Chainer Chemistry Documentation

Release 0.2.0

Preferred Networks, Inc.

Mar 01, 2018

Contents

Chainer Chemistry is a collection of tools to train and run neural networks for tasks in biology and chemistry using Chainer .

CHAPTER 1

Features

- State-of-the-art deep learning neural network models (especially graph convolutions) for chemical molecules (NFP, GGNN, Weave, SchNet etc.)
- Preprocessors of molecules tailored for these models
- Parsers for several standard file formats (CSV, SDF etc.)
- Loaders for several well-known datasets (QM9, Tox21 etc.)

1.1 Installation

1.1.1 Dependency

Following packages are required to install Chainer Chemistry and are automatically installed when you install the library by *pip* command.

- chainer
- pandas
- scikit-learn
- tqdm

Also, it uses following library, which you need to manually install.

- rdkit

See the official document for installation. If you have setup `anaconda`, you may install `rdkit` by following command:

```
$ conda install -c rdkit rdkit
```

1.1.2 Install via pip

It can be installed by `pip` command:

```
$ pip install chainer-chemistry
```

1.1.3 Install from source

The tarball of the source tree is available via `pip download chainer-chemistry`. You can use `setup.py` to install Chainer Chemistry from the tarball:

```
$ tar zxf chainer-chemistry-x.x.x.tar.gz
$ cd chainer-chemistry-x.x.x
$ python setup.py install
```

Install from the latest source from the master branch:

```
$ git clone https://github.com/pfnet-research/chainer-chemistry.git
$ pip install -e chainer-chemistry
```

1.1.4 Run example training code

The official repository provides examples of training several graph convolution networks. The code can be obtained by cloning the repository:

```
$ git clone https://github.com/pfnet-research/chainer-chemistry.git
```

The following code is how to train Neural Fingerprint (NFP) with the Tox21 dataset on CPU:

```
$ cd chainer-chemistry/examples/tox21
$ python train_tox21.py --method=nfp  --gpu=-1  # set --gpu=0 if you have GPU
```

1.2 Tutorial

1.2.1 Abstract

In this tutorial, we predict Highest Occupied Molecular Orbital (HOMO) level of the molecules in QM9 dataset [1][2] by Neural Finger Print (NFP) [3][4]. We concentrate on exaplaining usage of Chainer Chemistry briefly and do not look over the detail of NFP implementation.

1.2.2 Tested Environment

- Chainer Chemistry >= 0.0.1 (See *Installation*)
- Chainer >= 2.0.2
- CUDA == 8.0, CuPy >= 1.0.3 (Required only when using GPU)
  - For CUDA 9.0, CuPy >= 2.0.0 is required
- sklearn >= 0.17.1 (Only for preprocessing)

1.2.3 QM9 Dataset

QM9 is a publicly available dataset of small organic molecule structures and their simulated properties for data driven researches of material property prediction and chemical space exploration. It contains 133,885 stable small organic molecules made up of CHONF. The available properties are geometric, energetic, electronic, and thermodynamic ones.

In this tutorial, we predict HOMO level in the properties. Physically, we need quantum chemical calculations to compute HOMO level. From mathematical viewpoint it requires a solution of an internal eigenvalue problem for a Hamiltonian matrix. It is a big challenge to predict HOMO level accurately by a neural network, because the network should approximate both calculating the Hamiltonian matrix and solving the internal eigenvalue problem.

1.2.4 HOMO prediction by NFP

At first you should clone the library repository from GitHub. There is a Python script `examples/qm9/train_qm9.py` in the repository. It executes a whole training procedure, that is, downloads QM9 dataset, preprocess it, define an NFP model and run trainning on them.

Execute the following commands on a machine satisfying the tested environment in environment.

```
~$ git clone git@github.com:pfnet-research/chainer-chemistry.git
~$ cd chainer-chemistry/examples/qm9/
```

Hereafter all shell commands should be executed in this directory.

If you are a beginner for Chainer, Chainer handson will greatly help you. Especially the explanation of inclusion relationship of Chainer classes in Sec. 4 in Chap. 2 is helpful when you read the sample script.

Next the dataset preparation part and the model definition part in `train_qm9.py` are explained. If you are not interested in them, skip *Dataset Preparation* and *Model Definition*, and jump to *Run*.

Dataset Preparation

Chainer Chemistry accepts the same dataset type with Chainer, such as `chainer.datasets.SubDataset`. In this section we learn how to download QM9 dataset and use it as a Chainer dataset.

