LSTM-Wortart-Tagger

In dieser Woche sollen Sie einen Wortart-Tagger auf Basis von LSTMs implementieren. Die Trainingsdaten und Entwicklungsdaten laden Sie von folgender URL herunter:
`http://www.cis.uni-muenchen.de/~schmid/lehre/data/Tiger-morph.zip`

Jede Zeile der Datei enthält ein Wort und ein Tag mit einem Tabulator als Trennzeichen. Am Ende jedes Satzes befindet sich eine Leerzeile.

Der Tagger verarbeitet die Buchstabenfolgen der Wörter mit einem bidirektionalen LSTM und konkateniert die Endzustände des Forward-LSTMs und des Backward-LSTMs zu einer Wortrepräsentation. Die Folge der Wortrepräsentationen wird von einem weiteren bidirektionalen LSTM verarbeitet. Der Ausgabe-Tensor dieses zweiten LSTMs wird mit einer Feedforward-Ebene verarbeitet. Darauf folgt die Ausgabeebene, welche für jedes Wort und jedes mögliche Tag einen Logit-Wert liefert.

Modul Data

Schreiben Sie für die Datenvorverarbeitung eine Klasse **Data** in einer Datei `data.py`, mit einer Methode **init_train(self, train_data_file, dev_data_file)**, welche

- zwei Dateien mit Trainingsdaten und Entwicklungsdaten einliest und speichert,
- die Liste der Tags aus den Trainingsdaten extrahiert und allen Tags Indizes (0, 1, 2, ...) mittels einer Tabelle (Dictionary) zuweist.
- die Liste der Buchstaben und sonstigen Eingabezeichen mit Häufigkeiten aus den Trainingsdaten extrahiert und jedem Zeichen mit einer Häufigkeit größer als 1 einen fortlaufenden Index (2, 3, ...) zuweist. Die Indices 0 und 1 sind für das Paddingsymbol und das unbekannte Zeichen reserviert.

Die Klasse Data besitzt außerdem

- eine Methode **tags_to_ids(self, tags)** zur Abbildung einer Folge von Tags auf eine Folge von Indizes, wobei unbekannte Tags den Index -1 erhalten.
- eine Methode **ids_to_tags** zur umgekehrten Abbildung einer Folge von Tag-Indizes auf eine Folge von Tagstrings (Das unbekannte Tag mit Index -1 muss nicht abgebildet werden können.)
- eine Methode **words_to_idvecs(self, words)**, welche eine Wortliste erhält und
  - die Buchstaben der Wörter auf Indizes abbildet, wobei unbekannten Zeichen der Index 1 zugewiesen wird,
  - diese Zahlenfolgen mit dem PyTorch-Befehl `pad_sequence` mit Padding-Symbolen auffüllt und den zweidimensionalen Ergebnis-Tensor und die Liste der Wortlängen zurückgibt.

- ein Attribut **num_tags**, welches die Zahl der Tags liefert.
- ein Attribut **train_sentences**, welches die Trainingsdaten speichert.
- ein Attribut **dev_sentences**, welches die Entwicklungsdaten speichert.
- eine Methode `save(self, parfilename)`, welche mit pickle die Tabellen speichert, die für die Abbildung zwischen Wörtern bzw. Tags und Zahlen benötigt werden.
- eine Methode `init_test`, welche mit pickle diese Tabellen wieder einliest.
- eine Methode `__init__(self, *args)`, welche `init_test` aufruft, falls `args` nur ein Argument enthält und sonst `init_train`.
- eine Generator-Methode `sentences(file)`, welche eine bereits geöffnete Datei einliest, welche in jeder Zeile einen bereits tokenisierten Satz enthält. Die Methode gibt die entsprechende Wortfolge mit dem Befehl `yield` zurück.

Die Klasse **Data** soll folgendermaßen benutzt werden:

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

# Training
for words, tags in data.train_sentences:   # über Trainingssätze iterieren
   result = data.words_to_idvecs(words)    # Zeichen auf Zahlen abbilden
   char_tensor, length_vector = result
   tag_ids = data.tag_to_ids(tags)         # Tags auf Zahlen abbilden

# Evaluation auf Developmentdaten
for words, tags in data.dev_sentences:     # über Entwicklungsdaten iterieren
   result = data.words_to_idvecs(words)    # Zeichen auf Zahlen abbilden
   char_tensor, length_vector = result
   best_tag_ids = annotate(char_tensor, length_vector) # beste Tagfolge berechnen
   best_tags = data.ids_to_tags(best_tag_ids)  # Indizes auf Tags abbilden

