoto. In alternativa si può escludere l'uso delle word embedding direttamente dalla configurazione di avvio dello specifico parser modificando lo script di apprendimento nella cartella `config`. Se si vuole avviare il training per uno specifico insieme parser, invece che per

tutti, è sufficiente passare la lista dei codici dei parser (Tabella A.1) separati da virgola dopo il comando `train`.

Parser	**Cartella**	**Codice**
CM14	`ChenManning2014`	`cm`
BA15	`Ballesteros2015`	`bm`
KG16:T	`KiperwasserGoldberg2016/barchybrid`	`kgt`
KG16:G	`KiperwasserGoldberg2016/bmstparser`	`kgg`
AN16	`Andor2016`	`an`
CH16	`Cheng2016`	`ch`
DM17	`DozatManning2016`	`dm`
SH17	`Shi2017`	`shi`
NG17	`Nguyen2017`	`ng`

Tabella A.1: Nomi delle cartelle usate per dividere i file relativi ai parser. Nell'ultima colonna il codice può essere usato quandosi avvia il container per specificare quale/i parser apprendere o testare.

Il comando per il testing è molto simile a quello per il training con la differenza che il parametro da passare al container è `test` anziché `train` e c'è bisogno di montare un'ulteriore cartella di nome `preds` dove saranno salvate le predizioni, in formato CoNLL, restituite dai parser:

```
$ sudo docker run \
       -v <train-file>:/root/corpus/train.conll \
       -v <dev-file>:/root/corpus/dev.conll \
       -v <test-file>:/root/corpus/test.conll \
       -v <embed-file>:/root/embed/embed.txt \
       -v <models-folder>:/root/models \
       -v <preds-folder>:/root/preds \
```

```
dockerparser test [cm,ba,kgt,kgg,an,ch,dm,shi,ng]
```

Usando i template dei comandi di training e testing riportati sopra creiamo uno script `start.sh` che avvia i due processi per i file del setup di Figura A.2a; lo script è riportato in Figura A.3.

setup1/start.sh

```
1  #!/bin/bash
2
3  # Training di tutti i parser
4  sudo docker run \
5  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-train.conll:/root/corpus/train.conll \
6  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-dev.conll:/root/corpus/dev.conll \
7  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-test.conll:/root/corpus/test.conll \
8  -v $(pwd)/embed/postwita-2.2.txt:/root/embed/embed.txt \
9  -v $(pwd)/models:/root/models \
10 dockerparser train
11
12 # Testing sul dev set
13 sudo docker run \
14 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-train.conll:/root/corpus/train.conll \
15 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-dev.conll:/root/corpus/dev.conll \
16 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-dev.conll:/root/corpus/test.conll \
17 -v $(pwd)/embed/postwita-2.2.txt:/root/embed/embed.txt \
18 -v $(pwd)/models:/root/models \
19 -v $(pwd)/preds-dev:/root/preds \
20 dockerparser test
21
22 # Testing sul test set
23 sudo docker run \
24 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-train.conll:/root/corpus/train.conll \
25 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-dev.conll:/root/corpus/dev.conll \
26 -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-test.conll:/root/corpus/test.conll \
27 -v $(pwd)/embed/postwita-2.2.txt:/root/embed/embed.txt \
28 -v $(pwd)/models:/root/models \
29 -v $(pwd)/preds-test:/root/preds \
30 dockerparser test
```

Figura A.3: Lo script che avvia il processo di training (righe 3-9) e testing (righe 11-18 per il dev set e righe 20-27 per il test set) che fa uso del comando `-v` per montare le cartelle memorizzate sul sistema locale nell'immagine del container da eseguire.

Alla riga 4 lo script `start.sh` monta il treebank memorizzato in locale `it_postwita-ud-train.conll` col nome `train.conll` all'interno del percorso `/root/corpus` del sistema virtuale del container. Lo stesso viene fatto, nelle righe da 5 a 7, per i file relativi a development set, test set e word embedding. Alla riga 8 viene montata la cartella `models` nella quale il container salverà i modelli appresi. L'uso del comando `pwd` serve per restituire il percorso assoluto della directory corrente; questo perché la piattaforma Docker lavora solo su percorsi assoluti. Al termine della fase di apprendimento[4] i modelli saranno salvati tutti nella cartella `models` nel sistema locale e potranno essere utilizzati in seguito per effettuare le predizioni.

Per effettuare le predizioni sia sul development set che sul test set è necessario scrivere un `run` per ognuno. Poiché il container è costruito in modo da restituire le predizioni del file con nome `test.conll`, per generare le predizioni sul development set dobbiamo montare il dev treebank come `test.conll` (riga 14 di Figura A.3). Ciò che è importante ricordare è che i nomi e i percorsi dei file a destra del carattere `:` sono fissi indipendentemente dal setup perché sono quelli che lo script `launcher.sh` si aspetta di trovare, mentre i nomi a sinistra del carattere `:` possono essere diversi da setup a setup.

