
Berkeley Neural Parser

A high-accuracy parser with models for 11 languages, implemented in Python. Based on Constituency Parsing with a Self-Attentive Encoder from ACL 2018, with additional changes described in Multilingual Constituency Parsing with Self-Attention and Pre-Training.

New February 2021: Version 0.2.0 of the Berkeley Neural Parser is now out, with higher-quality pre-trained models for all languages. Inference now uses PyTorch instead of TensorFlow (training has always been PyTorch-only). Drops support for Python 2.7 and 3.5. Includes updated support for training and using your own parsers, based on your choice of pre-trained model.

Contents

1. Installation
2. Usage
3. Available Models
4. Training
5. Reproducing Experiments
6. Citation
7. Credits

If you are primarily interested in training your own parsing models, skip to the Training section of this README.

Installation

To install the parser, run the command:

```
1 $ pip install benepar
```

Note: benepar 0.2.0 is a major upgrade over the previous version, and comes with entirely new and higher-quality parser models. If you are not ready to upgrade, you can pin your benepar version to the previous release (0.1.3).

Python 3.6 (or newer) and PyTorch 1.6 (or newer) are required. See the PyTorch website for instruction on how to select between GPU-enabled and CPU-only versions of PyTorch; benepar will automatically use the GPU if it is available to pytorch.

The recommended way of using benepar is through integration with spaCy. If using spaCy, you should install a spaCy model for your language. For English, the installation command is:

```
1 $ python -m spacy download en_core_web_md
```

The spaCy model is only used for tokenization and sentence segmentation. If language-specific analysis beyond parsing is not required, you may also forego a language-specific model and instead use a multi-language model that only performs tokenization and segmentation. One such model, newly added in spaCy 3.0, should work for English, German, Korean, Polish, and Swedish (but not Chinese, since it doesn't seem to support Chinese word segmentation).

Parsing models need to be downloaded separately, using the commands:

```
1 >>> import benepar
2 >>> benepar.download('benepar_en3')
```

See the Available Models section below for a full list of models.

Usage

Usage with spaCy (recommended)

The recommended way of using benepar is through its integration with spaCy:

```
1 >>> import benepar, spacy
2 >>> nlp = spacy.load('en_core_web_md')
3 >>> if spacy.__version__.startswith('2'):
4     nlp.add_pipe(benepar.BeneparComponent("benepar_en3"))
5 else:
6     nlp.add_pipe("benepar", config={"model": "benepar_en3"})
7 >>> doc = nlp("The time for action is now. It's never too late to do something.")
8 >>> sent = list(doc.sents)[0]
9 >>> print(sent._.parse_string)
10 (S (NP (NP (DT The) (NN time)) (PP (IN for) (NP (NN action))))) (VP (VBZ is) (ADVP (RB now))) (. .))
11 >>> sent._.labels
12 ('S',)
13 >>> list(sent._.children)[0]
14 The time for action
```

Since spaCy does not provide an official constituency parsing API, all methods are accessible through the extension namespaces `Span._` and `Token._`.

The following extension properties are available: - `Span._.labels`: a tuple of labels for the given span. A span may have multiple labels when there are unary chains in the parse tree. - `Span._.parse_string`: a string representation of the parse tree for a given span. - `Span._.constituents`: an iterator over `Span` objects for sub-constituents in a pre-order traversal of

the parse tree. - `Span._.parent`: the parent `Span` in the parse tree. - `Span._.children`: an iterator over child `Spans` in the parse tree. - `Token._.labels`, `Token._.parse_string`, `Token._.parent`: these behave the same as calling the corresponding method on the length-one `Span` containing the token.

These methods will raise an exception when called on a span that is not a constituent in the parse tree. Such errors can be avoided by traversing the parse tree starting at either sentence level (by iterating over `doc.sents`) or with an individual `Token` object.

Usage with NLTK

There is also an NLTK interface, which is designed for use with pre-tokenized datasets and treebanks, or when integrating the parser into an NLP pipeline that already performs (at minimum) tokenization and sentence splitting. For parsing starting with raw text, it is **strongly encouraged** that you use `spaCy` and `benepar.BeneparComponent` instead.