# Annotation von Testdaten
data = Data(paramfile)
for words in data.sentences():             # über Entwicklungsdaten iterieren
   result = data.words_to_idvecs(words)    # Zeichen auf Zahlen abbilden
   char_tensor, length_vector = result
   best_tag_ids = annotate(char_tensor, length_vector) # beste Tagfolge berechnen
   best_tags = data.ids_to_tags(best_tag_ids)  # Indizes auf Tags abbilden
```

Modul Model

Nun schreiben Sie eine Klasse **Model** in einer Datei `model.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 die Padding- und Unknown-Symbole berücksichtigen. Sie sollten auf die Embeddings und alle Zwischenrepräsentationen Dropout anwenden, um Overfitting zu vermeiden. Schauen Sie

sich die Dokumentation der Module genau an. Jedes Modul besitzt zwei Schnittstellen, eine für den Konstruktor und eine für die Anwendung des Modules.

Bidirektionale LSTM erhalten Sie mit der Option *bidirectional* des LSTM-Konstruktors. Ihre Ausgabevektoren haben doppelte Länge. Mit der Option *num_layers* erzeugen Sie ein LSTM mit mehreren Ebenen. Bei dem LSTM, welches die Wortrepräsentationen verarbeitet, sollten Sie zwei Ebenen verwenden und die Dropout-Option des LSTMs verwenden, damit auch auf die Zwischenrepräsentationen des LSTMs Dropout angewendet wird. Die Eingabe des ersten BiLSTMs ist dreidimensional. Die erste Dimension ist die Batch-Dimension. Sie brauchen daher die `batch_first`-Option des LSTMs. Die Eingabe des zweiten BiLSTMs ist zweidimensional. Sie brauchen hier keine Batch-Dimension hinzuzufügen.

Die Klasse **Model** besitzt zwei Schnittstellen, eine für den Konstruktor (Methode __init__) und eine für die Anwendung des Netzwerkes (Methode forward). Die **forward**-Methode berechnet keinen Softmax, weil Sie zum Training das CrossEntropyLoss verwenden werden, welches den SoftMax selbst berechnet.

Zur Steigerung der Effizienz sollten Sie in der `forward`-Methode die Eingabe des ersten LSTMs mit der PyTorch-Methode `pack_padded_sequence` komprimieren und die Ausgabe mit `pad_packed_sequence` wieder in einen normalen Tensor verwandeln.

Zur Extraktion der Endzustände sollten Sie den Ausgabe-Tensor des ersten LSTMs in der letzten Dimension zunächst mit der PyTorch-Methode `chunk` in zwei gleich große Tensoren mit den Ausgaben der Forward- und Backward-Richtung aufspalten. Beim Backward-Tensor finden Sie die Endzustände an Buchstabenposition 0. Beim Forward-Tensor variiert die Position je nach Wortlänge. Hier sollten Sie Advanced Indexing verwenden, um alle Endzustände mit einem PyTorch-Befehl zu extrahieren.

Verwendung:

```
model = Model(data.num_chars, data.num_tags, emb_size, char_lstm_size,
              word_lstm_size, hidden_size, dropout_rate)
tag_logits = model(char_id_tensor, word_lengths)
```

Trainingsprogramm

Schreiben Sie nun 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 `Model`. Dann iteriert es `num_epochs`-mal über die Trainingsdaten und trainiert nacheinander auf jedem Trainingssatz (ohne Parallelverarbeitung der Sätze). Vor jeder neuen Epoche ordnen Sie die Trainingsdaten zufällig um mit dem Befehl: `random.shuffle(data.train_sentences)`

Verwenden Sie Gradient Clipping mit der Methode `torch.nn.utils.clip_grad_norm_` und `max_norm=1`, um übergroße Gradienten zu verhindern, die das Training instabil machen würden.

Verwenden Sie außerdem den Lernraten-Scheduler `CosineAnnealingLR`, um im Laufe des Trainings die Lernrate auf ein Viertel zu reduzieren.

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 von `torch.nn.Module` geerbten 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, parfilename)
```

in einer Datei mit dem Namen `parfilename` gespeichert. (Sie können hier auch nur das `state_dict` des Modelles speichern. Dann sollten Sie aber auch die Netzwerkgrößen speichern, damit Sie das Modell später initialisieren können.)

Programmaufruf:

```
python tagger-train.py trainfile.txt devfile.txt paramfile
       --emb_size=200 --char_lstm_size=400 --word_lstm_size=400
       --hidden_layer_size=400 --dropout_rate=0.3
       --num_epochs=20 --learning_rate=0.0001
```

Die Kommandozeilenargumente sollten Sie mit dem Modul `argparse` verarbeiten. Alle Argumente, die mit '–' beginnen, sollten optionale Argumente mit Defaultwerten sein.

Ihr Programm soll in der Lage sein, eine Grafikkarte (sofern vorhanden) zu nutzen. Dazu verschieben Sie ggf. mit dem Befehl `model = model.to(device)` das Modell auf GPU. Die globale Variable `device` definieren Sie einmal zu Beginn mit dem Befehl

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

Anwendungsprogramm

Schließlich müssen Sie noch ein Programm `python tagger.py parfile inputfile` schreiben, welches die Namen der Parameterdatei und der Eingabedatei als Argumente nimmt. Mit den Befehlen

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

lesen Sie die Abbildungstabellen und das neuronale Netzwerk ein. (Wenn sich das Modell beim Abspeichern auf GPU befand, wird es auch hier wieder direkt zur GPU verschoben. Das können Sie mit der Option `map_location` des `load`-Befehles ändern.) 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 auf dem Bildschirm aus.

mögliche Parameterwerte für das Training:

Embeddings-Größe: 100-300
Char-LSTM-Größe: 100-300
Word-LSTM-Größe: 200-400

Hidden-Größe: 200-400
Optimierer: SGD mit Lernrate 0.1-1.0 oder Adam mit Lernrate 0.0001-0.001.
Loss-Funktion: CrossEntropyLoss

Bitte geben Sie Ihren Code und eine Datei mit den im Training ausgegebenen Development-Genauigkeiten ab.

Vorüberlegungen:

- Welche Datenstrukturen brauchen Sie in der Klasse *Data*?
- Zeichnen Sie den Aufbau des neuronalen Netzes.
- Welche Schritte müssen in den Programmen `tagger-train.py` und `tagger.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.