Lemmatisierung mit einem Encoder-Decoder-Modell: Vorverarbeitung und Neuronales Netz

In den kommenden zwei Wochen werden Sie ein Encoder-Decoder-Modell mit Attention implementieren und für die Lemmatisierung von Wörtern verwenden. In der aktuellen Aufgabe implementieren Sie die Vorverarbeitung und das neuronale Netz.

Datenvorverarbeitung

Schreiben Sie eine Klasse `Data` in einer Datei `data.py` für die Vorverarbeitung der Lemmatisierungsdaten. Die Klasse liest die Daten ein, wandelt Buchstaben in Zahlen um und fasst Beispiele zu Batches zusammen. Die Klasse besitzt folgende Methoden:

- `read_data(filename)`: Diese Methode liest eine Datei ein, in der jede Zeile ein Wort und ein Lemma enthält, die durch ein Tabulatorzeichen getrennt sind. Die Liste der Paare wird zurückgegeben.
- `make_table(words)`: Diese Methode zählt die Buchstaben in der übergebenen Liste von Wörtern, extrahiert alle Buchstaben, die mindestens zweimal aufgetreten sind, und weist ihnen fortlaufende Nummern ab 2 zu. Die Nummern 0 und 1 werden dem Paddingsymbol und dem unbekannten Zeichen zugewiesen. Die Methode gibt ein Dictionary zurück, welches die Symbole auf ihre Nummern abbildet sowie den maximalen Index plus 1.
- `init_train(self, traindata, devdata)`: Diese Methode liest die beiden Argumentdateien mit der Methode `read_data` in `self.train_data` und `self.dev_data` ein. Sie extrahiert mit einem zip-Befehl die Wörter und Lemmata aus den Trainingsdaten. Sie ruft die Methode `make_table` mit der Liste der Wörter auf und speichert die beiden Rückgabewerte in `self.src_char_to_id` und `self.num_src_chars`. Analog werden mit der Liste der Lemmata `self.tgt_char_to_id` und `self.num_tgt_chars` erzeugt. Zusätzlich wird ein Dictionary `self.id_to_tgt_char` erzeugt, welches die Nummern wieder auf die Lemma-Symbole abbildet. Dann speichern Sie noch die Nummer des Paddingsymbols in `self.pad_id` und die Nummer des unbekannten Zeichens in `self.unk_id`.

  Zum Schluss iteriert die Methode über alle Wort-Lemma-Paare in den Trainingsdaten, berechnet das maximale Verhältnis der Lemmalänge zur Wortlänge und speichert es in `self.max_len_factor`.
- `save(self, paramfile)`: Diese Methode speichert (mindestens) die Attribute `self.src_char_to_id`, `self.id_to_tgt_char`, `self.pad_id`, `self.unk_id` und `self.max_len_factor` mit Pickle in der Argumentdatei.
- `init_test(self, filename)`: Diese Datei liest die mit `save` gespeicherten Parameter wieder aus einer Datei ein.
- `__init__(self, *args)`: Die Konstruktormethode ruft `init_test(*args)` auf, falls die Länge von `args` gleich 1 ist, und anderenfalls `init_train(*args)`.

- `train_batches(self, max_batch_size)`: Diese Methode ordnet `self.train_data` zufällig mit dem Befehl `random.shuffle` um und ruft dann die Methode `batches` (s.u.) auf, um eine Folge von Batches zu generieren, die zurückgegeben werden.

- `dev_batches(self, max_batch_size)`: analog zu `train_batches`, aber für Developmentdaten und ohne Shuffling.

- `batches(self, data, max_batch_size)`: Diese Methode erzeugt aus den Eingabedaten `data` eine Folge von Batches für die Verarbeitung mit dem neuronalen Netz. Die Summe der Längen aller Wörter und Lemmata in einem Batch soll so nahe wie möglich an `max_batch_size` sein, darf diesen Wert aber nicht übersteigen.