Dopo aver creato lo script dell'esperimento di un particolare setup, per eseguirlo è sufficiente digitare il comando

```
~/setup1$ ./start.sh
```

che avvierà l'apprendimento dei modelli e in seguito le predizioni. Al termine dello script, nella macchina locale, saranno state create le cartelle `models`, `preds-dev` e `preds-test` che conterranno rispettivamente i modelli appresi, le predizioni per il development set e quelle per il test set (Figura A.2c). I file di predizione potranno essere usati per effettuare le valutazioni delle prestazioni usando appositi strumenti di valutazione.

Grazie all'uso del container si possono eseguire le procedure di training e testing sui diversi parser usando un unico comando nella shell del sistema

[4]Sono necessari diversi giorni per completare l'apprendimento di tutti i parser.

locale, senza entrare direttamente in quello virtuale. Se si ha necessità di accedere all'interno dell'ambiente dell'immagine è possibile farlo inserendo i parametri `-it` di seguito al comando `run`:

```
$ sudo docker run -it dockerparser
root@773c713506a4:~#
```

In questo modo è possibile usare interattivamente, tramite shell, il sistema virtuale del container, come se fossimo in un normale ambiente Linux:

```
root@773c713506a4:~# ls
config launcher.sh parser utils
```

Se si vuole avviare una sessione interattiva in cui sono montati alcuni file o cartelle è sufficiente seguire lo stesso schema presente in Figura A.3 aggiungendo, dopo i parametri `-it`, la lista dei file da montare tramite il parametro `-v`.

Poiché un container è un'istanza runtime dell'immagine dopo aver effettuato il logout, tutti i file salvati all'interno del container saranno persi, a meno che non si memorizzino su cartelle montante dal sistema locale tramite il parametro `-v`.

A.1 Personalizzare il training o il testing

Supponiamo di voler modificare qualche parametro nello script di learning per il parser Chen and Manning (2014). Per farlo è sufficiente creare una copia della cartella `docker/config/ChenManning2014`, fornita come default, nella cartella dell'esperimento:

```
~$ cp -R docker/config/ChenManning2014/ setup1/config/
   ↪ myChenManning2014
```

Per ogni cartella in `config` di ciascun parser ci sono almeno due file, uno script per il train e l'altro per il test. Modificandoli nelle opportune parti è possibile gestire diverse configurazioni del parser differenti da quella di default, che viene automaticamente eseguita quando si lancia la procedura.

Lo script `train.sh` di default relativo alla procedura di training del parser Chen and Manning (2014) è mostrato in Figura A.4. Se volessimo

config/ChenManning2014/train.sh

```
1  #!/bin/bash
2
3  train=$1
4  dev=$2
5  model=$3
6  embed=$4
7
8  java -Xmx6g -cp stanford-corenlp-3.8.0.jar \
9         edu.stanford.nlp.parser.nndep.DependencyParser \
10        -hiddenSize 200 \
11        -adaAlpha 0.01 \
12        -embedFile ${embed} \
13        -embeddingSize 100 \
14        -trainFile ${train} \
15        -devFile ${dev} \
16        -model ${model}
```

Figura A.4: Lo script `train.sh` di default che specifica i parametri del modello per il parser Chen and Manning (2014).

aumentare la memoria massima allocabile dalla Java Virtual Machine durante l'apprendimento a 8GB ci basterebbe cambiare il parametro `-Xmx6g` alla riga 8 in `-Xmx8g`. Se invece volessimo cambiare la dimensione del livello nascosto da 200 a 300 ci basterebbe cambiare il valore del parametro `-hiddenSize`. Nell'aggiungere o modificare i parametri di un parser è opportuno fare riferimento alla guida d'uso presente nella repository.

Una volta modificati i parametri di nostro interesse per uno specifico parser è necessario aggiungere una riga al comando `docker run` (riga 8 della Figura A.5) per montare, tramite il parametro `-v`, la configurazione del parser

personalizzata nella posizione di quella di default dell'ambiente virtuale, in modo da sovrascriverla.

setup1/start-cm+dm.sh

```
1  #!/bin/bash
2
3  sudo docker run \
4  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-train.conll:/root/corpus/train.conll \
5  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-dev.conll:/root/corpus/dev.conll \
6  -v $(pwd)/UD_Italian-PoSTWITA-2.2/it_postwita-ud-test.conll:/root/corpus/test.conll \
7  -v $(pwd)/embed/postwita-2.2.txt:/root/embed/embed.txt \
8  -v $(pwd)/config/myChenManning2014:/root/config/ChenManning2014 \
9  -v $(pwd)/models:/root/models \
10 dockerparser train cm,dm
```

Figura A.5: Script che apprende solo i modelli per i parser Chen and Manning (2014) e Dozat and Manning (2017) poiché i loro codici `cm` e `dm` compaiono separati da virgola dopo il comando `train`. Per il parser Chen and Manning (2014) è usata una configurazione personalizzata (riga 8), che varia da quella di default, ma che è montata nella stessa posizione in modo da sovrascriverla.

