Laboratorio di TLN prima parte

Giacometti Lorenzo[915752], Giovanale Matteo[926431], and Daniel Haures[919707]

Università degli Studi di Torino
Dipartimento di Informatica

Abstract. Implementazione di un dependency parser proiettivo per la lingua italiana.

1 Introduzione

L'analisi sintattica delle dipendenze è un componente cruciale nell'elaborazione del linguaggio naturale (NLP), poiché consente di comprendere le relazioni strutturali tra le parole in una frase.

Il presente lavoro si concentra sull'implementazione di un dependency parser proiettivo per l' italiano. I parser proiettivi, che garantiscono che le dipendenze non si incrocino, sono particolarmente adatti per lingue con strutture sintattiche meno flessibili. Dopo una breve panoramica dell'algoritmo, descriviamo in dettaglio il processo di implementazione, includendo le scelte progettuali e implementative. Successivamente, discutiamo i risultati ottenuti su un test corpus e forniamo un' analisi delle performance del nostro codice.

2 Implementazione

Il parser costruito è di tipo *transition based*, dove le varie transizioni da effettuare per il parsing sono guidate da un oracolo probabilistico, il quale è stato implementato attraverso una rete neurale Long-Short-Term-Memory che prende come feature gli embedding e i PoS delle parole valutate dal Parser.

La costruzione è stata prevista in due fasi: una fase di Training [2.1] per configurare e allenare LSTM a prevedere le mosse corrette e una fase di Evaluation [2.2] per verificare i risultati ottenuti.

Il funzionamento del nostro codice si basa sull' utilizzo di funzioni contenute nelle tre classi principali: *Dependencies*, *Oracle* e *Parser*. La classe *Dependencies* [2.3] contiene la struttura dati con cui vengono rappresentate le dipendenze della frase. La classe *Oracle* [2.4] implementa al suo interno la costruzioni delle features, l'encoder BERT e l'oracolo sviluppato tramite l' LSTM in Pytorch. La classe *Parser* [2.5] implementa sia funzioni relative alla parsing della frase sia una funzione che permette di effettuare *reverse enegineering* di una frase già etichettata, utile alla fase di training.

2.1 Fase di Training

La fase di training si basa sull' addestramento di un modello LSTM per il parsing delle dipendenze, utilizzando gli embedding ottenuti dall' encoder BERT. L' addestramento fa si che che il modello LSTM impari a predire le mosse di parsing corrette.

Prima Parte Le funzione utilizzate sono *encode_ moves()* e *create_ batches()*. La prima consente attraverso la funzione *simulate_ parse()* del Parser, di ritrovare le *mosse* da effettuare al fine di ricostruire le dipendenze di una frase nel training set.

Per ogni frase avremo quindi una serie di mosse, di ogni mossa l'oracolo considera gli ultimi tre elementi dello stack e i primi tre del buffer. Le *features* saranno perciò composte dall' embedding e dal PoS degli elementi descritti.

La seconda funzione *create_ batches()* consente di salvare a gruppi di N frasi, le loro *features* e *mosse* effettuate.

Seconda Parte La seconda parte consiste nel fine-tuning degli iper-parametri della rete LSTM tramite l'addestramento *train_ on_ batches()* sui file salvati precedentemente e la stima dell'accuracy sul development set.

2.2 Fase di Evaluation

La fase di valutazione del codice si concentra sull'applicazione del modello addestrato per analizzare nuove frasi e determinare l' accuratezza del parsing a dipendenze. La funzione *predicted_ on_ test()* consente di costruire un file conllu contenente le dipendenze predette per ogni frase.

2.3 Dependencies

In questa classe sono contenute le funzioni per andare a salvare, gestire le dipendenze create dal nostro parser e controllare se esiste un arco di dipendenza da head a child.

2.4 Oracle

In questa classe di oggetti sono contenute tutte le funzioni per andare ad implementare l' encoder di BERT preaddestrato per l'italiano e il nostro oracolo LSTM. Innanzitutto andiamo ad inizializzare l' encoder BERT per ricavare gli embeddigs e impostiamo i parametri ottimali per il nostro oracolo LSTM. I risultati migliori li abbiamo ottenuti con la seguente configurazione:

Table 1. parametri oracolo LSTM

input size	4608 + 1200
hidden size	512
n layers*	1
output size	3
epoch	4
learning rate	0.0005

Funzione encode(): Nella funzione *encode()* utilizziamo l' encoder di BERT per andare a generare gli embeddings contestualizzati dato il testo e la lemmatizzazione di quest'ultimo. la frase lemmatizzata ci è utile perché, se BERT non riesce a tokenizzare una parola specifica, allora gli verrà fornito il lemma della suddetta.

Funzione score(): Tramite *score()* invece andiamo a restituire in output un vettore di probabilità contenente le tre mosse eseguibili dal Parser con associate le relative percentuali. Questa funzione rappresenta l'oracolo del programma, il quale prende in input il vettore degli embeddings e del PoS. Quest'ultimo verrà 'espanso' per conferirgli un peso maggiore all'interno della rete neurale, dato che, i vettori di embeddings, sono già molto grandi.

Funzioni per le feature: Le funzioni *extract pos features()* e *extract embedding features()* permettono di estrarre le feature dallo stato del parser in modo da poterle utilizzare sull'oracolo.

2.5 Parser

Questa classe è progettata per costruire la struttura di dipendenze grammaticali di una frase usando le decisioni fornite dall' oracolo. Il parser simula il processo di analisi delle dipendenze, decidendo come le parole nella frase sono collegate sintatticamente.

Funzione transition(): questo metodo aggiorna lo stato del parser eseguendo una delle tre possibili mosse:

- Shift codificato con il numero 0
- Left-Arc codificato con il numero 1
- Right-Arc codificato con il numero 2

Funzione parsing(): il metodo parsing è il cuore del parser. Esegue i seguenti passi:

- Inizializzazione: imposta lo stack, il buffer e le dipendenze iniziali.

- Codifica: Utilizza l'oracolo per ottenere gli embedding delle parole nella frase.
- Ciclo di Parsing: finché ci sono mosse valide:
  - estrae le caratteristiche delle parole e delle posizioni.
  - ottiene i punteggi delle possibili mosse dall'oracolo.
  - seleziona la mossa con il punteggio più alto.
  - esegue la mossa selezionata aggiornando lo stack, il buffer e le dipendenze.

Funzione simulate_parse(): quest' ultima funzione è necessaria per eseguire un' operazione *reverse engineering* sul corpus del training. In sostanza ci permette di ricostruire le mosse per arrivare alle head finali del corpus annotato.

3 Risultati

Per la valutazione utilizziamo il programma eval07.pl, scritto in Pearl. Sono stati usati i file *ouput_real.conllu* e *output_predicted.conllu* per calcolare i vari punteggi di accuratezza. I risultati mostrano che il parser ha una buona precisione nella costruzione delle dipendenze.

```
C:\Users\danie\Desktop\UNITO\PROGETTI_MAGISTRALE\TLN\1_Parser\data>
perl eval07.pl -q -g output_real.conllu -s output_predicted.conllu
  Labeled   attachment score: 9450 / 11381 * 100 = 83.03 %
  Unlabeled attachment score: 9450 / 11381 * 100 = 83.03 %
  Label accuracy score:       11381 / 11381 * 100 = 100.00 %
```

Fig. 1. Risultati del Parser