CS310 Natural Language Processing
Assignment 5: Dependency Parsing
Total points: 50 + (10 bonus)

Tasks

- Train a feed-forward neural network-based dependency parser and evaluate its performance on the provided treebank dataset.
- (Bonus 1) Implement the arc-eager approach for the parser
- (Bonus 2) Re-implement the feature extractor using Bi-LSTM as the backbone encoder.

Submit

- See REAME.md for submission requirements.
- Report document is not mandatory for Task 1-5. It is required for bonus Task 6 and 7.

Requirements

1) Implement the Feature Extractor (15 points)

Following Chen and Manning (2014), the parser will use features from the top three words on stack s_0, s_1, s_2, the top three words on buffer b_0, b_1, b_2. Using the implementation from the lab practice, they correspond to `stack[-1], stack[-2], stack[-3]`, and `buffer[-1], buffer[-2], buffer[-3]`, respectively. We also include their POS tags as feature, i.e., $t_1, \dots, t_n$.
Specifically, words and tags are first associated with embedding vectors: $e(w_i)$ and $e(t_i)$ for $i = 1, \dots, n$, where n is the length of input sentence. Then the feature for the current configuration c is:

$$\phi(c) = e(s_2) \oplus e(s_1) \oplus e(s_0) \oplus e(b_0) \oplus e(b_1) \oplus e(b_2) \oplus$$
$$e(ts_2) \oplus e(ts_1) \oplus e(ts_0) \oplus e(tb_0) \oplus e(tb_1) \oplus e(tb_2)$$

Here, ts_i is the POS tag of the ith word on stack, and tb_i is the tag of the ith word on buffer. Note that they are NOT the ith word in the sentence.
In some configurations, the stack or buffer may have fewer than 3 words. In those cases, use pseudo tokens "<NULL>" to replace the missing blanks, which is also associated with an embedding $e("\langle NULL \rangle")$. So is the special token "<ROOT>", $e("\langle ROOT \rangle")$. Their POS tags (which do not exist) can be some pseudo values as well.
For example, if the buffer contains ["apple", "trees", "grow"] and the stack contains ["<ROOT>", "the"], then the concatenated word vectors are:

$$e("\langle NULL \rangle") \oplus e("\langle ROOT \rangle") \oplus e("the") \oplus e("apple") \oplus e("trees") \oplus e("grow")$$

You can infer the concatenated tag vectors similarly.
Chen et. al. (2014) uses embedding size $d = 50$. You can use larger values.

2) Implement Scoring Oracle (10 points)

The feature extracted will be passed to the scoring **oracle** that determines the transition action t given the feature $\phi(c)$ extracted from the feature function. The scoring function could be multiple layer perceptron (MLP): $Score_\theta(\phi(c), t) = MLP_\theta(\phi(c))[t]$, where $MLP_\theta(x) = W^{[2]} \cdot \tanh(W^{[1]} \cdot x + b^{[1]}) + b^{[1]}$
(Chen et. al. (2014) uses hidden layer size $h = 200$ and tanh activation. You can use ReLU.)
The oracle model should be implemented as a sub-class of `torch.nn.Module`. Two versions of oracle are required: `BaseModel` (using words only) and `WordPOSModel` (using words + POS)

3) Training (5 points)

In which the `forward` function takes input a sequence of training data instances A training instance has two parts, X and y. X represents the current configuration states, that is, the integer IDs of the 6 words in stack and buffer, and their POS tag IDs. You should implement how these IDs are converted to embedding vectors as described in Task 1 and 2, and then compute the loss(negative log-likelihood loss) between the predicted transition action $\hat{y}$ and the ground truth y.

4) Implement the Parser for Inference (15 points)

We want the parser to be able to parse an unannotated input sentence, so you should also implement a `parse_sentence` function (as a member of a `Parser` class) that takes a sequence of words as input, and return the parsed tree. This function is very similar to forward in terms of computation, with the exception that it does not know the gold-standard transition actions. This function's job is to accomplish the following transition-based greedy parsing:

Algorithm 1 Greedy transition-based parsing

Input: sentence $s = w_1, \ldots, x_w,\ t_1, \ldots, t_n$, parameterized function $\text{SCORE}_\theta(\cdot)$ with parameters θ.
$c \leftarrow \text{INITIAL}(s)$
while not $\text{TERMINAL}(c)$ **do**
  $\hat{t} \leftarrow \arg\max_{t \in \text{LEGAL}(c)} \text{SCORE}_\theta(\phi(c), t)$
  $c \leftarrow \hat{t}(c)$
return $tree(c)$

Figure from Kiperwasser and Goldberg (2016)

There are several notable places of the above algorithm:

- ✧ You can implement the exit condition TERMINAL(c) using an **if** statement to check if "<ROOT>" is the only element on stack and the buffer is empty.
- ✧ The argmax operation means that you select the highest scoring transition, but unfortunately it is possible that the highest scoring transition is not possible. Therefore, instead of selecting the highest-scoring action, you should select the highest scoring *permitted* transition, indicated by the $t \in \text{LEGAL}(c)$ subscript.