Bibliografia

Andor, D., Alberti, C., Weiss, D., Severyn, A., Presta, A., Ganchev, K., Petrov, S., and Collins, M. (2016). Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2442–2452, Berlin, Germany. Association for Computational Linguistics.

Attardi, G., Saletti, S., Simi, M., and Lavelli, A. (2015). Evolution of italian treebank and dependency parsing towards universal dependencies. In *Proceedings of the Second Italian Conference on Computational Linguistics CLiC-it 2015: 3-4 December 2015, 2015*. Torino: Accademia University Press.

Ballesteros, M., Dyer, C., and Smith, N. A. (2015). Improved transition-based parsing by modeling characters instead of words with lstms. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 349–359, Lisbon, Portugal. Association for Computational Linguistics.

Baroni, M., Bernardini, S., Ferraresi, A., and Zanchetta, E. (2009). The wacky wide web: a collection of very large linguistically processed web-crawled corpora. *Language Resources and Evaluation*, 43(3):209–226.

Bosco, C., Lombardo, V., Vassallo, D., and Lesmo, L. (2000). Building a treebank for italian: a data-driven annotation schema. In *In Proceedings of the Second International Conference on Language Resources and Evaluation LREC-2000*, pages 99–105.

Bosco, C., Montemagni, S., and Simi, M. (2013). Converting italian treebanks: Towards an italian stanford dependency treebank. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 61–69, Sofia, Bulgaria. Association for Computational Linguistics.

Buchholz, S. and Marsi, E. (2006). Conll-x shared task on multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 149–164, New York City. Association for Computational Linguistics.

Chen, D. and Manning, C. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar. Association for Computational Linguistics.

Cheng, H., Fang, H., He, X., Gao, J., and Deng, L. (2016). Bi-directional attention with agreement for dependency parsing. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2204–2214, Austin, Texas. Association for Computational Linguistics.

Choi, J. D., Tetreault, J., and Stent, A. (2015). It depends: Dependency parser comparison using a web-based evaluation tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 387–396, Beijing, China. Association for Computational Linguistics.

Clark, A., Fox, C., and Lappin, S. (2010). *The Handbook of Computational Linguistics and Natural Language Processing*. Wiley-Blackwell.

de Marneffe, M.-C., Dozat, T., Silveira, N., Haverinen, K., Ginter, F., Nivre, J., and Manning, C. D. (2014). Universal stanford dependencies: A cross-linguistic typology. In Calzolari, N., Choukri, K., Declerck, T.,

Loftsson, H., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., and Piperidis, S., editors, *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 4585–4592, Reykjavik, Iceland. European Language Resources Association (ELRA). ACL Anthology Identifier: L14-1045.

de Marneffe, M.-C. and Manning, C. D. (2008). The Stanford typed dependencies representation. In *Coling 2008: Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 1–8, Manchester, UK. Coling 2008 Organizing Committee.

Dozat, T. and Manning, C. D. (2017). Deep biaffine attention for neural dependency parsing. In *Proceedings of the 2017 International Conference on Learning Representations*.

Dozat, T., Qi, P., and Manning, C. D. (2017). Stanford's graph-based neural dependency parser at the conll 2017 shared task. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 20–30, Vancouver, Canada. Association for Computational Linguistics.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. (2015). Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343, Beijing, China. Association for Computational Linguistics.

Goldberg, Y. and Nivre, J. (2012). A dynamic oracle for arc-eager dependency parsing. In *Proceedings of COLING 2012*, pages 959–976, Mumbai, India. The COLING 2012 Organizing Committee.

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

Hall, J., Nilsson, J., Nivre, J., Eryigit, G., Megyesi, B., Nilsson, M., and Saers, M. (2007). Single malt or blended? a study in multilingual parser optimization. In *Proceedings of the CoNLL Shared Task Session of EMNLP-CoNLL 2007*, pages 933–939, Prague, Czech Republic. Association for Computational Linguistics.

Johansson, R. and Nugues, P. (2006). Investigating multilingual dependency parsing. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 206–210, New York City. Association for Computational Linguistics.

Jurafsky, D. and Martin, J. H. (2017). Speech and language processing (3rd edition draft). `https://web.stanford.edu/~jurafsky/slp3/` draft August 28, 2017.

