Humboldt-Universität zu Berlin
Institut für Informatik
Prof. Dr. Alan Akbik

Berlin, 17.05.2024
Lehrstuhl Maschinelles Lernen

Introduction to Natural Langauge Processing

Exercise 4

Submission: Hand your homework until 27.05.2024 (Monday), 23:59 via Github Classroom. Each exercise is worth a total of **10 points** and you need **80%** (of total points over all exercises) to be admitted to the final exam. Sometimes, there are optional tasks that will give you extra points. Exercises in Github Classroom should be completed and submitted individually. However, we encourage you to work on the problems in teams. Please note the information available in Moodle: `https://hu.berlin/nlp24-moodle`.

Github Classroom: `https://classroom.github.com/a/VBjsdHXJ`

Task 1 (Well-formed parantheses) **1 + 1 + 1 + 1 = 4 points**

In this task, we want to identify whether a string is a valid combination of parentheses. We aim to address this issue using the concept of recurrence, as it is utilized in recurrent neural networks or similar types (LSTMs, GRUs, etc.).
The classifier is supposed to detect whether the string is a valid combination of parentheses. The string is valid if it consists only of pairs of parentheses, where the opening parentheses come first. Some examples:

- "": Valid
- "()": Valid
- "(())": Valid
- "()()": Valid
- "())(": Invalid
- ")(": Invalid

Task: Determine the weights for a classifier with a single RNN Layer, a hidden dimension of 2, an input dimension of 2, and a ReLU activation function (which maps all negative values to zero). The last hidden state passed to a single 2x2 linear layer (with a bias term) to determine whether the input string is "well-formed." Your task is to set the weights for the model in file `task_1/weights.txt`. The run script for this task has four modes:

- `python run_task_1.py examples`: Show a random list of example strings.
- `python run_task_1.py single`: Compute the hidden states and final prediction for a single random example.
- `python run_task_1.py interactive`: Compute the hidden states and final prediction for an instance that you can interactively enter (you can use this to test your model on the examples we provide on the exercise sheet).

- `python run_task_1.py evaluate`: Evaluate your model on a larger number of samples and print the accuracy

Think about how you would solve this using a pushdown automaton. What information would be stored in the stack? An ideal classifier will have 100% and fulfill all of the following subtasks (try to solve them step by step).

(a) One of your hidden dimensions should track the excess of open parentheses. E.g.:
- "(()": 1
- "((((": 4
- "(()(": 2

(b) The classifier should output 0 (i.e., not well formed) if there is an excess of open parentheses. It should output 1 (well-formed) for well-formed examples. This should allow you to reach $\approx 75\%$ accuracy.

(c) The classifier should also track whether an excess closing parenthesis was encountered and output 0 accordingly, which should occur in certain cases:
- ")"
- "())"
- "(()))"
- "(()()))"

(d) When an excess closing parentheses is encountered, it's vital to note that the program's state becomes invalid and should not change, even if more opening parentheses are found. You need to prevent the counter from going above 0 again to achieve that. I.e., the classifier should correctly classify the following cases as not well-formed:
- ")("
- ")(()"
- "())((())"

With the final classifier, you should be able to reach an accuracy of 100%.

Task 2 (RNN Model in PyTorch) **2 + 2 + 2 = 6 points**

In this task, we are building a part-of-speech tagger in PyTorch as in the last exercise but using recurrent networks. As you know from the lecture, the underlying concept of recurrent networks is to save the output of a layer and feed this output back to the input.
We implemented all data-loading and processing logic. You only need to complete the model class. We provide you with tests to test your implementations (basic checks, primarily for expected tensor shapes). You can find them in the `tests` directory. You can test your implementations with:

```
python -m pytest
```

(a) Complete the class `RecurrentModel` in `task_2/model.py`. You are allowed to use PyTorch's `RNNCell` class **but not the RNN class**. Add the `RNNCell` to the constructor and implement the forward method. If you look into the *forward* method, you see that everything is already implemented except for the function *rnn_forward*. This method

takes embeddings as input and should return the hidden states after applying the recurrent network.

Summarized:

- Add an instance attribute to the class (`RNNCell`).
- Implement the recurrent forward method *rnn_forward*: Iterate over each token embedding, pass the embedding through your `RNNCell`, and pass the returned hidden state back into the `RNNCell` together with the next token. Check out this example for reference: https://pytorch.org/docs/stable/generated/torch.nn.RNNCell.html.

(b) Extend your *rnn_forward* to be bidirectional. As we saw in the last exercise, it may be beneficial to consider context before and after a word to obtain a better representation. However, unidirectional RNNs only capture context before the current word. Add a second `RNNCell` to your model to account for this issue.

- Add a second instance attribute to your class for the backward RNN (using `RNNCell` again).
- Extend the *rnn_forward* to be bidirectional by iterating over the embeddings in both directions (forward and backward). You should handle both cases (uni- and bidirectional) in the *rnn_forward* using the flag `self.bidirectional`. Once you obtain both hidden states (from your forward and backward RNN), you need to concatenate the representations into one (using `torch.cat`). Be sure to check that you concatenate them in the correct order. For example, the first token gets the first hidden state from the forward RNN and the last hidden state from the backward RNN!
- Important: Each token now has two hidden states, one from the forward and one from the backward RNN. The classifier layer typically expects input features of size `hidden_dim` for each token. However, your input features to the classifier are now twice as large. Please adjust this in the constructor.

(c) Train two models (unidirectional and bidirectional) using your implementation using the following hyperparameters:

1. Learning Rate: 0.1 (command line argument `--learning_rate X`)
2. Hidden Dimension: 100 (command line argument `--hidden_dim X`)
3. Train Epochs: 20 (command line argument `--epochs X`)
4. Seed: 42 (command line argument `--seed X`)
5. Bidirectional RNN (command line argument `-bidirectional`)

Training should take at most 5 minutes on standard machines. Check the argument parser in `run_task_2.py` for details. Example execution:

```
python run_task_2.py --lr 0.1
```

Create a final text file in a tabular format that logs the hyperparameter used and the test accuracy. Important: This must be a separate file, not the log files in the result folder. This helps us to evaluate your implementation quickly.

(d) OPTIONAL: Extend your model by using `LSTMCell`, as demonstrated here: https://pytorch.org/docs/stable/generated/torch.nn.LSTMCell.html. Training takes some time on standard machines without GPUs, so it is not necessary to do it.