# Parsing mit neuronalen Netzen: Training und Anwendung

## Parsertraining

Erstellen Sie eine Datei `train-parser.py`, welche das Training implementiert und Ihren Code aus den letzten beiden Aufgaben importiert.

Implementieren Sie analog zur Aufgabe LSTM-Tagger eine Klasse `Vocab`, deren init-Funktion die Trainingsdaten als Argument erhält und die Konstituentenlabels auf fortlaufende Zahlen beginnend mit 1 abbildet. Der Index 0 ist für die Spezialkategorie `No_Constituent` reserviert. Erstellen Sie Dictionaries für die Abbildung von Labels auf Zahlen und umgekehrt. Erzeugen Sie ferner ein Dictionary für die Abbildung von Buchstaben auf Zahlen, wobei die Nummern 0 und 1 für das Paddingsymbol und das UnkChar-Symbol reserviert sind. Buchstaben, welche nur einmal aufgetreten sind, erhalten keinen eigenen Index sondern dienen zum Training des UnkChar-Symbols. Schreiben Sie eine Methode `words_to_idvecs(words)`, welche eine Wortliste erhält und einen gepaddeten Buchstaben-Tensor und eine Liste der Wortlängen liefert. Schreiben Sie außerdem zwei Methoden `num_char_types()` und `num_label_types()`, welche die Zahl der Zeichen bzw. die Zahl der syntaktischen Labels liefern. Schreiben Sie dann noch eine Methode `vocab.store_parameters(filename)`, welche die Abbildungstabellen für Buchstaben und Labels in einer Datei speichert. Erweitern Sie die init-Funktion der Klasse so, dass sie prüft, ob ein String (=Dateiname) als Argument übergeben wurde, und in diesem Fall die Tabellen aus der Datei einliest und damit das Objekt initialisiert.

Schreiben Sie die Hauptfunktion `train`. Diese liest mit Hilfe der Funktionen aus der ersten Parseraufgabe Trainingsdaten und Developmentdaten aus zwei Dateien ein, deren Namen als Kommandozeilenargumente übergeben werden. Sie speichert die Daten jeweils als eine Liste von Paaren, wobei jedes Paar aus einer Wortliste und einer Konstituentenliste besteht. Dann werden ein Vocab-Objekt und ein neuronales Modell erzeugt. Sie trainieren für `num_epochs` Epochen (bspw. `num_epochs=50`) auf den Trainingsdaten, die Sie zu Beginn jeder Epoche mit `random.shuffle` umordnen. `num_epochs` ist ein optionaler Kommandozeilenparameter.

Das neuronale Netzwerk aus der letzten Aufgabe gibt Ihnen einen 2-dimensionalen Tensor zurück, der für jeden Span eine Liste von Scores liefert. An Position 0 des Vektors befinden sich die Bewertungen für den Span (0,1). Dann folgen (0,2),...,(0,n),(1,2),...,(1,n),...,(n-1,n), wobei n die Zahl der Wörter ist. Um das Loss für diesen Score-Tensor zu berechnen, müssen Sie einen Vektor `label_ids` erstellen, der für jeden Span die korrekte Label-ID enthält.

Gehen Sie dabei so vor, dass Sie zunächst einen 1-dimensionalen Tensor der richtigen Länge mit `NO_CONST_LABEL_ID` füllen und in `label_ids` speichern. Dann erzeugen Sie ein Dictionary `span_id`, welches jeden Span (s, e) auf seine Position abbildet, wobei der Span (0, 1) die Position 0 hat. (Hier kann wieder die Methode `triu_indices` nützlich sein.) Dann iterieren Sie über alle Goldstandard-Konstituenten (l, s, e) des aktuellen Parsebaumes und setzen mit dem Befehl `label_ids[span_id[s,e]] = l` das Label des Spans (s, e) auf die Label-ID l.

Als Trainingskriterium verwenden Sie das `CrossEntropyLoss` von PyTorch. Wenden Sie **Gradient Clipping** mit einer Gradient-Norm von 1 an, um sehr große Gradienten zu vermeiden. Nach jeder Epoche berechnen Sie die Zahl der falsch gelabelten Spans in den Development-Daten. Wenn die Zahl der Fehler die bisher kleinste war, speichern Sie das Netzwerk mit der Methode `torch.save`. Außerdem müssen Sie eine Methode `vocab.store_parameters` aufrufen, um die Abbildungstabellen für Buchstaben und Labels zu speichern. Speichern Sie die Tabellen und das Netzwerk in separaten Dateien mit gleichem Basisnamen und unterschiedlichen Dateiendungen (.io bzw. .pth).

Bei einer guten Implementierung kann die Zahl der falsch gelabelten Konstituenten in den Development-Daten im Laufe des Trainings unter 6000 sinken. Planen Sie genug Zeit für das Training ein, da es auch auf der GPU mehrere Stunden dauern kann.

**Aufruf: python train-parser.py train-parses dev-parses parfile**

## Parsing

Nun implementieren Sie den eigentlichen Parser. Ihr Programm liest das gespeicherte neuronale Netzwerk ein und analysiert damit Sätze aus einer Datei. Jede Zeile der Datei enthält einen bereits tokenisierten Satz (keinen Parsebaum!).

Erstellen Sie eine Datei **parser-analyze.py** für das Parsen neuer Sätze, in der Sie die Klasse `Model` aus der zweiten Teilaufgabe importieren und die Klasse `Vocab` aus der dritten Teilaufgabe.

Implementieren Sie eine Funktion **parse**, welche eine Wortliste als Eingabe erhält und erzeugen Sie mit der Methode `words_to_char_id_tensors(words)` aus der Klasse `Vocab` Präfix- und Suffixtensoren. Rufen Sie das neuronale Netz mit den Tensoren auf, um die Scores für alle Spans und Labels zu berechnen. Berechnen Sie für jeden Span das wahrscheinlichste Label und seinen Score. Dann extrahieren Sie mit dem Viterbi-Algorithmus den besten Parsebaum. Der beste Parsebaum maximiert die Summe der Label-Scores. Der Viterbi-Algorithmus liefert eine Liste von Konstituenten-Tripeln. Diese wird mit der entsprechenden Methode aus Aufgabe 1 in einen Parsebaum in Klammernotation umgewandelt und zurückgegeben. Bei der Implementierung des Viterbi-Algorithmus orientieren Sie sich am Pseudocode aus den Vorlesungsfolien. Versuchen Sie, die beste Zerlegung einer Konstituente in Tochterkonstituenten mittels einer einzigen PyTorch-Operation zu berechnen. Dazu speichern Sie die Viterbi-Scores in einem zweidimensionalen Tensor `vitscore[s,e]`.

Schreiben Sie nun das Hauptprogramm. Dieses liest zuerst das Vokabular mit den Abbildungstabellen aus der letzten Aufgabe ein, dann das trainierte Netzwerk, und schließlich die Sätze. Dann werden die Sätze geparst und die Parsebäume auf Stdout ausgegeben. Erstellen Sie zum Testen eine Textdatei mit einem tokenisierten Satz pro Zeile und parsen Sie die Sätze mit ihrem Parser.

**Aufruf: python parser-analyse.py parfile sentences**

Schicken Sie mir bitte den kompletten ausführbaren Code mit den trainierten Parameterdateien, damit ich das Programm direkt testen kann.

### Vorüberlegungen

- Wie sollte die Funktion *parse* vorgehen?

Bitte schicken Sie mir den kompletten Code, der zur Ausführung der beiden Programme notwendig ist, und eine Datei *num-errors.txt* mit der Anzahl der Fehler nach jeder Trainingsepoche.