Sample usage with NLTK:

```
1 >>> import benepar
2 >>> parser = benepar.Parser("benepar_en3")
3 >>> input_sentence = benepar.InputSentence(
4     words=["'", 'Fly', 'safely', '.', ''],
5     space_after=[False, True, False, False, False],
6     tags=['`', 'VB', 'RB', '.', ''],
7     escaped_words=['`', 'Fly', 'safely', '.', ''],
8 )
9 >>> tree = parser.parse(input_sentence)
10 >>> print(tree)
11 (TOP (S (` `) (VP (VB Fly) (ADVP (RB safely)))) (. .) (' ')))
```

Not all fields of `benepar.InputSentence` are required, but at least one of `words` and `escaped_words` must be specified. The parser will attempt to guess the value for missing fields, for example:

```
1 >>> input_sentence = benepar.InputSentence(
2     words=["'", 'Fly', 'safely', '.', ''],
3 )
4 >>> parser.parse(input_sentence)
```

Use `parse_sents` to parse multiple sentences.

```
1 >>> input_sentence1 = benepar.InputSentence(
2     words=['The', 'time', 'for', 'action', 'is', 'now', '.'],
3 )
4 >>> input_sentence2 = benepar.InputSentence(
```

```
5     words=['It', "'s", 'never', 'too', 'late', 'to', 'do', 'something',  
6           '.'],  
7 >>> parser.parse_sents([input_sentence1, input_sentence2])
```

Some parser models also allow Unicode text input for debugging/interactive use, but passing in raw text strings is *strongly discouraged* for any application where parsing accuracy matters.

```
1 >>> parser.parse("'Fly safely.'") # For debugging/interactive use only  
.
```

When parsing from raw text, we recommend using spaCy and `benepar.BeneparComponent` instead. The reason is that parser models do not ship with a tokenizer or sentence splitter, and some models may not include a part-of-speech tagger either. A toolkit must be used to fill in these pipeline components, and spaCy outperforms NLTK in all of these areas (sometimes by a large margin).

Available Models

The following trained parser models are available. To use spaCy integration, you will also need to install a spaCy model for the appropriate language.

Model	Language	Info
benepar_en3	English	95.40 F1 on revised WSJ test set. The training data uses revised tokenization and syntactic annotation based on the same guidelines as the English Web Treebank and OntoNotes, which better matches modern tokenization practices in libraries like spaCy. Based on T5-small.

Model	Language	Info
benepar_en3_large	English	96.29 F1 on revised WSJ test set. The training data uses revised tokenization and syntactic annotation based on the same guidelines as the English Web Treebank and OntoNotes, which better matches modern tokenization practices in libraries like spaCy. Based on T5-large.

Model	Language	Info
benepar_zh2	Chinese	92.56 F1 on CTB 5.1 test set. Usage with spaCy allows supports parsing from raw text, but the NLTK API only supports parsing previously tokenized sentences. Based on Chinese ELECTRA-180G-large.
benepar_ar2	Arabic	90.52 F1 on SPMRL2013/2014 test set. Only supports using the NLTK API for parsing previously tokenized sentences. Parsing from raw text and spaCy integration are not supported. Based on XLM-R.

Model	Language	Info
benepar_de2	German	92.10 F1 on SPMRL2013/2014 test set. Based on XLM-R.
benepar_eu2	Basque	93.36 F1 on SPMRL2013/2014 test set. Usage with spaCy first requires implementing Basque support in spaCy. Based on XLM-R.
benepar_fr2	French	88.43 F1 on SPMRL2013/2014 test set. Based on XLM-R.
benepar_he2	Hebrew	93.98 F1 on SPMRL2013/2014 test set. Only supports using the NLTK API for parsing previously tokenized sentences. Parsing from raw text and spaCy integration are not supported. Based on XLM-R.

Model	Language	Info
benepar_hu2	Hungarian	96.19 F1 on SPMRL2013/2014 test set. Usage with spaCy requires a Hungarian model for spaCy. The NLTK API only supports parsing previously tokenized sentences. Based on XLM-R.
benepar_ko2	Korean	91.72 F1 on SPMRL2013/2014 test set. Can be used with spaCy's multi-language sentence segmentation model (requires spaCy v3.0). The NLTK API only supports parsing previously tokenized sentences. Based on XLM-R.

Model	Language	Info
benepar_pl2	Polish	97.15 F1 on SPMRL2013/2014 test set. Based on XLM-R.
benepar_sv2	Swedish	92.21 F1 on SPMRL2013/2014 test set. Can be used with spaCy's multi-language sentence segmentation model (requires spaCy v3.0). Based on XLM-R.

Model	Language	Info
benepar_en3_ws_j	English	<p>Consider using benepar_en3 or benepar_en3_large instead. 95.55 F1 on canonical WSJ test set used for decades of English constituency parsing publications. Based on BERT-large-uncased. We believe that the revised annotation guidelines used for training benepar_en3 / benepar_en3_large are more suitable for downstream use because they better handle language usage in web text, and are more consistent with modern practices in dependency parsing and libraries like spaCy.</p> <p>Nevertheless, ¹¹ we provide the benepar_en3_ws_j model for cases where using the</p>

Training

Training requires cloning this repository from GitHub. While the model code in `src/benepar` is distributed in the `benepar` package on PyPI, the training and evaluation scripts directly under `src/` are not.

