LSTM-Wortart-Tagger (Teil 2)

In dieser Übung werden Sie den Wortart-Tagger fertigstellen.

Schreiben Sie ein Programm `tagger-train.py`, welches das Training implementiert. Es erzeugt ein Objekt der Klasse `Data`, um die Daten einzulesen, und ein Objekt der Klasse `TaggerModell`. Dann iteriert es `numEpochs`-mal über die Trainingsdaten und trainiert nacheinander auf jedem Trainingssatz. Sie sollten die Trainingsdaten vor jeder neuen Epoche zufällig umordnen mit dem Befehl: `random.shuffle(data.train_sentences)`

Nach jeder Epoche wird über alle Entwicklungsdaten iteriert, um die Tagging-Genauigkeit zu berechnen und auszugeben. Da Sie Dropout verwenden, müssen Sie mit Hilfe der Methoden `model.train()` und `model.eval()` zwischen Trainings- und Evaluierungsmodus umschalten. Wenn die aktuelle Genauigkeit höher als alle bisher erzielten Genauigkeiten ist, wird das Modell mit dem Befehl

```
torch.save(model, parfile)
```

gespeichert. Dabei ist `parfile` ein bereits geöffnetes Dateiobjekt.

Fügen Sie außerdem eine Methode `save(self, parfile)` zum Modul `Data.py` hinzu, welche mit pickle die Tabellen speichert, die für die Abbildung zwischen Wörtern und Zahlen benötigt werden. Dann benennen Sie die alte Methode `__init__` in `init_train` um, schreiben eine neue Methode `init_test`, welche die Index-Tabellen aus einem geöffneten Dateiobjekt einliest und schließlich noch eine neue Methode `__init__(self, *args)`, welche `init_test` aufruft, falls args nur ein Argument enthält und sonst `init_train`.

Programmaufruf:

```
python tagger-train.py trainfile.txt devfile.txt paramfile --num_epochs=20
       --num_words=10000 --emb_size=200 --lstm_size=400 --hidden_size=400
       --dropout_rate=0.3 --learning_rate=0.0001
```

Die Kommandozeilenargumente sollten Sie mit dem Modul `argparse` verarbeiten. `hidden_size` ist die Größe der Hidden-Ebene des Ausgabenetzwerkes.

Wenn Ihr Rechner eine gute Grafikkarte besitzt, können Sie diese für das Training nutzen, indem Sie mit dem Befehl `model = model.to(device)` das Modell auf GPU verschieben. Die globale Variable `device` definieren Sie einmal zu Beginn mit dem Befehl

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

Schließlich müssen Sie noch ein Programm `tagger-annotate.py` schreiben, welches die Namen der Parameterdatei und der Eingabedatei als Argumente nimmt und die annotierten Sätze auf dem Bildschirm ausgibt. Mit den Befehlen

```
data  = Data(parfile+".io")
model = torch.load(parfile+".pt")
```

lesen Sie die Abbildungstabellen und das neuronale Netzwerk ein. Mit der Methode `sentences` der Klasse `Data` lesen Sie die Sätze einzeln ein und geben die Wörter und ihre wahrscheinlichsten Tags im gleichen Format wie die Trainingsdaten aus.

Zum Modul `Data.py` fügen Sie eine neue Methode hinzu:

`sentences(filename)` ist ein Generator, der eine Datei einliest und mit dem Befehl `yield` nach dem Einlesen eines Satzes die Liste der Wörter zurückgibt. Jede Zeile der Datei enthält einen bereits tokenisierten Satz.

mögliche Parameterwerte:

Vokabulargröße: 10000-50000

Embeddings-Größe: 100-300

LSTM-Größe: 200-500

Hidden-Größe: 200-500

Optimierer: Adam mit Lernrate circa 0.0001 oder SGD mit Lernrate 0.5.

Loss-Funktion: CrossEntropyLoss

Bitte geben Sie Ihren Code und eine Datei mit Angaben zur erzielten Genauigkeit auf den Development-Daten ab.

Vorüberlegungen:

- Welche Schritte müssen in den Programmen `tagger-train.py` und `tagger-annotate.py` ausgeführt werden?

Debugging:

Wenn Sie die Ursache eines PyTorch-Fehlers nicht finden, können Sie so vorgehen:

- Prüfen Sie die Fehlermeldung. Oft weist ein Eingabetensor falsche Dimensionen oder einen falschen Datentyp auf. Oder die Argumente befinden sich nicht alle auf der GPU. PyTorch sagt Ihnen dies dann. Wenn sich die Fehlermeldung auf einen Befehl in einer PyTorch-Bibliothek bezieht, müssen Sie nachschauen, welcher Befehl Ihres Codes zuletzt ausgeführt wurde. Prüfen Sie dann die Argumente dieses Befehles.
- Wenn das Programm einfach abstürzt, dann müssen Sie zuerst herausfinden, an welcher Stelle Ihres Programms das Problem auftaucht (z.B. durch Kontrollausgaben).
- Bei Problemen mit Indexoperationen auf Tensoren sollten Sie prüfen, ob der Index kleiner als die Tensorgröße ist. Dasselbe gilt für Embedding-Operationen. Ein häufiger Fehler besteht darin, die Embeddingtabelle zu klein zu wählen, weil nicht an das unknown-Token gedacht wird. Ungültige Indizes können vor allem in Verbindung mit der GPU zu großen Debuggingproblemen führen.
- Wenn Sie Fehlermeldungen von Cuda oder CuDNN bekommen, hilft Ihnen vielleicht auch ein Test auf CPU weiter.