Kiperwasser, E. and Goldberg, Y. (2016). Simple and accurate dependency parsing using bidirectional lstm feature representations. *Transactions of the Association for Computational Linguistics*, 4:313–327.

Kuhlmann, M., Gómez-Rodríguez, C., and Satta, G. (2011). Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, Oregon, USA. Association for Computational Linguistics.

Kuncoro, A., Ballesteros, M., Kong, L., Dyer, C., and Smith, N. A. (2016). Distilling an ensemble of greedy dependency parsers into one mst parser. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1744–1753, Austin, Texas. Association for Computational Linguistics.

Kübler, S., McDonald, R., and Nivre, J. (2009). Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 2(1):1–127.

Marcus, M. P., Marcinkiewicz, M. A., and Santorini, B. (1993). Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19(2):313–330.

McDonald, R., Crammer, K., and Pereira, F. (2006). Spanning tree methods for discriminative training of dependency parsers. Technical Report MS-CIS-06-11, University of Pennsylvania Department of Computer and Information Science.

McDonald, R. and Nivre, J. (2014). Recent advances in dependency parsing. Tutorial at EACL 2014: `http://stp.lingfil.uu.se/~nivre/eacl14.html`.

McDonald, R. and Pereira, F. (2006). Online learning of approximate dependency parsing algorithms. In *Proceedings of EACL*, pages 81–88.

McDonald, R., Pereira, F., Ribarov, K., and Hajic, J. (2005). Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 523–530, Vancouver, British Columbia, Canada. Association for Computational Linguistics.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013a). Efficient estimation of word representations in vector space. arXiv:1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013b). Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 3111–3119, USA. Curran Associates Inc.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Montemagni, S., Barsotti, F., Battista, M., Calzolari, N., O.Corazzari, Zampolli, A., Fanciulli, F., Massetani, M., Raffaelli, R., Basili, R., Pazienza,

M. T., Saracino, D., Zanzotto, F., Mana, N., Pianesi, F., and Delmonte, R. (2000). The italian syntactic-semantic treebank: Architecture, annotation, tools and evaluation. In *Proceedings of the COLING-2000 Workshop on Linguistically Interpreted Corpora*, pages 18–27, Centre Universitaire, Luxembourg. International Committee on Computational Linguistics.

Nguyen, D. Q., Dras, M., and Johnson, M. (2017). A novel neural network model for joint pos tagging and graph-based dependency parsing. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 134–142, Vancouver, Canada. Association for Computational Linguistics.

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT).*

Nivre, J. (2004). Incrementality in deterministic dependency parsing. In Keller, F., Clark, S., Crocker, M., and Steedman, M., editors, *Proceedings of the ACL Workshop Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57, Barcelona, Spain. Association for Computational Linguistics.

Nivre, J., de Marneffe, M.-C., Ginter, F., Goldberg, Y., Hajic, J., Manning, C. D., McDonald, R., Petrov, S., Pyysalo, S., Silveira, N., Tsarfaty, R., and Zeman, D. (2016). Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016).*

Ruder, S. (2017). An overview of gradient descent optimization algorithms. arXiv:1609.04747.

Sagae, K. and Lavie, A. (2006). Parser combination by reparsing. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, NAACL-Short '06, pages 129–132, Stroudsburg, PA, USA. Association for Computational Linguistics.

Sanguinetti, M., Bosco, C., Lavelli, A., Mazzei, A., Antonelli, O., and Tamburini, F. (2018). Postwita-ud: an italian twitter treebank in universal dependencies. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Paris, France. European Language Resources Association (ELRA).

Sanguinetti, M., Bosco, C., Mazzei, A., Lavelli, A., and Tamburini, F. (2017). Annotating italian social media texts in universal dependencies. In *Proceedings of the Fourth International Conference on Dependency Linguistics (Depling 2017)*, pages 229–239. Linköping University Electronic Press.

Shi, T., Huang, L., and Lee, L. (2017a). Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 12–23, Copenhagen, Denmark. Association for Computational Linguistics.

Shi, T., Wu, F. G., Chen, X., and Cheng, Y. (2017b). Combining global models for parsing universal dependencies. In *Proceedings of the CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 31–39, Vancouver, Canada. Association for Computational Linguistics.

Surdeanu, M. and Manning, C. D. (2010). Ensemble models for dependency parsing: Cheap and good? In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 649–652, Los Angeles, California. Association for Computational Linguistics.

Zeman, D. and Žabokrtský, Z. (2005). Improving parsing accuracy by combining diverse dependency parsers. In *Proceedings of the Ninth International Workshop on Parsing Technology*, pages 171–178, Vancouver, British Columbia. Association for Computational Linguistics.