PyTorch-NLP Documentation

Release 0.5.0

Michael Petrochuk

Oct 11, 2020

Package Reference

PyTorch-NLP is a library for Natural Language Processing (NLP) in Python. It's built with the very latest research in mind, and was designed from day one to support rapid prototyping. PyTorch-NLP comes with pre-trained embeddings, samplers, dataset loaders, metrics, neural network modules and text encoders. It's open-source software, released under the BSD3 license.

CHAPTER 1

torchnlp.datasets package

The `torchnlp.datasets` package introduces modules capable of downloading, caching and loading commonly used NLP datasets.

Modules return a `torch.utils.data.Dataset` object i.e, they have `__getitem__` and `__len__` methods implemented. Hence, they can all be passed to a `torch.utils.data.DataLoader` which can load multiple samples parallelly using `torch.multiprocessing` workers.

`torchnlp.datasets.`**`wmt_dataset`**(*directory='data/wmt16_en_de', train=False, dev=False, test=False, train_filename='train.tok.clean.bpe.32000', dev_filename='newstest2013.tok.bpe.32000', test_filename='newstest2014.tok.bpe.32000', check_files=['train.tok.clean.bpe.32000.en'], url='https://drive.google.com/uc?export=download&id=0B_bZck-ksdkpM25jRUN2X2UxMm8'*)

The Workshop on Machine Translation (WMT) 2014 English-German dataset.

Initially this dataset was preprocessed by Google Brain. Though this download contains test sets from 2015 and 2016, the train set differs slightly from WMT 2015 and 2016 and significantly from WMT 2017.

The provided data is mainly taken from version 7 of the Europarl corpus, which is freely available. Note that this the same data as last year, since Europarl is not anymore translted across all 23 official European languages. Additional training data is taken from the new News Commentary corpus. There are about 50 million words of training data per language from the Europarl corpus and 3 million words from the News Commentary corpus.

A new data resource from 2013 is the Common Crawl corpus which was collected from web sources. Each parallel corpus comes with a annotation file that gives the source of each sentence pair.

References

- https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/data_generators/translate_ende.py # noqa: E501
- http://www.statmt.org/wmt14/translation-task.html

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the dev split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the dev split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import wmt_dataset  # doctest: +SKIP
>>> train = wmt_dataset(train=True)  # doctest: +SKIP
>>> train[:2]  # doctest: +SKIP
[{
  'en': 'Res@@ um@@ ption of the session',
  'de': 'Wiederaufnahme der Sitzungsperiode'
}, {
  'en': 'I declare resumed the session of the European Parliament ad@@ jour@@ ned␣
↪on...'
  'de': 'Ich erklär@@ e die am Freitag , dem 17. Dezember unterbro@@ ch@@ ene...'
}]
```

torchnlp.datasets.**iwslt_dataset**(*directory='data/iwslt/', train=False, dev=False, test=False, language_extensions=['en', 'de'], train_filename='{source}-{target}/train.{source}-{target}.{lang}', dev_filename='{source}-{target}/IWSLT16.TED.tst2013.{source}-{target}.{lang}', test_filename='{source}-{target}/IWSLT16.TED.tst2014.{source}-{target}.{lang}', check_files=['{source}-{target}/train.tags.{source}-{target}.{source}'], url='https://wit3.fbk.eu/archive/2016-01/texts/{source}/{target}/{source}-{target}.tgz'*)

Load the International Workshop on Spoken Language Translation (IWSLT) 2017 translation dataset.

In-domain training, development and evaluation sets were supplied through the website of the WIT3 project, while out-of-domain training data were linked in the workshop's website. With respect to edition 2016 of the evaluation campaign, some of the talks added to the TED repository during the last year have been used to define the evaluation sets (tst2017), while the remaining new talks have been included in the training sets.

The English data that participants were asked to recognize and translate consists in part of TED talks as in the years before, and in part of real-life lectures and talks that have been mainly recorded in lecture halls at KIT and Carnegie Mellon University. TED talks are challenging due to their variety in topics, but are very benign as they are very thoroughly rehearsed and planned, leading to easy to recognize and translate language.

Note: The order examples are returned is not guaranteed due to `iglob`.

References

- http://workshop2017.iwslt.org/downloads/iwslt2017_proceeding_v2.pdf
- http://workshop2017.iwslt.org/

Citation: M. Cettolo, C. Girardi, and M. Federico. 2012. WIT3: Web Inventory of Transcribed and Translated Talks. In Proc. of EAMT, pp. 261-268, Trento, Italy.

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the dev split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **language_extensions** (list of str) – Two language extensions ['en'l'de'l'it'l'ni'l'ro'] to load.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the dev split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset file.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import iwslt_dataset  # doctest: +SKIP
>>> train = iwslt_dataset(train=True)  # doctest: +SKIP
>>> train[:2]  # doctest: +SKIP
[{
  'en': "David Gallo: This is Bill Lange. I'm Dave Gallo.",
  'de': 'David Gallo: Das ist Bill Lange. Ich bin Dave Gallo.'
}, {
  'en': "And we're going to tell you some stories from the sea here in video.",
  'de': 'Wir werden Ihnen einige Geschichten über das Meer in Videoform erzählen.'
}]
```

torchnlp.datasets.**multi30k_dataset**(*directory='data/multi30k/', train=False, dev=False, test=False, train_filename='train', dev_filename='val', test_filename='test', check_files=['train.de', 'val.de'], urls=['http://www.quest.dcs.shef.ac.uk/wmt16_files_mmt/training.tar.gz', 'http://www.quest.dcs.shef.ac.uk/wmt16_files_mmt/validation.tar.gz', 'http://www.quest.dcs.shef.ac.uk/wmt16_files_mmt/mmt16_task1_test.tar.gz']*)

Load the WMT 2016 machine translation dataset.

As a translation task, this task consists in translating English sentences that describe an image into German, given the English sentence itself. As training and development data, we provide 29,000 and 1,014 triples respectively, each containing an English source sentence, its German human translation. As test data, we provide a new set of 1,000 tuples containing an English description.

Status: Host `www.quest.dcs.shef.ac.uk` forgot to update their SSL certificate; therefore, this dataset does not download securely.

References

- http://www.statmt.org/wmt16/multimodal-task.html
- http://shannon.cs.illinois.edu/DenotationGraph/

Citation

```
@article{elliott-EtAl:2016:VL16,
    author    = {{Elliott}, D. and {Frank}, S. and {Sima'an}, K. and {Specia}, L.}
,
    title     = {Multi30K: Multilingual English-German Image Descriptions},
    booktitle = {Proceedings of the 5th Workshop on Vision and Language},
    year      = {2016},
    pages     = {70--74},
    year      = 2016
}
```

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the dev split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_directory** (*str, optional*) – The directory of the training split.
- **dev_directory** (*str, optional*) – The directory of the dev split.
- **test_directory** (*str, optional*) – The directory of the test split.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **urls** (*str, optional*) – URLs to download.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import multi30k_dataset  # doctest: +SKIP
>>> train = multi30k_dataset(train=True)  # doctest: +SKIP
>>> train[:2]  # doctest: +SKIP
[{
  'en': 'Two young, White males are outside near many bushes.',
  'de': 'Zwei junge weiße Männer sind im Freien in der Nähe vieler Büsche.'
}, {
  'en': 'Several men in hard hatsare operating a giant pulley system.',
  'de': 'Mehrere Männer mit Schutzhelmen bedienen ein Antriebsradsystem.'
}]
```

torchnlp.datasets.**snli_dataset**(*directory='data/', train=False, dev=False, test=False, train_filename='snli_1.0_train.jsonl', dev_filename='snli_1.0_dev.jsonl', test_filename='snli_1.0_test.jsonl', extracted_name='snli_1.0', check_files=['snli_1.0/snli_1.0_train.jsonl'], url='http://nlp.stanford.edu/projects/snli/snli_1.0.zip'*)

Load the Stanford Natural Language Inference (SNLI) dataset.

The SNLI corpus (version 1.0) is a collection of 570k human-written English sentence pairs manually labeled for balanced classification with the labels entailment, contradiction, and neutral, supporting the task of natural language inference (NLI), also known as recognizing textual entailment (RTE). We aim for it to serve both as a benchmark for evaluating representational systems for text, especially including those induced by representation learning methods, as well as a resource for developing NLP models of any kind.

Reference: https://nlp.stanford.edu/projects/snli/

Citation: Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. 2015. A large annotated corpus for learning natural language inference. In Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP).

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import snli_dataset  # doctest: +SKIP
>>> train = snli_dataset(train=True)  # doctest: +SKIP
>>> train[0]  # doctest: +SKIP
{
  'premise': 'Kids are on a amusement ride.',
  'hypothesis': 'A car is broke down on the side of the road.',
  'label': 'contradiction',
  'premise_transitions': ['shift', 'shift', 'shift', 'shift', 'shift', 'shift', ..
↪.],
  'hypothesis_transitions': ['shift', 'shift', 'shift', 'shift', 'shift', 'shift',
↪ ...],
}
```

torchnlp.datasets.**simple_qa_dataset**(*directory='data/', train=False, dev=False, test=False, extracted_name='SimpleQuestions_v2', train_filename='annotated_fb_data_train.txt', dev_filename='annotated_fb_data_valid.txt', test_filename='annotated_fb_data_test.txt', check_files=['SimpleQuestions_v2/annotated_fb_data_train.txt'], url='https://www.dropbox.com/s/tohrsllcfy7rch4/SimpleQuestions_v2.tgz?raw=1*)

Load the SimpleQuestions dataset.

