Buchstabenbasierter Wortart-Tagger

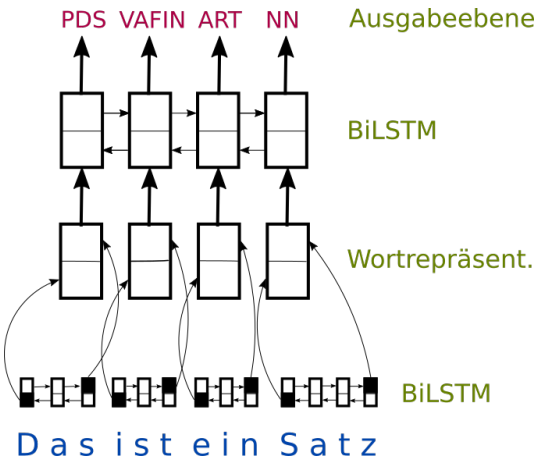
Sie sollen den Tagger aus der letzten Übungsaufgabe verbessern, indem Sie die Wort-Embeddings durch eine Repräsentation ersetzen, die auf Basis der Buchstabenfolge der Wörter berechnet wird. Dazu wird ein weiteres bidirektionales LSTM benutzt, welches über die Buchstabenfolge iteriert. Die beiden Endzustände dieses bidirektionalen LSTMs werden konkateniert und bilden die Repräsentation des Wortes.

Was müssen Sie ändern?

- Die Klasse **Data** erstellt nun eine Indextabelle für Buchstaben statt für Wörter. Buchstaben und Zeichen, die nur einmal auftauchten, erhalten keinen Index.
- Die Methode **words2IDs** wird durch eine Methode **words2IDvecs** ersetzt. Diese Methode iteriert über alle Wörter und berechnet zunächst das Präfix und Suffix der Länge 10 für jedes Wort. (Wenn das Wort kürzer ist, wird es am Ende (Präfix) bzw. Anfang (Suffix) mit Leerzeichen aufgefüllt. Die Buchstaben werden dann mit Hilfe der Indextabelle auf Zahlen abgebildet, wobei unbekannte Zeichen den Index 0 erhalten. Die Funktion gibt eine Matrix mit Präfix-Buchstaben-IDs und eine Matrix mit Suffix-Buchstaben-IDs zurück. Die i-te Zeile enthält jeweils die Buchstaben-IDs des Präfixes/Suffixes des i-ten Wortes.

  Wichtig: Die Präfixe müssen in umgekehrter Reihenfolge verarbeitet werden. Sie sollten daher die Buchstabenreihenfolge vor dem Nachschlagen der IDs umkehren.
- Die Wort-Embedding-Ebene im neuronalen Netz wird durch ein bidirektionales **buchstabenbasiertes LSTM** ersetzt. Die Matrix mit den Buchstaben-Indizes für die Wortsuffixe (analog Wortpräfixe) wird zunächst von einer (Buchstaben-) Embedding-Ebene in einen 3D-Tensor transformiert, der dann mit dem buchstabenbasierten Forward-LSTM für Suffixe (analog Backward-LSTM für Präfixe) verarbeitet wird. Die Buchstabenfolgen aller Wörter werden dabei parallel verarbeitet. Da sich die Eingabe von Forward- und Backward-LSTM unterscheidet, kann hier das BiLSTM nicht mit der *bidirectional*-Option von *nn.LSTM* implementiert werden. Stattdessen werden zwei separate LSTMs benötigt. Die Endzustände dieser beiden buchstabenbasierten LSTMs werden dann für jedes Wort konkateniert und bilden die Eingabe des übergeordneten wortbasierten LSTM.

Netzwerk-Schaubild:



Annotationsprogramm

Sie sollen außerdem ein Programm schreiben, welches mit dem trainierten neuronalen Netz Texte annotiert.

Zunächst müssen Sie dafür sorgen, dass das Modell während des Trainings gespeichert wird. Dazu prüfen Sie nach jeder Evaluierung auf den Entwicklungsdaten, ob das Ergebnis besser als alle vorherigen war, und speichern in diesem Fall. Das Modell selbst speichern Sie mit dem Befehl `torch.save(model, paramFile+".rnn")`. Zusätzlich müssen Sie noch die Indextabellen der Klasse *Data* speichern. Dazu schreiben Sie eine Methode `storeParams(paramFile+".io")`.

Dann schreiben Sie ein Programm **rnn-annotate.py paramFile testFile**, welches das Modell und die Indextabellen einliest und dann Sätze aus der Argumentdatei annotiert und auf stdout ausgibt. Das Format der Eingabedatei entspricht dem der Trainingsdaten. Die Spalte mit den Tags kann hier jedoch fehlen.

Damit Sie die Klasse *Data* einfach im Annotations-Programm wiederverwenden können, definieren Sie den Konstruktor wie folgt:

```
def __init__( self, *args ):
   if len(args) == 1:
      # Aufruf aus rnn-annotate.py
      self.init_test( *args )
   else:
      # Aufruf aus rnn-train.py
      self.init_train( *args )
```

und verwenden ihn so: `data = Data(paramFile+'.io)`. Der Konstruktor liest in diesem Fall nur die Tabellen ein, damit die Methoden *words2IDvecs* und *IDs2tags* verfügbar sind.

Abgabe: Ihr Programmcode und Parameterdateien mit einem trainierten Modell