Lemmatisierung mit einem Encoder-Decoder-System: Training und Anwendung

In dieser Woche implementieren Sie die Trainings- und Anwendungsprogramme.

Lemmatisierung

Im Training und bei der Evaluierung auf Development-Daten ist die Folge der Ausgabesymbole bekannt. Daher kann jede Decoder-Ebene die Eingabe komplett auf einmal abarbeiten. Dies ist nicht möglich, wenn das Modell tatsächlich Wörter lemmatisieren soll, weil hier die Ausgabesymbole unbekannt sind.

Sie fügen daher zum Modul `Model` eine weitere Methode `lemmatize(src_letter_ids, src_lengths, max_tgt_length)` hinzu, welche die Ausgabesymbole einzeln generiert. Die Methode ruft den Encoder auf und initialisiert dann einen Tensor `tgt_char_ids` der Dimension Bx1 mit `pad_id`, wobei `B` die aktuelle Batchgröße ist.

Dann wird von 1 bis max_tgt_length iteriert. In jeder Iteration werden zunächst die Embeddings von `tgt_char_ids` nachgeschlagen. Dann wird LSTM1 mit den Embeddings aufgerufen:

```
dec_states, previous_states1 = self.lstm1(embs, previous_states1)
```

In der ersten Iteration enthält `previous_states1` die Encoder-Endzustände, in späteren Iterationen die Zustände aus der vorherigen Iteration. Als Nächstes werden die Kontextvektoren für `dec_states` berechnet und mit `dec_states` konkateniert. Das Ergebnis wird mit LSTM2 verarbeitet, das analog zu LSTM1 aufgerufen wird. Die dritte LSTM-Ebene gleicht der zweiten.

Zum Schluss werden noch die Projektionsebene und die Ausgabeebene angewendet. Eine argmax-Operation liefert die Ausgabesymbole `tgt_char_ids` der aktuellen Iteration. Die Ausgabesymbole werden über alle Iterationen hinweg im Tensor `all_tgt_char_ids` aufgesammelt. Die Iteration bricht ab, wenn in jeder Ausgabezeile ein Padding-Symbol enthalten ist. Dazu kann folgender Test verwendet werden:

```
torch.all(torch.any(all_tgt_char_ids == pad_id, dim=1), dim=0)
```

Der Tensor `all_tgt_char_ids` wird zurückgegeben.

Fügen Sie überall noch Dropout-Ebenen ein außer nach der Projektionsebene.

Training

Schreiben Sie ein Programm `lemmatizer-train.py`, welches den Lemmatisierer trainiert. Als Loss-Funktion verwenden Sie `CrossEntropyLoss`, als Optimizer `AdamW`.

Verwenden Sie `argparse.ArgumentParser`, um folgende Kommandzeilen-Argumente einzulesen: `trainfile`, `devfile`, `paramfile`, `embeddings_size=100`, `lstm_size=400`,

`num_epochs=50`, `batch_size=1000`, `dropout_rate=0.5`. Definieren Sie bei den optionalen Argumenten – das sind alle außer den ersten drei Argumenten – die oben angegebenen Defaultwerte. Versuchen Sie auf einer Grafikkarte zu arbeiten und verringern Sie die Batchgröße, wenn der Speicherplatz nicht reicht.

Erzeugen Sie ein `Data`-Modul. Speichern Sie es mit der Methode `save` in der Datei `args.paramfile + '.io'`, wobei `args.paramfile` ein Kommandozeilen-Argument ist.

Trainieren Sie den Lemmatisierer auf den Trainingsdaten und evaluieren Sie ihn nach jeder Epoche auf den Development-Daten. Verwenden Sie bei der Evaluierung die Methode `lemmatize`. Geben Sie die Genauigkeit (Anteil der korrekt lemmatisierten Wörter) aus und speichern Sie das aktuelle Modell in der Datei `args.paramfile + '.pth'`, falls die Genauigkeit höher als alle bisherigen Genauigkeiten ist.

Anwendung

Schreiben Sie ein Programm `lemmatizer.py`, welches den Lemmatisierer auf Eingabedaten anwendet. Das Programm erhält zwei obligatorische Kommandozeilen-Argumente `paramfile` und `inputfile` und das optionale Argument `batch_size`. Der Inhalt der Datei `inputfile` ähnelt `train.txt`, enthält aber keine Lemmas. Die Ausgabe des Programmes hat dasselbe Format wie die Trainingsdaten und geht auf den Bildschirm.

Geben Sie Folgendes ab:

- die Liste der Genauigkeiten nach den einzelnen Epochen
- die Ausgabe Ihres Lemmatisierers für die ersten 100 Wörter der Developmentdaten (Die Eingabedatei des Lemmatisierers können Sie mit folgendem Befehl erstellen: `head -100 dev.txt | cut -f1 > input.txt`)
- Ihren vollständigen Code inklusive dem (eventuell verbesserten) Code aus den letzten beiden Aufgaben