Single-relation factoid questions (simple questions) are common in many settings (e.g. Microsoft's search query logs and WikiAnswers questions). The SimpleQuestions dataset is one of the most commonly used benchmarks for studying single-relation factoid questions.

Reference: https://research.fb.com/publications/large-scale-simple-question-answering-with-memory-networks/

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import simple_qa_dataset  # doctest: +SKIP
>>> train = simple_qa_dataset(train=True)  # doctest: +SKIP
SimpleQuestions_v2.tgz:  15%|| 62.3M/423M [00:09<00:41, 8.76MB/s]
>>> train[0:2]  # doctest: +SKIP
[{
  'question': 'what is the book e about',
  'relation': 'www.freebase.com/book/written_work/subjects',
  'object': 'www.freebase.com/m/01cj3p',
  'subject': 'www.freebase.com/m/04whkz5'
}, {
  'question': 'to what release does the release track cardiac arrest come from',
  'relation': 'www.freebase.com/music/release_track/release',
  'object': 'www.freebase.com/m/0sjc7c1',
  'subject': 'www.freebase.com/m/0tp2p24'
}]
```

torchnlp.datasets.**imdb_dataset**(*directory='data/', train=False, test=False, train_directory='train', test_directory='test', extracted_name='aclImdb', check_files=['aclImdb/README'], url='http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz', sentiments=['pos', 'neg']*)

Load the IMDB dataset (Large Movie Review Dataset v1.0).

This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. Provided a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well. Raw text and already processed bag of words formats are provided.

Note: The order examples are returned is not guaranteed due to `iglob`.

Reference: http://ai.stanford.edu/~amaas/data/sentiment/

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_directory** (*str, optional*) – The directory of the training split.
- **test_directory** (*str, optional*) – The directory of the test split.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset `tar.gz` file.
- **sentiments** (*list of str, optional*) – Sentiments to load from the dataset.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import imdb_dataset  # doctest: +SKIP
>>> train = imdb_dataset(train=True)  # doctest: +SKIP
>>> train[0:2]  # doctest: +SKIP
[{
  'text': 'For a movie that gets no respect there sure are a lot of memorable␣
↪quotes...',
  'sentiment': 'pos'
}, {
  'text': 'Bizarre horror movie filled with famous faces but stolen by Cristina␣
↪Raines...',
  'sentiment': 'pos'
}]
```

torchnlp.datasets.**wikitext_2_dataset**(*directory='data/', train=False, dev=False, test=False, train_filename='wiki.train.tokens', dev_filename='wiki.valid.tokens', test_filename='wiki.test.tokens', extracted_name='wikitext-2', check_files=['wikitext-2/wiki.train.tokens'], url='https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-2-v1.zip', unknown_token='<unk>', eos_token='</s>'*)

Load the WikiText-2 dataset.

The WikiText language modeling dataset is a collection of over 100 million tokens extracted from the set of verified Good and Featured articles on Wikipedia. The dataset is available under the Creative Commons Attribution-ShareAlike License.

Reference: https://einstein.ai/research/the-wikitext-long-term-dependency-language-modeling-dataset

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.
- **unknown_token** (*str, optional*) – Token to use for unknown words.
- **eos_token** (*str, optional*) – Token to use at the end of sentences.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import wikitext_2_dataset  # doctest: +SKIP
>>> train = wikitext_2_dataset(train=True)  # doctest: +SKIP
>>> train[:10]  # doctest: +SKIP
['</s>', '=', 'Valkyria', 'Chronicles', 'III', '=', '</s>', '</s>', 'Senjō', 'no']
```

torchnlp.datasets.**penn_treebank_dataset**(*directory='data/penn-treebank',*
train=False, dev=False, test=False,
train_filename='ptb.train.txt',
dev_filename='ptb.valid.txt',
test_filename='ptb.test.txt',
check_files=['ptb.train.txt'],
urls=['https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb
'https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb.valid.
'https://raw.githubusercontent.com/wojzaremba/lstm/master/data/ptb.test.tx
unknown_token='<unk>', eos_token='</s>')

Load the Penn Treebank dataset.

This is the Penn Treebank Project: Release 2 CDROM, featuring a million words of 1989 Wall Street Journal material.

Reference: https://catalog.ldc.upenn.edu/LDC99T42

Citation: Marcus, Mitchell P., Marcinkiewicz, Mary Ann & Santorini, Beatrice (1993). Building a Large Annotated Corpus of English: The Penn Treebank

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **name** (*str, optional*) – Name of the dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **urls** (*str, optional*) – URLs to download.
- **unknown_token** (*str, optional*) – Token to use for unknown words.
- **eos_token** (*str, optional*) – Token to use at the end of sentences.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import penn_treebank_dataset  # doctest: +SKIP
>>> train = penn_treebank_dataset(train=True)  # doctest: +SKIP
>>> train[:10]  # doctest: +SKIP
['aer', 'banknote', 'berlitz', 'calloway', 'centrust', 'cluett', 'fromstein',
↪'gitano',
'guterman', 'hydro-quebec']
```

torchnlp.datasets.**ud_pos_dataset**(*directory='data/', train=False, dev=False, test=False, train_filename='en-ud-tag.v2.train.txt', dev_filename='en-ud-tag.v2.dev.txt', test_filename='en-ud-tag.v2.test.txt', extracted_name='en-ud-v2', check_files=['en-ud-v2/en-ud-tag.v2.train.txt'], url='https://bitbucket.org/sivareddyg/public/downloads/en-ud-v2.zip'*)

Load the Universal Dependencies - English Dependency Treebank dataset.

Corpus of sentences annotated using Universal Dependencies annotation. The corpus comprises 254,830 words and 16,622 sentences, taken from various web media including weblogs, newsgroups, emails, reviews, and Yahoo! answers.

References

- http://universaldependencies.org/
- https://github.com/UniversalDependencies/UD_English

Citation: Natalia Silveira and Timothy Dozat and Marie-Catherine de Marneffe and Samuel Bowman and Miriam Connor and John Bauer and Christopher D. Manning (2014). A Gold Standard Dependency Corpus for {E}nglish

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import ud_pos_dataset  # doctest: +SKIP
>>> train = ud_pos_dataset(train=True)  # doctest: +SKIP
>>> train[17]  # doctest: +SKIP
{
  'tokens': ['Guerrillas', 'killed', 'an', 'engineer', ',', 'Asi', 'Ali', ',',
↪'from',
             'Tikrit', '.'],
  'ud_tags': ['NOUN', 'VERB', 'DET', 'NOUN', 'PUNCT', 'PROPN', 'PROPN', 'PUNCT',
↪'ADP',
              'PROPN', 'PUNCT'],
  'ptb_tags': ['NNS', 'VBD', 'DT', 'NN', ',', 'NNP', 'NNP', ',', 'IN', 'NNP', '.']
}
```

torchnlp.datasets.**trec_dataset**(*directory='data/trec/', train=False, test=False, train_filename='train_5500.label', test_filename='TREC_10.label', check_files=['train_5500.label'], urls=['http://cogcomp.org/Data/QA/QC/train_5500.label', 'http://cogcomp.org/Data/QA/QC/TREC_10.label'], fine_grained=False*)

Load the Text REtrieval Conference (TREC) Question Classification dataset.

TREC dataset contains 5500 labeled questions in training set and another 500 for test set. The dataset has 6 labels, 50 level-2 labels. Average length of each sentence is 10, vocabulary size of 8700.

References

- https://nlp.stanford.edu/courses/cs224n/2004/may-steinberg-project.pdf
- http://cogcomp.org/Data/QA/QC/
- http://www.aclweb.org/anthology/C02-1150

Citation: Xin Li, Dan Roth, Learning Question Classifiers. COLING'02, Aug., 2002.

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **urls** (*str, optional*) – URLs to download.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import trec_dataset  # doctest: +SKIP
>>> train = trec_dataset(train=True)  # doctest: +SKIP
>>> train[:2]  # doctest: +SKIP
[{
  'label': 'DESC',
  'text': 'How did serfdom develop in and then leave Russia ?'
}, {
  'label': 'ENTY',
  'text': 'What films featured the character Popeye Doyle ?'
}]
```

torchnlp.datasets.**reverse_dataset**(*train=False*, *dev=False*, *test=False*, *train_rows=10000*, *dev_rows=1000*, *test_rows=1000*, *seq_max_length=10*)

Load the Reverse dataset.

The Reverse dataset is a simple task of reversing a list of numbers. This dataset is useful for testing implementations of sequence to sequence models.

Parameters

- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_rows** (*int, optional*) – Number of training rows to generate.
- **dev_rows** (*int, optional*) – Number of development rows to generate.
- **test_rows** (*int, optional*) – Number of test rows to generate.
- **seq_max_length** (*int, optional*) – Maximum sequence length.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.random import set_seed
>>> set_seed(321)
>>>
>>> from torchnlp.datasets import reverse_dataset
>>> train = reverse_dataset(train=True)
>>> train[0:1]
[{'source': '6 2 5 8 7', 'target': '7 8 5 2 6'}]
```

torchnlp.datasets.**count_dataset**(*train=False*, *dev=False*, *test=False*, *train_rows=10000*, *dev_rows=1000*, *test_rows=1000*, *seq_max_length=10*)

Load the Count dataset.

The Count dataset is a simple task of counting the number of integers in a sequence. This dataset is useful for testing implementations of sequence to label models.

Parameters

- **train** (*bool, optional*) – If to load the training split of the dataset.

- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_rows** (*int, optional*) – Number of training rows to generate.
- **dev_rows** (*int, optional*) – Number of development rows to generate.
- **test_rows** (*int, optional*) – Number of test rows to generate.
- **seq_max_length** (*int, optional*) – Maximum sequence length.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.random import set_seed
>>> set_seed(321)
>>>
>>> from torchnlp.datasets import count_dataset
>>> train = count_dataset(train=True)
>>> train[0:2]
[{'numbers': '6 2 5 8 7', 'count': '5'}, {'numbers': '3 9 7 6 6 7', 'count': '6'}]
```

`torchnlp.datasets.`**zero_dataset**(*train=False, dev=False, test=False, train_rows=256, dev_rows=64, test_rows=64*)

Load the Zero dataset.

