Lemmatisierung mit einem Encoder-Decoder-System: Modell

In dieser Woche implementieren Sie das neuronale Netzwerk des Lemmatisierers in der Datei `model.py`. Das Programm umfasst die drei Klassen `Encoder`, `Attention` und `Model`.

Encoder

Die Klasse `Encoder(vocab_size, embedding_size, lstm_size, num_layers, dropout_rate)` verarbeitet die Buchstabenfolgen der Wörter mit einem tiefen bidirektionalen LSTM mit 2 oder mehr Ebenen und erzeugt eine Repräsentation für jede Buchstabenposition. Die `forward`-Funktion erhält als Eingabe einen 2-dimensionalen Tensor mit einem Batch von gepaddeten Buchstaben-ID-Folgen sowie eine Liste mit den Eingabelängen. Nach dem Embedding-Lookup wird der Tensor mit der PyTorch-Funktion `pack_padded_sequence` in eine kompakte, effizienter verarbeitbare Repräsentation transformiert, die von dem BiLSTM verarbeitet wird. Die Ausgabe des BiLSTMs wird mit der Funktion `pad_packed_sequence` wieder in einen gepaddeten Tensor `output` zurücktransformiert, der zurückgegeben wird.

Die Endzustände der Rückwärts-Richtung der letzten Ebene des LSTMs werden später zur Initialisierung der Zustände des Decoder-LSTMs verwendet. Daher müssen zusätzliche Tensoren zurückgegeben werden. Dazu speichern Sie den zweiten Rückgabewert des BiLSTMs in den Variablen `(hidden, cell)`

Das Modul gibt die Tensoren `output`, `hidden[-1:]` und `cell[-1:]` zurück. (Mit dem Operator [-1:] werden die Rückwärts-Zustände der letzten Ebene extrahiert.)

Attention

Die Klasse `Attention(enc_size, dec_size, dropout_rate)` implementiert das Attention-Modul und umfasst zwei Feedforward-Ebenen des Typs `Linear`. Die erste FF-Ebene erhält die Konkatenation eines Encoder-Zustands und eines Decoder-Zustands als Eingabe und liefert eine Ausgabe in der Größe eines Decoder-Zustandes. Die zweite FF-Ebene erzeugt einen Ausgabevektor der Länge 1 mit einem Logit-Wert.

Das Attention-Modul wird mit den Argumenten `(encoder_states, decoder_states)` aufgerufen. Zu dem dreidimensionalen Query-Tensor `decoder_states` wird zunächst eine zusätzliche Dimension 2 hinzugefügt, die mit dem `expand`-Befehl genausolang wie die Dimension 1 des Key/Value-Tensors `encoder_states` gemacht wird. Anschließend wird zum Tensor `encoder_states` eine zusätzliche Dimension 1 hinzugefügt und so lange wie die Dimension 1 des Tensors `decoder_states` gemacht. Die Tensoren `encoder_states` und `decoder_states` werden dann in der letzten Dimension konkateniert. Der neue Tensor wird durch die erste Feedforward-Ebene transformiert. Auf das Ergebnis wird eine tanh-Aktivierungsfunktion angewendet. Der Ergebnis-Tensor wird durch die zweite Feedforward-Ebene transformiert. Dann wird `softmax` auf die Encoder-Dimension des Tensors angewendet. Der Ergebnistensor mit den Attention-Wahrscheinlichkeiten wird punktweise mit `encoder_states` multipliziert

und über die Encoder-Dimension aufsummiert, um den Kontext-Tensor zu erhalten, der zurückgegeben wird.

Fassen Sie die beiden Feedforward-Ebenen, die tanh-Funktion, den SoftMax und Dropout zu einem PyTorch-Sequential-Modul zusammen.

Model

Die Hauptklasse `Model(src_vocab_size, tgt_vocab_size, embedding_size, lstm_size, num_layers, dropout_rate, pad_id)` umfasst ein Encoder-Modul und ein Attention-Modul und implementiert den Decoder. Zum Decoder gehören ein Embedding-Modul, zwei oder mehr separate, unidirektionale Batch-First-LSTMs, eine Projektions-Ebene und eine Ausgabe-Ebene. Die Werte der Parameter `embedding_size` und `lstm_size` sind bei Encoder und Decoder identisch. Speichern Sie die LSTMs in einer `ModuleList` mit dem Namen `self.lstm`, damit Sie später darüber iterieren können.

Sie verwenden in der Decoder-Embedding-Ebene `self.embedding` und der Ausgabe-Ebene `self.output_layer` dieselben Embedding-Vektoren ("Tied Embeddings"). Daher benötigen Sie die Projektions-Ebene `self.output_projection`, welche die LSTM-Zustands-Vektoren auf die Embeddinglänge projizieren. Mit dem Befehl `self.output_layer.weight = self.embedding.weight` ersetzen Sie bei der Initialisierung des Modules `Model` die Embeddings der Ausgabe-Ebene durch die Eingabe-Embeddings.

Die forward-Methode des Moduls `Model` erhält die Argumente `(src_letter_ids, src_lengths, tgt_letter_ids)`. Sie ruft zunächst den Encoder auf, welcher die Encoder-Zustände und die Rückwärts-Endzustände zurückgibt. Dann werden Embeddings für `tgt_letter_ids` nachgeschlagen. Das erste LSTM verarbeitet die Embeddings, wobei sie die Rückwärts-Endzustände als Startzustände erhält.

Dann wird dass Attention-Modul aufgerufen, welches den Kontext-Tensor liefert. Der Kontext-Tensor wird mit der Ausgabe des 1. LSTMs in der letzten Dimension konkateniert und vom 2. LSTM verarbeitet, welches ebenfalls die Rückwärts-Endzustände als Startzustände erhält. Dieselben Schritte werden mit allen weiteren LSTMs wiederholt.

Die Ausgabe des letzten LSTMs wird erst durch die Projektions-Ebene und dann durch die Ausgabe-Ebene verarbeitet, ohne Aktivierungsfunktionen anzuwenden. Der erhaltene Tensor mit Logitwerten wird zurückgegeben.

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 schreiben daher eine weitere Methode `lemmatize(src_letter_ids, src_lengths, max_tgt_length)`, 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.

Schreiben Sie zum Schluss noch Code, um das Modell zu testen. Der Code sollte eine Instanz von `Model` erzeugen und dann mit einem passenden Tensor aufrufen. Prüfen Sie, ob die Dimensionen des zurückgegeben Vektors stimmen. Die Methode `lemmatize` müssen Sie nicht testen.