HW3: Dependency Parsing

CSCI 662: Fall 2022

out: Oct 24, 2022
due: Nov 16, 2022

In this assignment, you will build an **arc-standard** shift-reduce neural dependency parser, similar to the one used by 'A Fast and Accurate Dependency Parser Using Neural Networks' by Danqi Chen and Chris Manning[1]. A mandatory simplified form is specified (see below), however you may also try implementing their exact model, implementing arc-eager parsing, or incorporating other modifications beyond theirs in your 'extra mile' work. Note that, unlike in previous assignments, the key machine learning portion of this assignment *should* be written in PyTorch. This is a good opportunity for you to use your google cloud credits, as the training time may be long unless you are using GPUs. It is *not* recommended that you train models on Vocareum.

The most time-consuming part of the homework, however, may be spent in properly forming a dependency configuration (i.e. stack, buffer, label) and feature set from the configuration. **It is strongly advised you do not wait until the last minute to do this homework!**

Data provided

We provide you with training, and dev dependency trees (converted from Penn Tree Bank) in CoNLL format. Note: this data is not generally available to the public (rights are held by LDC) so please do not distribute! The data has one word per line, with an empty space signifying a sentence break. Each line is in CoNLL data format, and has 10 tab-separated fields as follows:

1. current sentence word number (1-based)
2. the word
3. lemma form of the word
4. coarse-grained POS tag
5. fine-grained POS tag (in practice, both of these are the same)
6. morphological features (in practice, just the token '-')
7. token id of this word's parent (head), or 0 if this word's parent is the root. If this data has not yet been provided (i.e. the sentence isn't parsed) it is '-'.

[1] https://nlp.stanford.edu/pubs/emnlp2014-depparser.pdf

8. dependency label for the relation of this word to its parent. If this data has not yet been provided (i.e. the sentence isn't parsed) it is '-'.
9. other dependencies (in practice, just '-')
10. other info (in practice, just '-')

What you need to do (high level)

1. Convert projective members of the training and dev data into a (feature vector, label) format suitable for training
2. Build the architecture of the parser and then train a simple feed-forward model in PyTorch that takes in the converted data and learns to predict the labels.
3. Given test sentences in CoNLL format (with fields 7 and 8 of each line set to '-'), use the model to generate parse trees. Output CoNLL format where fields 1-6, 9, 10 are as before, but 7 and 8 now contain dependency tree info.

What features to implement? What can be different about your code compared to Danqi's code?

- Your feature set (in your core submission) should be the same as that described in section 3.1 of the paper under the heading "The choice of S^w, S^t, S^l." Please feel free to discuss on Piazza if the description is unclear; I personally think the narrative is a good model for how to write a clear paper. You may wish to consider additions or modifications to the feature set in your 'extra mile' section.
- Use the hyperparameters, learning method, random initialization, dropout, and regularization described in section 3.2. These should all be easy switches or simple additions in PyTorch; if something is not straight forward, let's discuss it in Piazza!
- Danqi learned POS tag and label arc embeddings from scratch but used pretrained word embeddings; you can learn all embeddings from scratch.
- Danqi implemented both arc-standard and arc-eager; you are only required to implement **arc-standard**.
- Rather than hand-implement the cubic activation function (I don't think such a function is a standard part of pytorch) you are welcome to use tanh for activation.
- You do not need to use the precomputation trick described in section 3.3. In practice parsing is fast enough on my gpu-free laptop.

Required Code/models

The following files are mandatory and should be uploaded to Vocareum; other supporting files (e.g. libraries, other programs) may also be a good idea to create and should also be uploaded. All code should be in Python 3.

- README describing the rest of the files and how to use them to prepare data, train a model, and run that model on new data. *Additionally* the README should describe the format of preparedata.py (i.e. which features are in which positions). If you make modifications beyond the base feature set, document the base set and the extended set separately.
- `preparedata.py` converts CoNLL data (train and dev) into features of the parser configuration paired with parser decisions. This should be human-readable, i.e. a text file of words/labels. The format should be described in the README.
- `train.py` trains a model given data preprocessed by `preparedata.py` and writes a model file.

- `parse.py`, given a trained model file (and possibly vocabulary file) reads in CoNLL data and writes CoNLL data where fields 7 and 8 contain dependency tree info. The syntax of the file should be `python parse.py -m [modelfile] -i [inputfile] -o [outputfile]`.
- `dev.converted` is the text file containing the results of running your `preparedata.py` on the dev data with the base feature set.
- `train.model` is the model file (also possibly vocab file named `train.vocab`), the result of running train.py on training and dev with base feature set.
- `LASTNAME_HW3.pdf` where LASTNAME is your last (family) name. This is your report for this HW.