The Zero dataset is a simple task of predicting zero from zero. This dataset is useful for integration testing. The extreme simplicity of the dataset allows for models to learn the task quickly allowing for quick end-to-end testing.

Parameters

- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_rows** (*int, optional*) – Number of training rows to generate.
- **dev_rows** (*int, optional*) – Number of development rows to generate.
- **test_rows** (*int, optional*) – Number of test rows to generate.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import zero_dataset
>>> train = zero_dataset(train=True)
>>> train[0:2]
[{'source': '0', 'target': '0'}, {'source': '0', 'target': '0'}]
```

torchnlp.datasets.**smt_dataset**(*directory='data/', train=False, dev=False, test=False, train_filename='train.txt', dev_filename='dev.txt', test_filename='test.txt', extracted_name='trees', check_files=['trees/train.txt'], url='http://nlp.stanford.edu/sentiment/trainDevTestTrees_PTB.zip', fine_grained=False, subtrees=False*)

Load the Stanford Sentiment Treebank dataset.

Semantic word spaces have been very useful but cannot express the meaning of longer phrases in a principled way. Further progress towards understanding compositionality in tasks such as sentiment detection requires richer supervised training and evaluation resources and more powerful models of composition. To remedy this, we introduce a Sentiment Treebank. It includes fine grained sentiment labels for 215,154 phrases in the parse trees of 11,855 sentences and presents new challenges for sentiment compositionality.

Reference: https://nlp.stanford.edu/sentiment/index.html

Citation: Richard Socher, Alex Perelygin, Jean Y. Wu, Jason Chuang, Christopher D. Manning, Andrew Y. Ng and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **test** (*bool, optional*) – If to load the test split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **test_filename** (*str, optional*) – The filename of the test split.
- **extracted_name** (*str, optional*) – Name of the extracted dataset directory.
- **check_files** (*str, optional*) – Check if these files exist, then this download was successful.
- **url** (*str, optional*) – URL of the dataset *tar.gz* file.
- **subtrees** (*bool, optional*) – Whether to include sentiment-tagged subphrases in addition to complete examples.
- **fine_grained** (*bool, optional*) – Whether to use 5-class instead of 3-class labeling.

Returns Returns between one and all dataset splits (train, dev and test) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import smt_dataset  # doctest: +SKIP
>>> train = smt_dataset(train=True)  # doctest: +SKIP
>>> train[5]  # doctest: +SKIP
{
  'text': "Whether or not you 're enlightened by any of Derrida 's lectures on ...
↪",
```

(continues on next page)

(continued from previous page)

```
  'label': 'positive'
}
```

torchnlp.datasets.**squad_dataset**(*directory='data/', train=False, dev=False, train_filename='train-v2.0.json', dev_filename='dev-v2.0.json', check_files_train=['train-v2.0.json'], check_files_dev=['dev-v2.0.json'], url_train='https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v2.0.json', url_dev='https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v2.0.json'*)

Load the Stanford Question Answering Dataset (SQuAD) dataset.

Stanford Question Answering Dataset (SQuAD) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable. SQuAD2.0 combines the 100,000 questions in SQuAD1.1 with over 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones. To do well on SQuAD2.0, systems must not only answer questions when possible, but also determine when no answer is supported by the paragraph and abstain from answering.

Reference: https://rajpurkar.github.io/SQuAD-explorer/ **Citation:** Rajpurkar, P., Jia, R. and Liang, P., 2018. Know what you don't know: Unanswerable questions for SQuAD. arXiv preprint arXiv:1806.03822.

Parameters

- **directory** (*str, optional*) – Directory to cache the dataset.
- **train** (*bool, optional*) – If to load the training split of the dataset.
- **dev** (*bool, optional*) – If to load the development split of the dataset.
- **train_filename** (*str, optional*) – The filename of the training split.
- **dev_filename** (*str, optional*) – The filename of the development split.
- **check_files_train** (*list, optional*) – All train filenames
- **check_files_dev** (*list, optional*) – All development filenames
- **url_train** (*str, optional*) – URL of the train dataset *.json* file.
- **url_dev** (*str, optional*) – URL of the dev dataset *.json* file.

Returns Returns between one and all dataset splits (train and dev) depending on if their respective boolean argument is `True`.

Return type `tuple` of `iterable` or `iterable`

Example

```
>>> from torchnlp.datasets import squad_dataset  # doctest: +SKIP
>>> train = squad_dataset(train=True)  # doctest: +SKIP
>>> train[0]['paragraphs'][0]['qas'][0]['question']  # doctest: +SKIP
'When did Beyonce start becoming popular?'
>>> train[0]['paragraphs'][0]['qas'][0]['answers'][0]  # doctest: +SKIP
{'text': 'in the late 1990s', 'answer_start': 269}
```

CHAPTER 2

torchnlp.download package

The `torchnlp.download` package contains modules useful for download and extracting datasets.

`torchnlp.download.`**`download_file_maybe_extract`**(*url*, *directory*, *filename=None*, *extension=None*, *check_files=[]*)

Download the file at `url` to `directory`. Extract to `directory` if tar or zip.

Parameters

- **`url`** (*`str or Path`*) – Url of file.
- **`directory`** (*`str`*) – Directory to download to.
- **`filename`** (*`str, optional`*) – Name of the file to download; Otherwise, a filename is extracted from the url.
- **`extension`** (*`str, optional`*) – Extension of the file; Otherwise, attempts to extract extension from the filename.
- **`check_files`** (*`list of str or Path`*) – Check if these files exist, ensuring the download succeeded. If these files exist before the download, the download is skipped.

Returns Filename of download file.

Return type (str)

Raises `ValueError` – Error if one of the `check_files` are not found following the download.

`torchnlp.download.`**`download_files_maybe_extract`**(*urls*, *directory*, *check_files=[]*)

Download the files at `urls` to `directory`. Extract to `directory` if tar or zip.

Parameters

- **`urls`** (*`str`*) – Url of files.
- **`directory`** (*`str`*) – Directory to download to.
- **`check_files`** (*`list of str`*) – Check if these files exist, ensuring the download succeeded. If these files exist before the download, the download is skipped.

Raises `ValueError` – Error if one of the `check_files` are not found following the download.

CHAPTER 3

torchnlp.encoders package

The `torchnlp.encoders` package supports encoding objects as a vector `torch.Tensor` and decoding a vector `torch.Tensor` back.

class `torchnlp.encoders.`**Encoder**(*enforce_reversible=False*)

Bases: `object`

Base class for a encoder employing an identity function.

Parameters **enforce_reversible** (*bool, optional*) – Check for reversibility on `Encoder.encode` and `Encoder.decode`. Formally, reversible means: `Encoder.decode(Encoder.encode(object_)) == object_`.

batch_decode(*iterator*, **args*, ***kwargs*)

Parameters

- **iterator** (*list*) – Batch of encoded objects.
- ***args** – Arguments passed to `decode`.
- ****kwargs** – Keyword arguments passed to `decode`.

Returns Batch of decoded objects.

Return type list

batch_encode(*iterator*, **args*, ***kwargs*)

Parameters

- **batch** (*list*) – Batch of objects to encode.
- ***args** – Arguments passed to `encode`.
- ****kwargs** – Keyword arguments passed to `encode`.

Returns Batch of encoded objects.

Return type list

decode (*encoded*)
Decodes an object.

Parameters **object** (*object*) – Encoded object.

Returns Object decoded.

Return type object

encode (*object_*)
Encodes an object.

Parameters **object** (*object*) – Object to encode.

Returns Encoding of the object.

Return type object

class torchnlp.encoders.**LabelEncoder**(*sample*, *min_occurrences=1*, *reserved_labels=['<unk>']*, *unknown_index=0*, ***kwargs*)
Bases: torchnlp.encoders.encoder.Encoder

Encodes an label via a dictionary.

Parameters

- **sample** (*list of strings*) – Sample of data used to build encoding dictionary.
- **min_occurrences** (*int, optional*) – Minimum number of occurrences for a label to be added to the encoding dictionary.
- **reserved_labels** (*list, optional*) – List of reserved labels inserted in the beginning of the dictionary.
- **unknown_index** (*int, optional*) – The unknown label is used to encode unseen labels. This is the index that label resides at.
- ****kwargs** – Keyword arguments passed onto Encoder.

