Training eines LSTM-Wortart-Taggers (Teil 1)

Grundlagen: Wortart-Annotation, rekurrente neuronale Netze, LSTMs, PyTorch

In den nächsten Übungen werden Sie einen Wortart-Tagger auf Basis von LSTMs trainieren. Trainingsdaten und Entwicklungsdaten finden Sie hier:

`http://www.cis.uni-muenchen.de/~schmid/lehre/data/Tiger-morph.zip`

Jede Zeile enthält ein Wort und ein Tag, die von einem Tabulatorzeichen getrennt sind. Am Ende jedes Satzes befindet sich eine Leerzeile.

Der Tagger bildet die Wörter erst auf Indizes und dann auf **Embeddings** ab. Die Folge der Embeddings wird mit einem **bidirektionalen LSTM** verarbeitet. Die Zustände des Forward- und Backward-LSTMs werden an jeder Position zu einer Kontext-abhängigen Repräsentation des entsprechenden Wortes konkateniert. Auf das BiLSTM folgt ein **Feedforward**-Netzwerk mit einer Hidden Layer. Die Ausgabeebene enthält für jedes Tag ein Neuron.

Schritte:

Schreiben Sie für die Datenvorverarbeitung eine Klasse **Data** in einer Datei `Data.py`, deren Konstruktor

- zwei Dateien mit Trainingsdaten und Entwicklungsdaten einliest und speichert,
- die Liste der Wörter aus den Trainingsdaten extrahiert, nach Häufigkeit sortiert und den n häufigsten Wörtern einen fortlaufenden Index (1,2,...,n) zuweist
- die Liste der Tags aus den Trainingsdaten extrahiert und allen Tags ebenfalls Indizes (0,1,2,...) zuweist.

Bei der Wort-Indextabelle ist der Index 0 für unbekannte Wörter reserviert.

Die Klasse besitzt außerdem

- eine Methode **words2IDs** zur Abbildung einer Folge von Wörtern auf eine Folge von Indizes
  (Wörter, denen kein Index explizit zugewiesen wurde, erhalten den Index 0.)
- eine Methode **tags2IDs** zur Abbildung einer Folge von Tags auf eine Folge von Indizes (Unbekannte Tags erhalten den Index -1.)
- eine Methode **IDs2tags** zur umgekehrten Abbildung einer Folge von Tag-Indizes auf eine Folge von Tagstrings
- ein Attribut **numTags**, welches die Zahl der Tags liefert.
- ein Attribut **trainSentences**, welches die Trainingsdaten speichert.
- ein Attribut **devSentences**, welches die Entwicklungsdaten speichert.

Die Klasse **Data** wird später folgendermaßen benutzt werden:

```
data = Data(trainfile, devfile, numWords) # Objekt der Klasse erzeugen

for words, tags in data.trainSentences:   # über Trainingssätze iterieren
   wordIDs = data.words2IDs(words)        # Wörter auf Zahlen abbilden
   tagIDs  = data.tag2IDs(tags)           # Tags auf Zahlen abbilden

for words, tags in data.devSentences:     # über Entwicklungsdaten iterieren
   wordIDs    = data.words2IDs(words)     # Wörter auf Zahlen abbilden
   bestTagIDs = annotate(wordIDs)         # beste Tagfolge berechnen
   bestTags   = data.IDs2tags(bestTagIDs) # Tagindizes auf Tagstrings abbilden
```

(Die Methode `annotate` werden Sie erst später implementieren.)

Als Nächstes schreiben Sie eine Klasse **TaggerModel** in einer Datei `TaggerModel.py`, welche das neuronale Netz für den Tagger mit Hilfe der PyTorch-Klassen *nn.Embedding*, *nn.LSTM*, *nn.Linear* und *nn.Dropout* implementiert. Bei der Erzeugung der Embeddingtabelle müssen Sie berücksichtigen, dass es ein zusätzliches unknown-Wort mit Index 0 gibt. Dropout sollte auf die Embeddings der Wörter und auf ihre kontextuellen Repräsentationen angewendet werden, um Overfitting zu vermeiden. Schauen Sie sich die Dokumentation der Module genau an. Jedes Modul besitzt eine Aufrufschnittstelle für den Konstruktor und eine Schnittstelle für die Anwendung des Modules. Das bidirektionale LSTM können Sie einfach mit der Option *bidirectional* des LSTM-Konstruktors erzeugen.

TaggerModel besitzt neben dem Konstruktor eine Methode **forward**, welche die Verarbeitung implementiert. Die Ausgabevektoren eines bidirektionalen LSTMs haben doppelte Länge. Sie müssen keine Batchverarbeitung implementieren. Die **forward**-Methode berechnet keinen Softmax, weil Sie zum Training das CrossEntropyLoss verwenden werden, welches den SoftMax selbst berechnet.

Verwendung:

```
tagger = TaggerModel(data.numTags, numWords, embSize, lstmSize, hiddenSize,
                     dropoutRate)
tagLogits = tagger(torch.LongTensor(wordIDs))
```

Sie können folgende Werte verwenden:

numWords=10000	Vokabulargröße
embSize=200	Länge der Embeddings
lstmSize=400	Zahl der Neuronen im LSTM
hiddenSize=200	Zahl der Neuronen in der Hidden Layer
dropoutRate=0.3	Wahrscheinlichkeit, mit der Neuronen-Aktivierungen im Training auf 0 gesetzt werden

Schreiben Sie für jedes der beiden Module eine Testfunktion **run_test**, welche die Funktionsweise überprüft und eine Warnung ausgibt, falls ein Fehler entdeckt wird. Mit dem Befehl

```
if __name__ == "__main__":
```

```
run_test()
```

am Ende der Moduldatei sorgen Sie dafür, dass die Testfunktion nur dann ausgeführt wird, wenn das Modul als Programm ausgeführt wird. Die Testfunktion des Moduls TaggerModel soll nur testen, ob die init-Funktion und die forward-Funktion ohne Fehlermeldungen durchlaufen. Verwenden Sie dazu zufällig erzeugte Eingabedaten mit den richtigen Dimensionen und Wertebereichen.

Vorüberlegungen:

- Welche weiteren Datenstrukturen sind in der Klasse *Data* sinnvoll?
- Zeichnen Sie den Aufbau des neuronalen Netzes.