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` für die Vorverarbeitung der Lemmatisiererdaten. 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 gegebenen 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 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. Schließlich 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`, 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 sollte dabei den Wert `max_batch_size` nicht übersteigen. Für jedes Batch wird die (innerhalb der Methode `batches` definierte) Unter-Funktion `process_batch` aufgerufen. Diese Funktion wandelt die Liste der Wörter mit Hilfe der Abbildungstabelle `self.src_char_to_id` in Listen von Buchstaben-Nummern um, merkt sich die Längen der Wörter in `src_lengths` und erzeugt mit dem Torch-Befehl `pad_sequence` einen gepaddeten Tensor `src_id_vecs`.

  Die Lemma-Buchstaben werden mit `self.tgt_char_to_id` auf Nummern abgebildet. Dann wird am Anfang und Ende jeder Folge ein Paddingsymbol hinzugefügt und ein gepaddeter Tensor `tgt_id_vecs` erzeugt. Das Tripel `(src_id_vecs, src_lengths, tgt_id_vecs)` wird mit yield zurückgegeben. Vergessen Sie nicht, auch das letzte Batch auszugeben, das kleiner sein kann.

- `input_batches(self, file, max_batch_size)`: Diese Methode erhält eine Datei nur mit Wörtern als Eingabe und erzeugt analog zu der Methode `batches` eine Folge von Batch-Tensoren. Mit `yield` wird eine Folge von Quadrupeln (`srcs`, `src_id_vecs`, `src_lengths`, `max_tgt_len`) generiert. Dabei ist `srcs` 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 sollte `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_chars(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 zurückgegeben.

Encoder

Das Encoder-Decoder-Netzwerk besteht aus einem Encoder, einem Attention-Modul und einem Decoder.

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`-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` 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.

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.

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 Vektors stimmen.