Example

```
>>> samples = ['label_a', 'label_b']
>>> encoder = LabelEncoder(samples, reserved_labels=['unknown'], unknown_index=0)
>>> encoder.encode('label_a')
tensor(1)
>>> encoder.decode(encoder.encode('label_a'))
'label_a'
>>> encoder.encode('label_c')
tensor(0)
>>> encoder.decode(encoder.encode('label_c'))
'unknown'
>>> encoder.vocab
['unknown', 'label_a', 'label_b']
```

batch_decode (*tensor*, **args*, *dim=0*, ***kwargs*)

Parameters

- **tensor** (*torch.Tensor*) – Batch of tensors.
- ***args** – Arguments passed to Encoder.batch_decode.
- **dim** (*int, optional*) – Dimension along which to split tensors.

- ****kwargs** – Keyword arguments passed to `Encoder.batch_decode`.

Returns Batch of decoded labels.

Return type list

batch_encode(*iterator*, **args*, *dim=0*, ***kwargs*)

Parameters

- **iterator** (*iterator*) – Batch of labels to encode.
- ***args** – Arguments passed to `Encoder.batch_encode`.
- **dim** (*int, optional*) – Dimension along which to concatenate tensors.
- ****kwargs** – Keyword arguments passed to `Encoder.batch_encode`.

Returns Tensor of encoded labels.

Return type torch.Tensor

decode(*encoded*)

Decodes `encoded` label.

Parameters **encoded** (*torch.Tensor*) – Encoded label.

Returns Label decoded from `encoded`.

Return type object

encode(*label*)

Encodes a `label`.

Parameters **label** (*object*) – Label to encode.

Returns Encoding of the label.

Return type torch.Tensor

vocab

List of labels in the dictionary.

Type Returns

Type list

vocab_size

Number of labels in the dictionary.

Type Returns

Type int

class `torchnlp.encoders.text.`**CharacterEncoder**(**args*, ***kwargs*)

Bases: `torchnlp.encoders.text.static_tokenizer_encoder.StaticTokenizerEncoder`

Encodes text into a tensor by splitting the text into individual characters.

Parameters

- ****args** – Arguments passed onto `StaticTokenizerEncoder.__init__`.
- ****kwargs** – Keyword arguments passed onto `StaticTokenizerEncoder.__init__`.

class torchnlp.encoders.text.**DelimiterEncoder**(*delimiter*, **args*, ***kwargs*)

Bases: torchnlp.encoders.text.static_tokenizer_encoder.StaticTokenizerEncoder

Encodes text into a tensor by splitting the text using a delimiter.

Parameters

- **delimiter** (*string*) – Delimiter used with `string.split`.
- ****args** – Arguments passed onto `StaticTokenizerEncoder.__init__`.
- ****kwargs** – Keyword arguments passed onto `StaticTokenizerEncoder.__init__`.

Example

```
>>> encoder = DelimiterEncoder('|', ['token_a|token_b', 'token_c'])
>>> encoder.encode('token_a|token_c')
tensor([5, 7])
>>> encoder.vocab
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'token_a', 'token_b', 'token_c']
```

class torchnlp.encoders.text.**MosesEncoder**(**args*, ***kwargs*)

Bases: torchnlp.encoders.text.static_tokenizer_encoder.StaticTokenizerEncoder

Encodes the text using the Moses tokenizer.

Tokenizer Reference: http://www.nltk.org/_modules/nltk/tokenize/moses.html

Parameters

- ****args** – Arguments passed onto `StaticTokenizerEncoder.__init__`.
- ****kwargs** – Keyword arguments passed onto `StaticTokenizerEncoder.__init__`.

NOTE: The *doctest* is skipped because running NLTK moses with Python 3.7's pytest halts on travis.

Example

```
>>> encoder = MosesEncoder(["This ain't funny.", "Don't?"]) # doctest: +SKIP
>>> encoder.encode("This ain't funny.") # doctest: +SKIP
tensor([5, 6, 7, 8, 9])
>>> encoder.vocab # doctest: +SKIP
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'This', 'ain', '&apos;t', 'funny', '.
↪', 'Don', '?']
>>> encoder.decode(encoder.encode("This ain't funny.")) # doctest: +SKIP
"This ain't funny."
```

torchnlp.encoders.text.**pad_tensor**(*tensor*, *length*, *padding_index=0*)

Pad a `tensor` to `length` with `padding_index`.

Parameters

- **tensor** (*torch.Tensor [n, ..]*) – Tensor to pad.
- **length** (*int*) – Pad the `tensor` up to `length`.
- **padding_index** (*int, optional*) – Index to pad tensor with.

Returns (torch.Tensor [length, ...]) Padded Tensor.

torchnlp.encoders.text.**stack_and_pad_tensors**(*batch*, *padding_index=0*, *dim=0*)
Pad a `list` of `tensors` (`batch`) with `padding_index`.

Parameters

- **batch** (`list` of `torch.Tensor`) – Batch of tensors to pad.
- **padding_index** (*int, optional*) – Index to pad tensors with.
- **dim** (*int, optional*) – Dimension on to which to concatenate the batch of tensors.

Returns SequenceBatch: Padded tensors and original lengths of tensors.

class torchnlp.encoders.text.**TextEncoder**(*enforce_reversible=False*)
Bases: `torchnlp.encoders.encoder.Encoder`

batch_decode(*tensor*, *lengths*, *dim=0*, **args*, ***kwargs*)

Parameters

- **batch** (list of `torch.Tensor`) – Batch of encoded sequences.
- **lengths** (*torch.Tensor*) – Original lengths of sequences.
- **dim** (*int, optional*) – Dimension along which to split tensors.
- ***args** – Arguments passed to `decode`.
- ****kwargs** – Key word arguments passed to `decode`.

Returns Batch of decoded sequences.

Return type list

batch_encode(*iterator*, **args*, *dim=0*, ***kwargs*)

Parameters

- **iterator** (*iterator*) – Batch of text to encode.
- ***args** – Arguments passed onto `Encoder.__init__`.
- **dim** (*int, optional*) – Dimension along which to concatenate tensors.
- ****kwargs** – Keyword arguments passed onto `Encoder.__init__`.

Returns

torch.Tensor, torch.Tensor: Encoded and padded batch of sequences; Original lengths of sequences.

decode(*encoded*)
Decodes an object.

Parameters **object** (*object*) – Encoded object.

Returns Object decoded.

Return type object

class torchnlp.encoders.text.**SpacyEncoder**(**args*, ***kwargs*)
Bases: `torchnlp.encoders.text.static_tokenizer_encoder.StaticTokenizerEncoder`

Encodes the text using spaCy's tokenizer.

Tokenizer Reference: https://spacy.io/api/tokenizer

Parameters

- ****args** – Arguments passed onto `StaticTokenizerEncoder.__init__`.
- **language** (*string, optional*) – Language to use for parsing. Accepted values are 'en', 'de', 'es', 'pt', 'fr', 'it', 'nl' and 'xx'. For details see https://spacy.io/models/#available-models
- ****kwargs** – Keyword arguments passed onto `StaticTokenizerEncoder.__init__`.

Example

```
>>> encoder = SpacyEncoder(["This ain't funny.", "Don't?"])
>>> encoder.encode("This ain't funny.")
tensor([5, 6, 7, 8, 9])
>>> encoder.vocab
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'This', 'ai', "n't", 'funny', '.', 'Do
↪', '?']
>>> encoder.decode(encoder.encode("This ain't funny."))
"This ai n't funny ."
```

batch_encode(*sequences*)

Parameters

- **iterator** (*iterator*) – Batch of text to encode.
- ***args** – Arguments passed onto `Encoder.__init__`.
- **dim** (*int, optional*) – Dimension along which to concatenate tensors.
- ****kwargs** – Keyword arguments passed onto `Encoder.__init__`.

Returns

torch.Tensor, torch.Tensor: Encoded and padded batch of sequences; Original lengths of sequences.

class torchnlp.encoders.text.**StaticTokenizerEncoder**(*sample, min_occurrences=1, append_sos=False, append_eos=False, tokenize=<function _tokenize>, detokenize=<function _detokenize>, reserved_tokens=['<pad>', '<unk>', '</s>', '<s>', '<copy>'], sos_index=3, eos_index=2, unknown_index=1, padding_index=0, **kwargs*)

Bases: `torchnlp.encoders.text.text_encoder.TextEncoder`

Encodes a text sequence using a static tokenizer.

Parameters

- **sample** (*collections.abc.Iterable*) – Sample of data used to build encoding dictionary.
- **min_occurrences** (*int, optional*) – Minimum number of occurrences for a token to be added to the encoding dictionary.
- **tokenize** (*callable*) – `callable` to tokenize a sequence.
- **detokenize** (*callable*) – `callable` to detokenize a sequence.
- **append_sos** (*bool, optional*) – If `True` insert SOS token at the start of the encoded vector.
- **append_eos** (*bool, optional*) – If `True` append EOS token onto the end to the encoded vector.
- **reserved_tokens** (*list of str, optional*) – List of reserved tokens inserted in the beginning of the dictionary.
- **sos_index** (*int, optional*) – The sos token is used to encode the start of a sequence. This is the index that token resides at.
- **eos_index** (*int, optional*) – The eos token is used to encode the end of a sequence. This is the index that token resides at.
- **unknown_index** (*int, optional*) – The unknown token is used to encode unseen tokens. This is the index that token resides at.
- **padding_index** (*int, optional*) – The unknown token is used to encode sequence padding. This is the index that token resides at.
- ****kwargs** – Keyword arguments passed onto `TextEncoder.__init__`.