Suggestions on how to structure your code

- `preparedata.py` should take in a dependency tree and, based on the heuristics discussed in class, determine parser actions, which will alter the parser configuration, from which the feature set can be determined. Meanwhile, `parse.py` should take in a sentence and from the parser configuration generate labels, which will determine each subsequent parser configuration (from which features can be determined). Both files, then, should point to a third library file which contains code that, given configuration at time t c_t and label l, determines c_{t+1}.
- Many of the features will be empty values (e.g. the leftmost child of the leftmost child of the second word on the stack...when the stack contains one item). This is a feed-forward network, though, so some placeholder `None` value should appear in these cases.
- Shift-reduce parsers only admit projective trees. Some of the training/dev data (a very small amount) is non-projective. These sentences should be detected and removed from training/dev. Remember, a tree is non-projective if it has crossing arcs. It should be straight forward to detect crossing arcs in `preparedata.py` upon reading in the sentence.
- There are different ways to structure your code but it will generally be important to design your parser class with modularity in mind. Generating training data for the model and using the parser with a trained model are very similar with the main difference being that one uses an oracle (ground truth) to generate actions, the other uses the predictions of a model. Your code structure should probably reflect that. High-level functions you may want to have in your parser class include initializing to the start state for a sentence, performing an action, getting the features of a current parse state, getting the ground truth action. The provided skeleton is one such way to develop your code, though more supporting functions are almost certainly a good idea.

Ideas for 'Extra Mile' work

This list is not exhaustive. As always, better analysis of the things you try and justification for what you try will lead to more credit.

- As noted above, modify the features
- Try building an arc-eager model. Do the results differ much, in terms of LAS/UAS, run time, training time, etc?
- Try different sets of external word embeddings, including the Collobert ones Danqi used and better ones that have come out since.
- Try to perfectly replicate Danqi's work: use a cubic nonlinear function, use external embeddings, use POS tagger-generated tags, etc.

- Compare different (reasonable) hyperparameter/learning configuration variants; hidden unit size, number of layers, learning rates, gradient descent variants and schedules, etc.
- Ablate feature sets, introduce other new (meaningful) features
- Look for papers that cited Danqi's paper. See if they came up with any advances and try implementing them.

Evaluation and Debugging

We have provided the Stanford parser jar, which is what we use to score your output on a hidden test set on Vocareum after parsing it using your `parse.py` and `train.model`. It works as follows: assuming you have generated e.g. file `parse.out` for an input file `parse.in`, do `java -cp stanford-parser.jar edu.stanford.nlp.trees.DependencyScoring -g parse.in -conllx True -s parse.out`.

Your Report

Your report should at a *minimum*:

- Contain a narrative expressed in the abstract and carrying through the work to get your central point across.
- Describe clearly the features used to predict labels.
- Report on the fraction of the training and dev data that was not projective and removed.
- Report on labeled and unlabeled accuracy scores on test for any models you try (submission report on Vocareum will include the scores.)
- Where relevant, show hyperparameters, discuss overfitting and loss convergence. Use graphs and tables *appropriately*, not superfluously. This means the graphs/tables should emphasize the message you are delivering, not simply be in place without thinking about why you are using that particular medium to convey an idea.

Use the ACL style files : `https://github.com/acl-org/acl-style-files`

Your report should be at least two pages long, including references, and not more than four pages long, not including references (i.e. you can have up to four pages of text if you need to). Just like a conference paper or journal article it should contain an abstract, introduction, experimental results, and conclusion sections (as well as other sections as deemed necessary). Unlike a conference paper/journal article, a complete related works section is not obligatory (but you may include it if it is relevant to what you do).

Grading

Grading will be roughly broken down as follows:

- about 50% – did you clearly communicate your description of what you implemented, how you implemented it, what your experiments were, and what conclusions you drew from them? This includes appropriate use of graphics and tables where warranted that clearly explain your point. This also includes well written explanations that tell a compelling story. Grammar and syntax are a small part of this (maybe 5% of the grade, so 10% of this section) but much more important is the narrative you tell. Also a part of this is that you clearly acknowledged your sources and influences with appropriate bibliography and, where relevant, cited influencing prior work.
- about 20% – is your code correct? Did you implement what was asked for, and did you do it correctly?

- about 20% – is your code well-written, documented, and robust? Will it run from a different directory than the one you ran it in? Does it rely on hard-codes? Is it commented and structured such that we can read it and understand what you are doing?
- about 10% – did you go the extra mile? Did you push beyond what was asked for in the assignment, trying new models, features, or approaches? Did you use motivation (and document appropriately) from another researcher trying the same problem or from an unrelated but transferrable other paper?

Rules

- This is an individual assignment. You may not work in teams or collaborate with other students. You must be the sole author of 100% of the code you turn in.
- You may not look for coded solutions on the web, or use code you find online or anywhere else. You can and are encouraged to read material beyond what you have been given in class (see above) but should not copy code.
- You may not download the data from any source other than the files provided on Vocareum, and you may not share the data with others outside this class, or keep the data after the class is over (the copyright is held by LDC).
- You may use packages in the Python Standard Library as well as PyTorch or, as needed, other machine learning packages. You may *not* use any dependency parsing code written by others, including feature calculation or configuration manipulation code.
- You may use external resources to learn basic functions of Python (such as reading and writing files, handling text strings, and basic math), but the extraction and computation of model parameters, as well as the use of these parameters for classification, must be your own work.
- Failure to follow the above rules is considered a violation of academic integrity, and is grounds for failure of the assignment, or in serious cases failure of the course.
- We use plagiarism detection software to identify similarities between student assignments, and between student assignments and known solutions on the web. Any attempt to fool plagiarism detection, for example the modification of code to reduce its similarity to the source, will result in an automatic failing grade for the course.
- If you have questions about what is and isn't allowed, post them to Piazza!