The following Python script downloads and saves the dataset in `.npz` format.

```
#!/usr/bin/env python
from chainer_chemistry import datasets as D
from chainer_chemistry.dataset.preprocessors import preprocess_method_dict
from chainer_chemistry.datasets import NumpyTupleDataset

preprocessor = preprocess_method_dict['nfp']()
dataset = D.get_qm9(preprocessor, labels='homo')
cache_dir = 'input/nfp_homo/'
os.makedirs(cache_dir)
NumpyTupleDataset.save(cache_dir + 'data.npz', dataset)
```

The last two lines save the dataset to `input/nfp_homo/data.npz` and we need not to download the dataset next time.

The following Python script read the dataset from the saved `.npz` file and split the data points into training and validation sets.

```
#!/usr/bin/env python
from chainer.datasets import split_dataset_random
from chainer_chemistry import datasets as D
from chainer_chemistry.dataset.preprocessors import preprocess_method_dict
from chainer_chemistry.datasets import NumpyTupleDataset

cache_dir = 'input/nfp_homo/'
dataset = NumpyTupleDataset.load(cache_dir + 'data.npz')
train_data_ratio = 0.7
train_data_size = int(len(dataset) * train_data_ratio)
train, val = split_dataset_random(dataset, train_data_size, 777)
print('train dataset size:', len(train))
print('validation dataset size:', len(val))
```

The function `split_dataset_random()` returns a tuple of two `chainer.datasets.SubDataset` objects (training and validation set). Now you have prepared training and validation data points and you can construct `chainer.iterator.Iterator` objects, needed for updaters in Chainer.

Model Definition

In Chainer, a neural network model is defined as a `chainer.Chain` object.

Graph convolutional networks such as NFP are generally connection of graph convolution layers and multi perceptron layers. Therefore it is convenient to define a class which inherits `chainer.Chain` and compose two `chainer.Chain` objects corresponding to the two kind of layers.

Execute the following Python script and check you can define such a class. `NFP` and `MLP` are already defined `chainer.Chain` classes.

```
#!/usr/bin/env python
import chainer
from chainer_chemistry.models import MLP, NFP

class GraphConvPredictor(chainer.Chain):

    def __init__(self, graph_conv, mlp):
        super(GraphConvPredictor, self).__init__()
        with self.init_scope():
            self.graph_conv = graph_conv
            self.mlp = mlp

    def __call__(self, atoms, adjs):
        x = self.graph_conv(atoms, adjs)
        x = self.mlp(x)
        return x

n_unit = 16
conv_layers = 4
model = GraphConvPredictor(NFP(n_unit, n_unit, conv_layers),
                           MLP(n_unit, 1))
```

Run

You have defined the dataset and the NFP model on Chainer. There are no other procedures specific to Chainer Chemistry. Hereafter you should just follow the usual procedures in Chainer to execute training.

The sample script examples/qm9/train_qm9.py contains all the procedures and you can execute training just by invoking the script. The following command starts training for 20 epochs and reports loss and accuracy during training. They are reported for each of main (dataset for training) and validation (dataset for validation).

The --gpu 0 option is to utilize a GPU with device id = 0. If you do not have a GPU, set --gpu -1 or just drop --gpu 0 to use CPU for all the calculation. In most cases, calculation with GPU is much faster than that only with CPU.

```
~/chainer-chemistry/examples/qm9$ python train_qm9.py --method nfp --label homo --gpu␣
↪0  # If GPU is unavailable, set --gpu -1

Train NFP model...
epoch       main/loss   main/accuracy  validation/main/loss  validation/main/accuracy␣
↪ elapsed_time
1           0.746135    0.0336724      0.680088              0.0322597               ␣
↪ 58.4605
2           0.642823    0.0311715      0.622942              0.0307055               ␣
↪ 113.748
(...)
19          0.540646    0.0277585      0.532406              0.0276445               ␣
↪ 1052.41
20          0.537062    0.0276631      0.551695              0.0277499               ␣
↪ 1107.29
```

After finished, you will find log file in result/ directory.

Evaluation

In the loss and accuracy report, we are mainly interested in validation/main/accuracy. Although it decreases during training, the accuracy field is actually mean absolute error. The unit is Hartree. Therefore the last line means validation mean absolute error is 0.0277499 Hartree. See scaled_abs_error() function in train_qm9.py for the detailed definition of mean absolute error.

You can also train other type models like GGNN, SchNet or WeaveNet, and other target values like LUMO, dipole moment and internal energy, just by changing --model and --label options, respectively. See output of python train_qm9.py --help.

1.2.5 Reference

[1] L. Ruddigkeit, R. van Deursen, L. C. Blum, J.-L. Reymond, Enumeration of 166 billion organic small molecules in the chemical universe database GDB-17, J. Chem. Inf. Model. 52, 2864–2875, 2012.

[2] R. Ramakrishnan, P. O. Dral, M. Rupp, O. A. von Lilienfeld, Quantum chemistry structures and properties of 134 kilo molecules, Scientific Data 1, 140022, 2014.