Example

```
>>> sample = ["This ain't funny.", "Don't?"]
>>> encoder = StaticTokenizerEncoder(sample, tokenize=lambda s: s.split())
>>> encoder.encode("This ain't funny.")
tensor([5, 6, 7])
>>> encoder.vocab
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'This', "ain't", 'funny.', "Don't?"]
>>> encoder.decode(encoder.encode("This ain't funny."))
"This ain't funny."
```

decode(*encoded*)

Decodes a tensor into a sequence.

Parameters **encoded** (*torch.Tensor*) – Encoded sequence.

Returns Sequence decoded from `encoded`.

Return type str

encode(*sequence*)

Encodes a `sequence`.

Parameters **sequence** (*str*) – String `sequence` to encode.

Returns Encoding of the `sequence`.

Return type torch.Tensor

vocab
List of tokens in the dictionary.

Type Returns

Type list

vocab_size
Number of tokens in the dictionary.

Type Returns

Type int

class torchnlp.encoders.text.**SubwordEncoder**(*sample, append_sos=False, append_eos=False, target_vocab_size=None, min_occurrences=1, max_occurrences=1000.0, reserved_tokens=['<pad>', '<unk>', '</s>', '<s>', '<copy>'], sos_index=3, eos_index=2, unknown_index=1, padding_index=0, **kwargs*)

Bases: torchnlp.encoders.text.text_encoder.TextEncoder

Invertibly encoding text using a limited vocabulary.

Applies Googles Tensor2Tensor SubwordTextTokenizer that invertibly encodes a native string as a sequence of subtokens from a limited vocabulary. In order to build the vocabulary, it uses recursive binary search to find a minimum token count *x* (s.t. *min_occurrences* <= *x* <= *max_occurrences*) that most closely matches the *target_size*.

Tokenizer Reference: https://github.com/tensorflow/tensor2tensor/blob/8bdecbe434d93cb1e79c0489df20fee2d5a37dc2/tensor2tensor/data_generators/text_encoder.py#L389

Parameters

- **sample** (*list*) – Sample of data used to build encoding dictionary.
- **append_sos** (*bool, optional*) – If True insert SOS token at the start of the encoded vector.
- **append_eos** (*bool, optional*) – If True append EOS token onto the end to the encoded vector.
- **target_vocab_size** (*int, optional*) – Desired size of vocab.
- **min_occurrences** (*int, optional*) – Lower bound for the minimum token count.
- **max_occurrences** (*int, optional*) – Upper bound for the minimum token count.
- **reserved_tokens** (*list of str, optional*) – List of reserved tokens inserted in the beginning of the dictionary.
- **sos_index** (*int, optional*) – The sos token is used to encode the start of a sequence. This is the index that token resides at.
- **eos_index** (*int, optional*) – The eos token is used to encode the end of a sequence. This is the index that token resides at.
- **unknown_index** (*int, optional*) – The unknown token is used to encode unseen tokens. This is the index that token resides at.
- **padding_index** (*int, optional*) – The padding token is used to encode sequence padding. This is the index that token resides at.

- ****kwargs** – Keyword arguments passed onto `TextEncoder.__init__`.

decode (*encoded*)
Decodes a tensor into a sequence.

Parameters **encoded** (*torch.Tensor*) – Encoded sequence.

Returns Sequence decoded from `encoded`.

Return type str

encode (*sequence*)
Encodes a `sequence`.

Parameters **sequence** (*str*) – String `sequence` to encode.

Returns Encoding of the `sequence`.

Return type torch.Tensor

vocab
List of tokens in the dictionary.

Type Returns

Type list

vocab_size
Number of tokens in the dictionary.

Type Returns

Type int

class `torchnlp.encoders.text.`**TreebankEncoder** (**args*, ***kwargs*)
Bases: `torchnlp.encoders.text.static_tokenizer_encoder.StaticTokenizerEncoder`

Encodes text using the Treebank tokenizer.

Tokenizer Reference: http://www.nltk.org/_modules/nltk/tokenize/treebank.html

Parameters

- ****args** – Arguments passed onto `StaticTokenizerEncoder.__init__`.
- ****kwargs** – Keyword arguments passed onto `StaticTokenizerEncoder.__init__`.

Example

```
>>> encoder = TreebankEncoder(["This ain't funny.", "Don't?"])
>>> encoder.encode("This ain't funny.")
tensor([5, 6, 7, 8, 9])
>>> encoder.vocab
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'This', 'ai', "n't", 'funny', '.', 'Do
↪', '?']
>>> encoder.decode(encoder.encode("This ain't funny."))
"This ain't funny."
```

class `torchnlp.encoders.text.`**WhitespaceEncoder** (**args*, ***kwargs*)
Bases: `torchnlp.encoders.text.delimiter_encoder.DelimiterEncoder`

Encodes a text by splitting on whitespace.

Parameters

- ****args** – Arguments passed onto `DelimiterEncoder.__init__`.
- ****kwargs** – Keyword arguments passed onto `DelimiterEncoder.__init__`.

Example

```
>>> encoder = WhitespaceEncoder(["This ain't funny.", "Don't?"])
>>> encoder.encode("This ain't funny.")
tensor([5, 6, 7])
>>> encoder.vocab
['<pad>', '<unk>', '</s>', '<s>', '<copy>', 'This', "ain't", 'funny.', "Don't?"]
>>> encoder.decode(encoder.encode("This ain't funny."))
"This ain't funny."
```

class `torchnlp.encoders.text.`**SequenceBatch**(*tensor*, *lengths*)

Bases: `tuple`

lengths

Alias for field number 1

tensor

Alias for field number 0

CHAPTER 4

torchnlp.metrics package

The `torchnlp.metrics` package introduces a set of modules able to compute common NLP metrics.

`torchnlp.metrics.`**`get_accuracy`**(*targets*, *outputs*, *k=1*, *ignore_index=None*)

Get the accuracy top-k accuracy between two tensors.

Parameters

- **`targets`** (1 - 2D `torch.Tensor`) – Target or true vector against which to measure saccuracy
- **`outputs`** (1 - 3D `torch.Tensor`) – Prediction or output vector
- **`ignore_index`** (*`int, optional`*) – Specifies a target index that is ignored

Returns `tuple` consisting of accuracy (`float`), number correct (`int`) and total (`int`)

Example

```
>>> import torch
>>> from torchnlp.metrics import get_accuracy
>>> targets = torch.LongTensor([1, 2, 3, 4, 5])
>>> outputs = torch.LongTensor([1, 2, 2, 3, 5])
>>> accuracy, n_correct, n_total = get_accuracy(targets, outputs, ignore_index=3)
>>> accuracy
0.8
>>> n_correct
4
>>> n_total
5
```

`torchnlp.metrics.`**`get_token_accuracy`**(*targets*, *outputs*, *ignore_index=None*)

Get the accuracy token accuracy between two tensors.

Parameters

- **targets** (1 - 2D `torch.Tensor`) – Target or true vector against which to measure saccuracy
- **outputs** (1 - 3D `torch.Tensor`) – Prediction or output vector
- **ignore_index** (*int, optional*) – Specifies a target index that is ignored

Returns `tuple` consisting of accuracy (`float`), number correct (`int`) and total (`int`)

Example

```
>>> import torch
>>> from torchnlp.metrics import get_token_accuracy
>>> targets = torch.LongTensor([[1, 1], [2, 2], [3, 3]])
>>> outputs = torch.LongTensor([[1, 1], [2, 3], [4, 4]])
>>> accuracy, n_correct, n_total = get_token_accuracy(targets, outputs, ignore_
↪index=3)
>>> accuracy
0.75
>>> n_correct
3.0
>>> n_total
4.0
```

`torchnlp.metrics.`**get_moses_multi_bleu**(*hypotheses*, *references*, *lowercase=False*)

Get the BLEU score using the moses *multi-bleu.perl* script.

Script: https://raw.githubusercontent.com/moses-smt/mosesdecoder/master/scripts/generic/multi-bleu.perl

Parameters

- **hypotheses** (*list of str*) – List of predicted values
- **references** (*list of str*) – List of target values
- **lowercase** (*bool*) – If true, pass the "-lc" flag to the *multi-bleu.perl* script

Returns (`np.float32`) The BLEU score as a float32 value.

Example

```
>>> hypotheses = [
...   "The brown fox jumps over the dog ",
...   "The brown fox jumps over the dog 2 "
... ]
>>> references = [
...   "The quick brown fox jumps over the lazy dog ",
...   "The quick brown fox jumps over the lazy dog "
... ]
>>> get_moses_multi_bleu(hypotheses, references, lowercase=True)
46.51
```

CHAPTER 5

torchnlp.nn package

The neural network nn package `torchnlp.nn` introduces a set of `torch.nn.Module` commonly used in NLP.

class `torchnlp.nn.`**LockedDropout**(*p=0.5*)

LockedDropout applies the same dropout mask to every time step.

Thank you to Sales Force for their initial implementation of *WeightDrop*. Here is their License.

Parameters **p** (*float*) – Probability of an element in the dropout mask to be zeroed.

forward(*x*)

Parameters **x** (`torch.FloatTensor` [sequence length, batch size, rnn hidden size]) – Input to apply dropout too.

class `torchnlp.nn.`**Attention**(*dimensions*, *attention_type='general'*)

Applies attention mechanism on the *context* using the *query*.

Thank you to IBM for their initial implementation of *Attention*. Here is their License.