5) Evaluation (5 points)

Run the evaluation script and check the LAS and UAS scores on the dev and test sets.

** Grading rubrics **

- If your model is implemented correctly and the training code can run without problem, then you get the full credits for Task 1, 2, and 3.
- If your UAS score x>=70% on the **test** set, then you get **full points** for Task 4 and 5.
- If 50%<=x<60%, you get **10 points** for Task 4 and **2.5 points** for Task 5
- If x < 50%, then you receive **0 points** for Task 4 and 5

****

Bonus Tasks

6) Implement the arc-eager approach (5 points)

You only need to modify the `get_training_instances` function and the `State` class. You do not need to actually train the parser. Note that you should add the new "Reduce" action to the code. Prove that your implementation is different from the arc-standard approach by showing an example. (The example should come in the report)

7) Implement the Bi-LSTM-based encoder (5 points)

Following Kiperwasser and Goldberg (2016), which uses words and POS tags as features. Given a n-words input sentence $w_1, \ldots, w_n$ together with the corresponding POS tags $t_1, \ldots, t_n$, we associate each word and POS tag with an embedding vector $e(w_i)$ and $e(t_i)$, and create a sequence of input vectors $x_{1:n}$ in which each x_i is the concatenation of the word and POS embeddings:

$$x_i = e(w_i) \oplus e(t_i)$$

The input $x_{1:n}$ is then passed to a Bi-LSTM encoder, which produces a hidden representation for each input element:

$$v_i = \text{BiLSTM}(x_i, i)$$

The feature function is the concatenated Bi-LSTM vectors of the top 3 items on the stack and the first item on the buffer. That is, for a configuration $c = \{\text{stack: } [\ldots|s_2|s_1|s_0], \text{buffer: } [b_0|\ldots]\}$, the feature for this configuration is:

$$\phi(c) = v_{s_2} \oplus v_{s_1} \oplus v_{s_0} \oplus v_{b_0}, \text{ where } v_i = \text{BiLSTM}(x_{1:n}, i)$$

In our implementation for the configuration state in the lab, the stack top s_0 corresponds to `state.stack[-1]`, and the buffer front b_0 corresponds to `state.buffer[-1]`, and so forth for other elements. If the stack contains fewer than 3 words or the buffer is empty, then use the special token "<NULL>" to fill up the blanks.

For example, if the buffer = […, "apple"] and stack = ["<ROOT>", "the"], then we should use $e(\text{"}\langle NULL\rangle\text{"}) \oplus e(\text{"}\langle ROOT\rangle\text{"}) \oplus v_{\text{"}the\text{"}} \oplus v_{\text{"}apple\text{"}}$ as the feature.

For another example, if the buffer is empty and stack = ["<ROOT>", "the"], then we should use $e(\text{"}\langle NULL\rangle\text{"}) \oplus e(\text{"}\langle ROOT\rangle\text{"}) \oplus v_{\text{"}the\text{"}} \oplus e(\text{"}\langle NULL\rangle\text{"})$ as the feature

Note that here the vectors for "<NULL>" and "<ROOT>" are not the output from Bi-LSTM because they not actual words, and they are just returned by the embedding table.

(If your Bi-LSTM implementation is complex, explain your code and demonstrate how performance changes in the report)

** Grading rubrics **

- For Task 6, you only need to change the `State` class and the `get_training_instances` function by adding "Reduce" to the transition action set. You do not need to train a new parser.
- For Task 7, you need to have better LAC and UAC scores than Task 5.

References

Chen, D. and C. D. Manning (2014). A fast and accurate dependency parser using neural networks. Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP).

Kiperwasser, E. and Y. Goldberg (2016). "Simple and accurate dependency parsing using bidirectional LSTM feature representations." Transactions of the Association for Computational Linguistics **4**: 313-327.