[3] Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., & Adams, R. P. (2015). Convolutional networks on graphs for learning molecular fingerprints. In Advances in neural information processing systems (pp. 2224-2232).

[4] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. arXiv preprint arXiv:1704.01212.

1.3 Contribution guide

We welcome any type of contribution that helps to improve and promote Chainer Chemistry. Typical contribution includes:

- Send pull requests (PRs) to the repository (We recommend developers making PRs to read the *Development policy* before starting to implement).
- Report bugs or problems as issues.
- Send questions to developer community sites like Stackoverflow or Chainer Slack (en, jp).
- Write a blog post about Chainer Chemistry or its use case.

1.4 Development policy

In this section, we describe the development policy that the core developers follow. Developers who are thinking to send PRs to the repository are encouraged to read the following sections before starting implementation.

1.4.1 Versioning policy

Basically, we follow the semantic versioning v2.0.0. In Chainer Chemistry, *public APIs* in the sense of semantic versioning are ones in the document.

We follow these rules about versioning during the major version zero in addition to ones described in the the semantic versioning:

- We do not plan any scheduled releases.
- We do not plan any pre releases.
- We release the minor version when the core development team agrees. Typically, we do so when (1) sufficient number of features are added since the last minor release (2) the latest release cannot run the example code in the master branch of the repository (3) critical bugs are found. But we are not restricted to them.
- If we find critical bugs, we should release a patch version or a minor version that fixes them. The core development team will determine which version to release.

We do not have a concrete plan about versioning strategy after v1.0.0.

1.4.2 Compatibiity policy

As an immediate consequence of the semantic versioning, we may break compatibility of public APIs including addition, deletion, change in semantics of them anytime in the major version zero. Since APIs of Chainer Chemistry are still immature and unstable, introduction of new features can sometime involve compatibility break. If we are faced with a dilemma between cost for backward compatibility and benefit of new features, we are likely to give up the former because we want to place importance on introducing new features as soon as possible. Of course, we care backward compatibility whenever it is easy and low-cost.

Like ChainerCV, Chainer Chemistry provides several off-the-shelf deep learning models (e.g. Neural Finger Print) whose papers are available in such as arXiv or conferences related to machine learning. Although, most of published papers reports evaluation results of the models with publicly available datasets, we do *NOT* guarantee the reproducibility of experiments in the papers.

At some point, coding examples in the master branch of the official repository may not work even with the latest release. In that case, users are recommended to either use the example code of the latest release or update the library code to the master branch.

1.4.3 Branch strategy

The official repository of Chainer Chemistry is https://github.com/pfnet-research/chainer-chemistry. We use the *master* branch of the repository for development. Therefore, developer who makes PRs should send them to the master branch.

During major version zero, we do not maintain any released versions. When a bug is found, changes for the bug should be merged to the next version (either minor or patch). If the bug is critical, we will release the next version as soon as possible.

1.4.4 Coding guideline

We basically adopt *PEP8 <https://www.python.org/dev/peps/pep-0008/>_* as a style guide. You can check it with *flake8*, which we can install by:

```
$ pip install flake8
```

and run with `flake8` command.

In addition to PEP8, we use upper camel case (e.g. `FooBar`) for class names and snake case (e.g. `foo_bar`) for function, method, variable and package names. Although we recommend developers to follow these rules as well, they are not mandatory.

For documents, we follow the Google Python Style Guide and compile it with Napoleon, which is an extension of Sphinx.

1.4.5 Testing guideline

Chainer Chemistry uses pytest as a unit-test framework. All unit tests are located in `tests/` directory. We can run tests with normal usage of pytest. For example, the following command runs all unit tests:

```
$ pytest tests
```

Some unit tests require GPUs, which are annotated with `@pytest.mark.gpu`. Therefore, you can skip them with `-m` option:

```
$ pytest -m "not gpu" tests
```

If a develop who write a unit test that uses GPUs, you must anotate it with `@pytest.mark.gpu`.

Similarly, some unit tests take long time to complete. We annotated them with `@pytest.mark.slow` and can skip them with `-m` option:

```
$ pytest -m "not slow" tests
```

Any unit test that uses GPUs muct be annotated with `@pytest.mark.slow`.

We can skip both GPU and slow tests with the following command:

```
$ pytest -m "not (gpu or slow)" tests
```

1.4.6 Terminology

In the context of machine learning, especially chemoinformatics, we use several terms such as feature, feature vectors, descriptor and so on to indicate representation of inputs. To avoid disambiguity and align naming convention within the library code, we use these terms in the following way:

- *Feature* is a representation of a sample of interest (typically molecules in Chainer Chemistry).
- *Label* is a target value of we want to predict.
- *Input feature* is a representation of a sample from which we want to predict the target value.