Parameters

- **dimensions** (*int*) – Dimensionality of the query and context.
- **attention_type** (*str, optional*) – How to compute the attention score:
  - dot: $score(H_j, q) = H_j^T q$
  - general: $score(H_j, q) = H_j^T W_a q$

Example

```
>>> attention = Attention(256)
>>> query = torch.randn(5, 1, 256)
>>> context = torch.randn(5, 5, 256)
>>> output, weights = attention(query, context)
>>> output.size()
torch.Size([5, 1, 256])
```

(continues on next page)

(continued from previous page)

```
>>> weights.size()
torch.Size([5, 1, 5])
```

forward(*query*, *context*)

Parameters

- **query** (torch.FloatTensor [batch size, output length, dimensions]) – Sequence of queries to query the context.
- **context** (torch.FloatTensor [batch size, query length, dimensions]) – Data over-which to apply the attention mechanism.

Returns

- **output** (torch.LongTensor [batch size, output length, dimensions]): Tensor containing the attended features.
- **weights** (torch.FloatTensor [batch size, output length, query length]): Tensor containing attention weights.

Return type tuple with *output* and *weights*

class torchnlp.nn.**CNNEncoder**(*embedding_dim*, *num_filters*, *ngram_filter_sizes=(2, 3, 4, 5)*, *conv_layer_activation=ReLU()*, *output_dim=None*)

A combination of multiple convolution layers and max pooling layers.

The CNN has one convolution layer for each ngram filter size. Each convolution operation gives out a vector of size num_filters. The number of times a convolution layer will be used is `num_tokens - ngram_size + 1`. The corresponding maxpooling layer aggregates all these outputs from the convolution layer and outputs the max.

This operation is repeated for every ngram size passed, and consequently the dimensionality of the output after maxpooling is `len(ngram_filter_sizes) * num_filters`. This then gets (optionally) projected down to a lower dimensional output, specified by `output_dim`.

We then use a fully connected layer to project in back to the desired output_dim. For more details, refer to "A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification", Zhang and Wallace 2016, particularly Figure 1.

Thank you to AI2 for their initial implementation of *CNNEncoder*. Here is their License.

Parameters

- **embedding_dim** (*int*) – This is the input dimension to the encoder. We need this because we can't do shape inference in pytorch, and we need to know what size filters to construct in the CNN.
- **num_filters** (*int*) – This is the output dim for each convolutional layer, which is the number of "filters" learned by that layer.
- **ngram_filter_sizes** (tuple of int, optional) – This specifies both the number of convolutional layers we will create and their sizes. The default of `(2, 3, 4, 5)` will have four convolutional layers, corresponding to encoding ngrams of size 2 to 5 with some number of filters.
- **conv_layer_activation** (*torch.nn.Module, optional*) – Activation to use after the convolution layers.
- **output_dim** (*int or None, optional*) – After doing convolutions and pooling, we'll project the collected features into a vector of this size. If this value is

None, we will just return the result of the max pooling, giving an output of shape len(ngram_filter_sizes) * num_filters.

forward(*tokens*, *mask=None*)

Parameters

- **tokens** (torch.FloatTensor [batch_size, num_tokens, input_dim]) – Sequence matrix to encode.
- **mask** (torch.FloatTensor) – Broadcastable matrix to *tokens* used as a mask.

Returns Encoding of sequence.

Return type (torch.FloatTensor [batch_size, output_dim])

get_input_dim()

get_output_dim()

class torchnlp.nn.**WeightDrop**(*module*, *weights*, *dropout=0.0*)

The weight-dropped module applies recurrent regularization through a DropConnect mask on the hidden-to-hidden recurrent weights.

Thank you to Sales Force for their initial implementation of *WeightDrop*. Here is their License.

Parameters

- **module** (torch.nn.Module) – Containing module.
- **weights** (list of str) – Names of the module weight parameters to apply a dropout too.
- **dropout** (*float*) – The probability a weight will be dropped.

Example

```
>>> from torchnlp.nn import WeightDrop
>>> import torch
>>>
>>> torch.manual_seed(123)
<torch._C.Generator object ...
>>>
>>> gru = torch.nn.GRUCell(2, 2)
>>> weights = ['weight_hh']
>>> weight_drop_gru = WeightDrop(gru, weights, dropout=0.9)
>>>
>>> input_ = torch.randn(3, 2)
>>> hidden_state = torch.randn(3, 2)
>>> weight_drop_gru(input_, hidden_state)
tensor(... grad_fn=<AddBackward0>)
```

class torchnlp.nn.**WeightDropGRU**(**args*, *weight_dropout=0.0*, ***kwargs*)

Wrapper around torch.nn.GRU that adds weight_dropout named argument.

Parameters **weight_dropout** (*float*) – The probability a weight will be dropped.

class torchnlp.nn.**WeightDropLSTM**(**args*, *weight_dropout=0.0*, ***kwargs*)

Wrapper around torch.nn.LSTM that adds weight_dropout named argument.

Parameters **weight_dropout** (*float*) – The probability a weight will be dropped.

class torchnlp.nn.**WeightDropLinear**(**args*, *weight_dropout=0.0*, ***kwargs*)
Wrapper around `torch.nn.Linear` that adds `weight_dropout` named argument.

Parameters **weight_dropout** (*float*) – The probability a weight will be dropped.

CHAPTER 6

torchnlp.random package

The `torchnlp.random` package introduces modules for finer grain control of random state.

class torchnlp.random.**RandomGeneratorState**(*random*, *torch*, *numpy*, *torch_cuda*)

numpy
Alias for field number 2

random
Alias for field number 0

torch
Alias for field number 1

torch_cuda
Alias for field number 3

torchnlp.random.**fork_rng**(*seed=None*, *cuda=False*)
Forks the *torch*, *numpy* and *random* random generators, so that when you return, the random generators are reset to the state that they were previously in.

Parameters

- **seed** (*int or None, optional*) – If defined this sets the seed values for the random generator fork. This is a convenience parameter.
- **cuda** (*bool, optional*) – If *True* saves the *cuda* seed also. Getting and setting the random generator state can be quite slow if you have a lot of GPUs.

torchnlp.random.**fork_rng_wrap**(*function=None*, ***kwargs*)
Decorator alias for *fork_rng*.

torchnlp.random.**get_random_generator_state**(*cuda: bool = False*) → torchnlp.random.RandomGeneratorState
Get the *torch*, *numpy* and *random* random generator state.

Parameters **cuda** (*bool, optional*) – If *True* saves the *cuda* seed also. Note that getting and setting the random generator state for CUDA can be quite slow if you have a lot of GPUs.

Returns RandomGeneratorState

torchnlp.random.**set_random_generator_state**(*state: torchnlp.random.RandomGeneratorState*)
Set the *torch*, *numpy* and *random* random generator state.

Parameters **state** (RandomGeneratorState) –

torchnlp.random.**set_seed**(*seed*, *cuda=False*)
Set seed values for random generators.

Parameters

- **seed** (*int*) – Value used as a seed.
- **cuda** (*bool, optional*) – If *True* sets the *cuda* seed also.

CHAPTER 7

torchnlp.samplers package

The `torchnlp.samplers` package introduces a set of samplers. Samplers sample elements from a dataset. `torchnlp.samplers` plug into `torch.utils.data.distributed.DistributedSampler` and `torch.utils.data.DataLoader`.

class `torchnlp.samplers.`**BalancedSampler**(*data_source*, *get_class=<function identity>*, *get_weight=<function BalancedSampler.<lambda>>*, ***kwargs*)

Weighted sampler with respect for an element's class.

Parameters

- **data** (*iterable*) –
- **get_class** (*callable, optional*) – Get the class of an item relative to the entire dataset.
- **get_weight** (*callable, optional*) – Define a weight for each item other than one.
- **kwargs** – Additional key word arguments passed onto *WeightedRandomSampler*.

Example

```
>>> from torchnlp.samplers import DeterministicSampler
>>>
>>> data = ['a', 'b', 'c'] + ['c'] * 100
>>> sampler = BalancedSampler(data, num_samples=3)
>>> sampler = DeterministicSampler(sampler, random_seed=12)
>>> [data[i] for i in sampler]
['c', 'b', 'a']
```

class `torchnlp.samplers.`**BPTTBatchSampler**(*data*, *bptt_length*, *batch_size*, *drop_last*, *type_='source'*)

Samples sequentially a batch of source and target slices of size `bptt_length`.

Typically, such a sampler, is used for language modeling training with backpropagation through time (BPTT).

Reference: https://github.com/pytorch/examples/blob/c66593f1699ece14a4a2f4d314f1afb03c6793d9/word_language_model/main.py#L61

Parameters

- **data** (*iterable*) –
- **bptt_length** (*int*) – Length of the slice.
- **batch_size** (*int*) – Size of mini-batch.
- **drop_last** (*bool*) – If `True`, the sampler will drop the last batch if its size would be less than `batch_size`.
- **type** (*str, optional*) – Type of batch ['source'|'target'] to load where a target batch is one timestep ahead.

Example

```
>>> sampler = BPTTBatchSampler(range(100), bptt_length=2, batch_size=3, drop_
↪last=False)
>>> list(sampler)[0] # First Batch
[slice(0, 2, None), slice(34, 36, None), slice(67, 69, None)]
```

class torchnlp.samplers.**BPTTSampler**(*data*, *bptt_length*, *type_='source'*)

Samples sequentially source and target slices of size `bptt_length`.

Typically, such a sampler, is used for language modeling training with backpropagation through time (BPTT).