  Für jedes Batch wird die Methode `self.make_batch_tensor(words, char_to_id, padding)` einmal für die Wortfolge und einmal für die Lemmafolge im Batch aufgerufen, um die Tensoren zu erzeugen. Das dritte Argument `padding` ist bei der Wortfolge eine leere Liste und bei der Lemmaliste eine Liste mit genau einem Paddingsymbol. Für jedes Batch wird ein Tripel `(src_id_tensor, src_lengths, tgt_id_tensor)` mit yield zurückgegeben. Vergessen Sie nicht, auch das letzte Batch auszugeben, das kleiner sein kann.

- `make_batch_tensor(self, words, char_to_id, padding=[])` wandelt die Liste der Wörter mit Hilfe der Abbildungstabelle `char_to_id` in Listen von Buchstaben-Nummern um, merkt sich die Längen der Wörter in `lengths` und erzeugt mit dem Torch-Befehl `pad_sequence` einen gepaddeten Tensor `id_tensor`. Zurückgegeben werden `id_tensor` und `lengths`.

- `input_batches(self, filename, max_batch_size)`: Diese Methode wird beim Lemmatisieren aufgerufen. Sie erhält den Namen einer Datei mit einem Wort pro Zeile als Eingabe und erzeugt analog zu der Methode `batches` eine Folge von Batch-Tensoren. Mit `yield` wird eine Folge von Quadrupeln `(src_words, src_id_tensor, src_lengths, max_tgt_len)` generiert. Dabei ist `src_words` die Liste der Wörter im Batch und

  ```
  max_tgt_len = max(src_lengths) * self.max_len_factor + 4
  ```

  die Gesamtlänge der Wörter plus `max_tgt_len` mal der Batchgröße. Dieser Wert darf `max_batch_size` nicht übersteigen. Lesen Sie nicht die ganze Datei auf einmal ein, sondern geben Sie das nächste Batch aus, sobald es voll ist.

- `tgt_ids_to_lemma(self, tgt_char_ids)`: Diese Methode erhält eine Folge von Buchstaben-Nummern als Eingabe und entfernt alle Nummern ab dem (nicht immer vorhandenen) pad_id-Element. Die übrigen Nummern werden mit `self.id_to_tgt_char` auf Buchstaben abgebildet und das Ergebnis als String zurückgegeben.

Encoder

Das Encoder-Decoder-Netzwerk besteht aus einem Encoder, einem Attention-Modul und einem Decoder. Erstellen Sie dafür eine Datei `model.py`.

Die Klasse `Encoder(vocab_size, emb_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`-Methode 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, effizient 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 zurücktransformiert. Die Repräsentationen des bidirektionalen Encoders haben die doppelte `hidden_size`-Länge und werden durch eine Feedforward-Ebene mit tanh-Aktivierungsfunktion auf einen Tensor `output` mit einfacher Länge der Repräsentationen transformiert.

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. Sie speichern dazu den zweiten Rückgabewert des BiLSTMs in den Variablen `hidden` und `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.

Die `forward`-Methode des Attention-Moduls wird im Training 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 (= Dimension 2) 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 und den SoftMax zu einem PyTorch-Sequential-Modul zusammen und fügen Sie an passender Stelle Dropout hinzu.

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.

In der Decoder-Embedding-Ebene `self.embedding` und der Ausgabe-Ebene `self.output_layer` verwenden Sie dieselben Embedding-Vektoren ("Tied Embeddings"). Daher benötigen Sie die Projektions-Ebene `self.output_projection`, welche die LSTM-Ausgabe-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 es die Rückwärts-Endzustände als Startzustände erhält.

Für alle weiteren Decoder LSTMs wird zunächst das Attention-Modul aufgerufen, um den Kontext-Tensor zu berechnen. Der Kontext-Tensor wird mit der Ausgabe des vorherigen LSTMs in der letzten Dimension konkateniert und vom aktuellen LSTM verarbeitet. Allen LSTMs werden die Rückwärts-Endzustände als Startzustände übergeben.

Die Ausgabe des letzten LSTMs wird erst durch die Projektions-Ebene und dann durch die Ausgabe-Ebene verarbeitet, in beiden Fällen ohne eine Aktivierungsfunktion. Der erhaltene Tensor mit Logitwerten wird zurückgegeben.

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 zurückgegebenen Tensors korrekt sind.