Software Requirements for Training

- Python 3.7 or higher.
- PyTorch 1.6.0, or any compatible version.
- All dependencies required by the `benepar` package, including: NLTK 3.2, torch-struct 0.4, transformers 4.3.0, or compatible.
- pytokenizations 0.7.2 or compatible.
- EVALB. Before starting, run `make` inside the `EVALB/` directory to compile an `evalb` executable. This will be called from Python for evaluation. If training on the SPMRL datasets, you will need to run `make` inside the `EVALB_SPMRL/` directory instead.

Training Instructions

A new model can be trained using the command `python src/main.py train` Some of the available arguments are:

Argument	Description	Default
<code>--model-path-base</code>	Path base to use for saving models	N/A
<code>--evalb-dir</code>	Path to EVALB directory	<code>EVALB/</code>
<code>--train-path</code>	Path to training trees	<code>data/wsj/train_02-21.LDC99T42</code>
<code>--train-path-text</code>	Optional non-destructive tokenization of the training data	Guess raw text; see <code>--text-processing</code>
<code>--dev-path</code>	Path to development trees	<code>data/wsj/dev_22.LDC99T42</code>

Argument	Description	Default
<code>--dev-path-text</code>	Optional non-destructive tokenization of the development data	Guess raw text; see <code>--text-processing</code>
<code>--text-processing</code>	Heuristics for guessing raw text from destructively tokenized tree files. See <code>load_trees()</code> in <code>src/treebanks.py</code>	Default rules for languages other than Arabic, Chinese, and Hebrew
<code>--subbatch-max-tokens</code>	Maximum number of tokens to process in parallel while training (a full batch may not fit in GPU memory)	2000
<code>--parallelize</code>	Distribute pre-trained model (e.g. T5) layers across multiple GPUs.	Use at most one GPU
<code>--batch-size</code>	Number of examples per training update	32
<code>--checks-per-epoch</code>	Number of development evaluations per epoch	4
<code>--numpy-seed</code>	NumPy random seed	Random
<code>--use-pretrained</code>	Use pre-trained encoder	Do not use pre-trained encoder
<code>--pretrained-model</code>	Model to use if <code>--use-pretrained</code> is passed. May be a path or a model id from the HuggingFace Model Hub	<code>bert-base-uncased</code>
<code>--predict-tags</code>	Adds a part-of-speech tagging component and auxiliary loss to the parser	Do not predict tags
<code>--use-chars-lstm</code>	Use learned CharLSTM word representations	Do not use CharLSTM

Argument	Description	Default
<code>--use-encoder</code>	Use learned transformer layers on top of pre-trained model or CharLSTM	Do not use extra transformer layers
<code>--num-layers</code>	Number of transformer layers to use if <code>--use-encoder</code> is passed	8
<code>--encoder-max-len</code>	Maximum sentence length (in words) allowed for extra transformer layers	512

Additional arguments are available for other hyperparameters; see `make_hparams()` in `src/main.py`. These can be specified on the command line, such as `--num-layers 2` (for numerical parameters), `--predict-tags` (for boolean parameters that default to False), or `--no-XXX` (for boolean parameters that default to True).

For each development evaluation, the F-score on the development set is computed and compared to the previous best. If the current model is better, the previous model will be deleted and the current model will be saved. The new filename will be derived from the provided model path base and the development F-score.

Prior to training the parser, you will first need to obtain appropriate training data. We provide instructions on how to process standard datasets like PTB, CTB, and the SMPRL 2013/2014 Shared Task data. After following the instructions for the English WSJ data, you can use the following command to train an English parser using the default hyperparameters:

```
1 python src/main.py train --use-pretrained --model-path-base models/
  en_bert_base
```

See [EXPERIMENTS.md](#) for more examples of good hyperparameter choices.