Reference: https://github.com/pytorch/examples/blob/c66593f1699ece14a4a2f4d314f1afb03c6793d9/word_language_model/main.py#L122

Parameters

- **data** (*iterable*) – Iterable data.
- **bptt_length** (*int*) – Length of the slice.
- **type** (*str, optional*) – Type of slice ['source'|'target'] to load where a target slice is one timestep ahead

Example

```
>>> from torchnlp.samplers import BPTTSampler
>>> list(BPTTSampler(range(5), 2))
[slice(0, 2, None), slice(2, 4, None)]
```

class torchnlp.samplers.**BucketBatchSampler**(*sampler*, *batch_size*, *drop_last*, *sort_key=<function identity>*, *bucket_size_multiplier=100*)

BucketBatchSampler toggles between *sampler* batches and sorted batches.

Typically, the *sampler* will be a *RandomSampler* allowing the user to toggle between random batches and sorted batches. A larger *bucket_size_multiplier* is more sorted and vice versa.

Background: `BucketBatchSampler` is similar to a `BucketIterator` found in popular libraries like `AllenNLP` and `torchtext`. A `BucketIterator` pools together examples with a similar size length to reduce the padding required for each batch while maintaining some noise through bucketing.

AllenNLP Implementation: https://github.com/allenai/allennlp/blob/master/allennlp/data/iterators/bucket_iterator.py

torchtext Implementation: https://github.com/pytorch/text/blob/master/torchtext/data/iterator.py#L225

Parameters

- **sampler** (*torch.data.utils.sampler.Sampler*) –
- **batch_size** (*int*) – Size of mini-batch.
- **drop_last** (*bool*) – If *True* the sampler will drop the last batch if its size would be less than *batch_size*.
- **sort_key** (*callable, optional*) – Callable to specify a comparison key for sorting.
- **bucket_size_multiplier** (*int, optional*) – Buckets are of size *batch_size* * *bucket_size_multiplier*.

Example

```
>>> from torchnlp.random import set_seed
>>> set_seed(123)
>>>
>>> from torch.utils.data.sampler import SequentialSampler
>>> sampler = SequentialSampler(list(range(10)))
>>> list(BucketBatchSampler(sampler, batch_size=3, drop_last=False))
[[6, 7, 8], [0, 1, 2], [3, 4, 5], [9]]
>>> list(BucketBatchSampler(sampler, batch_size=3, drop_last=True))
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

class torchnlp.samplers.**DeterministicSampler**(*sampler*, *random_seed*, *cuda=False*)

Maintains a random state such that *sampler* returns the same output every process.

Parameters

- **sampler** (*torch.data.utils.sampler.Sampler*) –
- **random_seed** (*int*) –
- **cuda** (*bool, optional*) – If *True* this sampler forks the random state of CUDA as well.

class torchnlp.samplers.**DistributedBatchSampler**(*batch_sampler*, ***kwargs*)

BatchSampler wrapper that distributes across each batch multiple workers.

Parameters

- **batch_sampler** (*torch.utils.data.sampler.BatchSampler*) –
- **num_replicas** (*int, optional*) – Number of processes participating in distributed training.
- **rank** (*int, optional*) – Rank of the current process within num_replicas.

Example

```
>>> from torch.utils.data.sampler import BatchSampler
>>> from torch.utils.data.sampler import SequentialSampler
>>> sampler = SequentialSampler(list(range(12)))
>>> batch_sampler = BatchSampler(sampler, batch_size=4, drop_last=False)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> list(DistributedBatchSampler(batch_sampler, num_replicas=2, rank=0))
[[0, 2], [4, 6], [8, 10]]
>>> list(DistributedBatchSampler(batch_sampler, num_replicas=2, rank=1))
[[1, 3], [5, 7], [9, 11]]
```

class torchnlp.samplers.**DistributedSampler**(*iterable*, *num_replicas=None*, *rank=None*)

Iterable wrapper that distributes data across multiple workers.

Parameters

- **iterable** (*iterable*) –
- **num_replicas** (*int, optional*) – Number of processes participating in distributed training.
- **rank** (*int, optional*) – Rank of the current process within num_replicas.

Example

```
>>> list(DistributedSampler(range(10), num_replicas=2, rank=0))
[0, 2, 4, 6, 8]
>>> list(DistributedSampler(range(10), num_replicas=2, rank=1))
[1, 3, 5, 7, 9]
```

torchnlp.samplers.**get_number_of_elements**(*object_*)

Get the sum of the number of elements in all tensors stored in *object_*.

This is particularly useful for sampling the largest objects based on tensor size like in: *OomBatchSampler.__init__.get_item_size*.

Parameters **object** (*any*) –

Returns The number of elements in the *object_*.

Return type (int)

class torchnlp.samplers.**NoisySortedSampler**(*data*, *sort_key=<function identity>*, *get_noise=<function _uniform_noise>*)

Samples elements sequentially with noise.

Background

NoisySortedSampler is similar to a BucketIterator found in popular libraries like *AllenNLP* and *torchtext*. A BucketIterator pools together examples with a similar size length to reduce the padding required for each batch. BucketIterator also includes the ability to add noise to the pooling.

AllenNLP Implementation: https://github.com/allenai/allennlp/blob/e125a490b71b21e914af01e70e9b00b165d64dcd/allennlp/data/iterators/bucket_iterator.py

torchtext Implementation: https://github.com/pytorch/text/blob/master/torchtext/data/iterator.py#L225

Parameters

- **data** (*iterable*) – Data to sample from.
- **sort_key** (*callable*) – Specifies a function of one argument that is used to extract a numerical comparison key from each list element.

- **get_noise** (*callable*) – Noise added to each numerical `sort_key`.

Example

```
>>> from torchnlp.random import set_seed
>>> set_seed(123)
>>>
>>> import random
>>> get_noise = lambda i: round(random.uniform(-1, 1))
>>> list(NoisySortedSampler(range(10), sort_key=lambda i: i, get_noise=get_noise))
[0, 1, 2, 3, 5, 4, 6, 7, 9, 8]
```

class `torchnlp.samplers.`**OomBatchSampler**(*batch_sampler*, *get_item_size*, *num_batches=5*)

Out-of-memory (OOM) batch sampler wraps *batch_sampler* to sample the *num_batches* largest batches first in attempt to cause any potential OOM errors early.

Credits: https://github.com/allenai/allennlp/blob/3d100d31cc8d87efcf95c0b8d162bfce55c64926/allennlp/data/iterators/bucket_iterator.py#L43

Parameters

- **batch_sampler** (*torch.utils.data.sampler.BatchSampler*) –
- **get_item_size** (*callable*) – Measure the size of an item given it's index *int*.
- **num_batches** (*int, optional*) – The number of the large batches to move to the beginning of the iteration.

class `torchnlp.samplers.`**RepeatSampler**(*sampler*)

Sampler that repeats forever.

Background: The repeat sampler can be used with the `DataLoader` with option to re-use worker processes. Learn more here: https://github.com/pytorch/pytorch/issues/15849

Parameters **sampler** (*torch.data.utils.sampler.Sampler*) –

class `torchnlp.samplers.`**SortedSampler**(*data*, *sort_key=<function identity>*)

Samples elements sequentially, always in the same order.

Parameters

- **data** (*iterable*) – Iterable data.
- **sort_key** (*callable*) – Specifies a function of one argument that is used to extract a numerical comparison key from each list element.

Example

```
>>> list(SortedSampler(range(10), sort_key=lambda i: -i))
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

CHAPTER 8

torchnlp.utils package

The `torchnlp.utils` package contains any other module or object that is useful in building out a NLP pipeline.

`torchnlp.utils.`**`collate_tensors`**(*batch*, *stack_tensors=<built-in method stack of type object>*)

Collate a list of type `k` (dict, namedtuple, list, etc.) with tensors.

Inspired by: https://github.com/pytorch/pytorch/blob/master/torch/utils/data/_utils/collate.py#L31

Parameters

- **`batch`** (*`list of k`*) – List of rows of type `k`.
- **`stack_tensors`** (*`callable`*) – Function to stack tensors into a batch.

Returns Collated batch of type `k`.

Return type `k`

Example use case: This is useful with `torch.utils.data.dataloader.DataLoader` which requires a collate function. Typically, when collating sequences you'd set `collate_fn=partial(collate_tensors, stack_tensors=encoders.text.stack_and_pad_tensors)`.

Example

```
>>> import torch
>>> batch = [
...   { 'column_a': torch.randn(5), 'column_b': torch.randn(5) },
...   { 'column_a': torch.randn(5), 'column_b': torch.randn(5) },
... ]
>>> collated = collate_tensors(batch)
>>> {k: t.size() for (k, t) in collated.items()}
{'column_a': torch.Size([2, 5]), 'column_b': torch.Size([2, 5])}
```

`torchnlp.utils.`**`flatten_parameters`**(*model*)

`flatten_parameters` of a RNN model loaded from disk.

torchnlp.utils.**get_tensors**(*object_*)
Get all tensors associated with `object_`

Parameters **object** (*any*) – Any object to look for tensors.

Returns List of tensors that are associated with `object_`.

Return type (list of torch.tensor)

torchnlp.utils.**get_total_parameters**(*model*)
Return the total number of trainable parameters in `model`.

Parameters **model** (*torch.nn.Module*) –

Returns The total number of trainable parameters in `model`.

Return type (int)

torchnlp.utils.**identity**(*x*)

torchnlp.utils.**is_namedtuple**(*object_*)

torchnlp.utils.**lengths_to_mask**(**lengths*, ***kwargs*)
Given a list of lengths, create a batch mask.

Example

```
>>> lengths_to_mask([1, 2, 3])
tensor([[ True, False, False],
        [ True,  True, False],
        [ True,  True,  True]])
>>> lengths_to_mask([1, 2, 2], [1, 2, 2])
tensor([[[ True, False],
         [False, False]],
<BLANKLINE>
        [[ True,  True],
         [ True,  True]],
<BLANKLINE>
        [[ True,  True],
         [ True,  True]]])
```

Parameters

- ***lengths** (*list of python:int or torch.Tensor*) –
- ****kwargs** – Keyword arguments passed to `torch.zeros` upon initially creating the returned tensor.

Returns torch.BoolTensor

torchnlp.utils.**sampler_to_iterator**(*dataset*, *sampler*)
Given a batch sampler or sampler returns examples instead of indices

Parameters

- **dataset** (*torch.utils.data.Dataset*) – Dataset to sample from.
- **sampler** (*torch.utils.data.sampler.Sampler*) – Sampler over the dataset.

Returns generator over dataset examples

torchnlp.utils.**split_list**(*list_*, *splits*)
Split `list_` using the `splits` ratio.

Parameters

- **list** (*list*) – List to split.
- **splits** (*tuple*) – Tuple of decimals determining list splits summing up to 1.0.

Returns Splits of the list.

Return type (list)

Example

```
>>> dataset = [1, 2, 3, 4, 5]
>>> split_list(dataset, splits=(.6, .2, .2))
[[1, 2, 3], [4], [5]]
```

torchnlp.utils.**tensors_to**(*tensors*, **args*, ***kwargs*)

Apply `torch.Tensor.to` to tensors in a generic data structure.

Inspired by: https://github.com/pytorch/pytorch/blob/master/torch/utils/data/_utils/collate.py#L31

Parameters

- **tensors** (*tensor, dict, list, namedtuple or tuple*) – Data structure with tensor values to move.
- ***args** – Arguments passed to `torch.Tensor.to`.
- ****kwargs** – Keyword arguments passed to `torch.Tensor.to`.

Example use case: This is useful as a complementary function to `collate_tensors`. Following collating, it's important to move your tensors to the appropriate device.

Returns The inputted `tensors` with `torch.Tensor.to` applied.

Example

```
>>> import torch
>>> batch = [
...   { 'column_a': torch.randn(5), 'column_b': torch.randn(5) },
...   { 'column_a': torch.randn(5), 'column_b': torch.randn(5) },
... ]
>>> tensors_to(batch, torch.device('cpu'))  # doctest: +ELLIPSIS
[{'column_a': tensor(...}]
```

torchnlp.utils.**torch_equals_ignore_index**(*tensor*, *tensor_other*, *ignore_index=None*)

Compute `torch.equal` with the optional mask parameter.

Parameters **ignore_index** (*int, optional*) – Specifies a `tensor` index that is ignored.

Returns (bool) Returns `True` if target and prediction are equal.

CHAPTER 9

torchnlp.word_to_vector package

The `torchnlp.word_to_vector` package introduces multiple pretrained word vectors. The package handles downloading, caching, loading, and lookup.

class `torchnlp.word_to_vector.`**GloVe**(*name='840B'*, *dim=300*, ***kwargs*)

Word vectors derived from word-word co-occurrence statistics from a corpus by Stanford.

GloVe is essentially a log-bilinear model with a weighted least-squares objective. The main intuition underlying the model is the simple observation that ratios of word-word co-occurrence probabilities have the potential for encoding some form of meaning.

Reference: https://nlp.stanford.edu/projects/glove/

Parameters

- **name** (*str*) – name of the GloVe vectors ('840B', 'twitter.27B', '6B', '42B')
- **cache** (*str, optional*) – directory for cached vectors
- **unk_init** (*callback, optional*) – by default, initialize out-of-vocabulary word vectors to zero vectors; can be any function that takes in a Tensor and returns a Tensor of the same size
- **is_include** (*callable, optional*) – callable returns True if to include a token in memory vectors cache; some of these embedding files are gigantic so filtering it can cut down on the memory usage. We do not cache on disk if `is_include` is defined.

Example

```
>>> from torchnlp.word_to_vector import GloVe  # doctest: +SKIP
>>> vectors = GloVe()  # doctest: +SKIP
>>> vectors['hello']  # doctest: +SKIP
-1.7494
0.6242
...
-0.6202
```

(continues on next page)

(continued from previous page)

```
2.0928
[torch.FloatTensor of size 100]
```

class torchnlp.word_to_vector.**BPEmb**(*language='en'*, *dim=300*, *merge_ops=50000*, ***kwargs*)

Byte-Pair Encoding (BPE) embeddings trained on Wikipedia for 275 languages

A collection of pre-trained subword unit embeddings in 275 languages, based on Byte-Pair Encoding (BPE). In an evaluation using fine-grained entity typing as testbed, BPEmb performs competitively, and for some languages better than alternative subword approaches, while requiring vastly fewer resources and no tokenization.

References

- https://arxiv.org/abs/1710.02187
- https://github.com/bheinzerling/bpemb

Parameters

- **language** (*str, optional*) – Language of the corpus on which the embeddings have been trained
- **dim** (*int, optional*) – Dimensionality of the embeddings
- **merge_ops** (*int, optional*) – Number of merge operations used by the tokenizer

Example

```
>>> from torchnlp.word_to_vector import BPEmb  # doctest: +SKIP
>>> vectors = BPEmb(dim=25)  # doctest: +SKIP
>>> subwords = "mel ford shire".split()  # doctest: +SKIP
>>> vectors[subwords]  # doctest: +SKIP
Columns 0 to 9
-0.5859 -0.1803  0.2623 -0.6052  0.0194 -0.2795  0.2716 -0.2957 -0.0492
1.0934
 0.3848 -0.2412  1.0599 -0.8588 -1.2596 -0.2534 -0.5704  0.2168 -0.1718
1.2675
 1.4407 -0.0996  1.2239 -0.5085 -0.7542 -0.9628 -1.7177  0.0618 -0.4025
1.0405
...
Columns 20 to 24
-0.0022  0.4820 -0.5156 -0.0564  0.4300
 0.0355 -0.2257  0.1323  0.6053 -0.8878
-0.0167 -0.3686  0.9666  0.2497 -1.2239
[torch.FloatTensor of size 3x25]
```

class torchnlp.word_to_vector.**FastText**(*language='en'*, *aligned=False*, ***kwargs*)

Enriched word vectors with subword information from Facebook's AI Research (FAIR) lab.

A approach based on the skipgram model, where each word is represented as a bag of character n-grams. A vector representation is associated to each character n-gram; words being represented as the sum of these representations.

References

- https://arxiv.org/abs/1607.04606
- https://fasttext.cc/
- https://arxiv.org/abs/1710.04087

Parameters

- **language** (*str*) – language of the vectors
- **aligned** (*bool*) – if True: use multilingual embeddings where words with the same meaning share (approximately) the same position in the vector space across languages. if False: use regular FastText embeddings. All available languages can be found under https://github.com/facebookresearch/MUSE#multilingual-word-embeddings
- **cache** (*str, optional*) – directory for cached vectors
- **unk_init** (*callback, optional*) – by default, initialize out-of-vocabulary word vectors to zero vectors; can be any function that takes in a Tensor and returns a Tensor of the same size
- **is_include** (*callable, optional*) – callable returns True if to include a token in memory vectors cache; some of these embedding files are gigantic so filtering it can cut down on the memory usage. We do not cache on disk if `is_include` is defined.

Example

```
>>> from torchnlp.word_to_vector import FastText  # doctest: +SKIP
>>> vectors = FastText()  # doctest: +SKIP
>>> vectors['hello']  # doctest: +SKIP
-0.1595
-0.1826
...
0.2492
0.0654
[torch.FloatTensor of size 300]
```

class torchnlp.word_to_vector.**CharNGram**(***kwargs*)

Character n-gram is a character-based compositional model to embed textual sequences.

Character n-gram embeddings are trained by the same Skip-gram objective. The final character embedding is the average of the unique character n-gram embeddings of wt. For example, the character n-grams (n = 1, 2, 3) of the word "Cat" are {C, a, t, #B#C, Ca, at, t#E#, #B#Ca, Cat, at#E#}, where "#B#" and "#E#" represent the beginning and the end of each word, respectively. Using the character embeddings efficiently provides morphological features. Each word is subsequently represented as xt, the concatenation of its corresponding word and character embeddings shared across the tasks.

Reference: http://www.logos.t.u-tokyo.ac.jp/~hassy/publications/arxiv2016jmt/

Parameters

- **cache** (*str, optional*) – directory for cached vectors
- **unk_init** (*callback, optional*) – by default, initialize out-of-vocabulary word vectors to zero vectors; can be any function that takes in a Tensor and returns a Tensor of the same size

- **is_include** (*callable, optional*) – callable returns True if to include a token in memory vectors cache; some of these embedding files are gigantic so filtering it can cut down on the memory usage. We do not cache on disk if `is_include` is defined.

Example

```
>>> from torchnlp.word_to_vector import CharNGram  # doctest: +SKIP
>>> vectors = CharNGram()  # doctest: +SKIP
>>> vectors['hello']  # doctest: +SKIP
-1.7494
0.6242
...
-0.6202
2.0928
[torch.FloatTensor of size 100]
```

CHAPTER 10

Indices and tables

- genindex
- modindex

Python Module Index

t

Index

I

L

M

N

O

P

R

S

T

U

V

W

Z