Evaluation Instructions

A saved model can be evaluated on a test corpus using the command `python src/main.py test . . .` with the following arguments:

Argument	Description	Default
<code>--model-path</code>	Path of saved model	N/A
<code>--evalb-dir</code>	Path to EVALB directory	<code>EVALB/</code>
<code>--test-path</code>	Path to test trees	<code>data/23.auto.clean</code>
<code>--test-path-text</code>	Optional non-destructive tokenization of the test data	Guess raw text; see <code>--text-processing</code>
<code>--text-processing</code>	Heuristics for guessing raw text from destructively tokenized tree files. See <code>load_trees()</code> in <code>src/treebanks.py</code>	Default rules for languages other than Arabic, Chinese, and Hebrew
<code>--test-path-raw</code>	Alternative path to test trees that is used for evalb only (used to double-check that evaluation against pre-processed trees does not contain any bugs)	Compare to trees from <code>--test-path</code>
<code>--subbatch-max-tokens</code>	Maximum number of tokens to process in parallel (a GPU does not have enough memory to process the full dataset in one batch)	500
<code>--parallelize</code>	Distribute pre-trained model (e.g. T5) layers across multiple GPUs.	Use at most one GPU
<code>--output-path</code>	Path to write predicted trees to (use <code>"-"</code> for stdout).	Do not save predicted trees
<code>--no-predict-tags</code>	Use gold part-of-speech tags when running EVALB. This is the standard for publications, and omitting this flag may give erroneously high F1 scores.	Use predicted part-of-speech tags for EVALB, if available

As an example, you can evaluate a trained model using the following command:

```
1 python src/main.py test --model-path models/en_bert_base_dev=*.pt
```

Exporting Models for Inference

The `benepar` package can directly use saved checkpoints by replacing a model name like `benepar_en3` with a path such as `models/en_bert_base_dev=95.67.pt`. However, releasing the single-file checkpoints has a few shortcomings: * Single-file checkpoints do not include the tokenizer or pre-trained model config. These can generally be downloaded automatically from the HuggingFace model hub, but this requires an Internet connection and may also (incidentally and unnecessarily) download pre-trained weights from the HuggingFace Model Hub * Single-file checkpoints are 3x larger than necessary, because they save optimizer state

Use `src/export.py` to convert a checkpoint file into a directory that encapsulates everything about a trained model. For example,

```
1 python src/export.py export \  
2   --model-path models/en_bert_base_dev=*.pt \  
3   --output-dir=models/en_bert_base
```

When exporting, there is also a `--compress` option that slightly adjusts model weights, so that the output directory can be compressed into a ZIP archive of much smaller size. We use this for our official model releases, because it's a hassle to distribute model weights that are 2GB+ in size. When using the `--compress` option, it is recommended to specify a test set in order to verify that compression indeed has minimal impact on parsing accuracy. Using the development data for verification is not recommended, since the development data was already used for the model selection criterion during training.

```
1 python src/export.py export \  
2   --model-path models/en_bert_base_dev=*.pt \  
3   --output-dir=models/en_bert_base \  
4   --test-path=data/wsj/test_23.LDC99T42
```

The `src/export.py` script also has a `test` subcommand that's roughly similar to `python src/main.py test`, except that it supports exported models and has slightly different flags. We can run the following command to verify that our English parser using BERT-large-uncased indeed achieves 95.55 F1 on the canonical WSJ test set:

```
1 python src/export.py test --model-path benepar_en3_wsj --test-path data  
  /wsj/test_23.LDC99T42
```

Reproducing Experiments

See [EXPERIMENTS.md](#) for instructions on how to reproduce experiments reported in our ACL 2018 and 2019 papers.

Citation

If you use this software for research, please cite our papers as follows:

```
1 @inproceedings{kitaev-etal-2019-multilingual,
2   title = "Multilingual Constituency Parsing with Self-Attention and
3     Pre-Training",
4   author = "Kitaev, Nikita and
5     Cao, Steven and
6     Klein, Dan",
7   booktitle = "Proceedings of the 57th Annual Meeting of the
8     Association for Computational Linguistics",
9   month = jul,
10  year = "2019",
11  address = "Florence, Italy",
12  publisher = "Association for Computational Linguistics",
13  url = "https://www.aclweb.org/anthology/P19-1340",
14  doi = "10.18653/v1/P19-1340",
15  pages = "3499--3505",
16 }
17
18 @inproceedings{kitaev-klein-2018-constituency,
19   title = "Constituency Parsing with a Self-Attentive Encoder",
20   author = "Kitaev, Nikita and
21     Klein, Dan",
22   booktitle = "Proceedings of the 56th Annual Meeting of the
23     Association for Computational Linguistics (Volume 1: Long Papers
24     )",
25   month = jul,
26   year = "2018",
27   address = "Melbourne, Australia",
28   publisher = "Association for Computational Linguistics",
29   url = "https://www.aclweb.org/anthology/P18-1249",
30   doi = "10.18653/v1/P18-1249",
31   pages = "2676--2686",
32 }
```

Credits

The code in this repository and portions of this README are based on <https://github.com/mitchellstern/minimal-span-parser>