For example, consider a suepervised learning task whose dataset consisting of input-output pairs `((x_1, y_1), ..., (x_N, y_N))`, where `N` is the number of samples. In Chainer Chemistry `` x_i` and ``y_i `` are called input feature and label, respectively and a pair of `(x_i, y_i)` is feature for each `i`.

1.4.7 Relation to Chainer

Chainer is a deep learning framework written in Python that features dynamic computational graph construction (the "define-by-run" paradigm) for flexible and intuitive model development. As the name indicates, Chainer Chemistry is an extension library of Chainer built on top of it. The core development team members of Chainer and that of Chainer Chemistry work together tightly.

1.5 API Reference

1.5.1 Dataset

Converters

chainer_chemistry.dataset.converters.concat_mols	Concatenates a list of molecules into array(s).

chainer_chemistry.dataset.converters.concat_mols

`chainer_chemistry.dataset.converters.`**`concat_mols`**`(`*batch*, *device=None*, *padding=0*`)`

Concatenates a list of molecules into array(s).

This function converts "an arrays of tuples" into "a tuple of arrays". Given an list of examples each of which consists of multiple arrays, this function takes an array in the same "relative" position within an example from each example, concatenates them along the newly-inserted first axis (called *batch dimension*) into one array. It does the same thing with all positions, and returns resulting arrays.

The output type depends on the type of examples in `batch`. For instance, consider each example consists of two arrays `(x, y)`. Then, this function concatenates `x` 's into one array, and `y` 's into another array, and returns a tuple of these two arrays. Another example: consider each example is a dictionary of two entries whose keys are `'x'` and `'y'`, respectively, and values are arrays. Then, this function concatenates `x` 's into one array, and `y` 's into another array, and returns a dictionary with two entries `x` and `y` whose values are the concatenated arrays.

When the arrays to concatenate have different shapes, the behavior depends on the `padding` value. If `padding` is `None`, it raises an error. Otherwise, it builds an array of the minimum shape that the contents of all arrays can be substituted to. The padding value is then used to the extra elements of the resulting arrays.

The current implementation is identical to `concat_examples()` of Chainer, except the default value of the `padding` option is changed to `0`.

Example

```
>>> import numpy
>>> from chainer_chemistry.dataset.converters import concat_mols
>>> x0 = numpy.array([1, 2])
>>> x1 = numpy.array([4, 5, 6])
>>> dataset = [x0, x1]
>>> results = concat_mols(dataset)
>>> print(results)
[[1 2 0]
 [4 5 6]]
```

See also:

`chainer.dataset.concat_examples()`

Parameters

- **batch** (*list*) – A list of examples. This is typically given by a dataset iterator.
- **device** (*int*) – Device ID to which each array is sent. Negative value indicates the host memory (CPU). If it is omitted, all arrays are left in the original device.
- **padding** – Scalar value for extra elements. If this is None (default), an error is raised on shape mismatch. Otherwise, an array of minimum dimensionalities that can accommodate all arrays is created, and elements outside of the examples are padded by this value.

Returns The type depends on the type of each example in the batch.

Return type Array, a tuple of arrays, or a dictionary of arrays

Indexers

chainer_chemistry.dataset.indexer.BaseIndexer	Base class for Indexer
chainer_chemistry.dataset.indexer.BaseFeatureIndexer	Base class for FeatureIndexer
chainer_chemistry.dataset.indexers.NumpyTupleDatasetFeatureIndexer	FeatureIndexer for NumpyTupleDataset

chainer_chemistry.dataset.indexer.BaseIndexer

class chainer_chemistry.dataset.indexer.**BaseIndexer**

Base class for Indexer

__init__()

Initialize self. See help(type(self)) for accurate signature.

chainer_chemistry.dataset.indexer.BaseFeatureIndexer

class chainer_chemistry.dataset.indexer.**BaseFeatureIndexer**(*dataset*)

Base class for FeatureIndexer

FeatureIndexer can be accessed by 2-dimensional indices, axis=0 is used for dataset index and axis=1 is used for feature index. For example, let *features* be the instance of *BaseFeatureIndexer*, then *features[i, j]* returns *i*-th dataset of *j*-th feature.

features[ind] works same with *features[ind, :]*

Note that the returned value will be numpy array, even though the dataset is initilized with other format (e.g. list).

__init__(*dataset*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(dataset)	Initialize self.
`check_type_feature_index`(j)	
`create_feature_index_list`(feature_index)	
`extract_feature`(i, j)	Extracts *i*-th data's *j*-th feature
`extract_feature_by_slice`(slice_index, j)	Extracts *slice_index*-th data's *j*-th feature.
`features_length`()	Returns length of features
`postprocess`(item)	
`preprocess`(item)	

Attributes

`dataset_length`
`shape`

chainer_chemistry.dataset.indexers.NumpyTupleDatasetFeatureIndexer

class chainer_chemistry.dataset.indexers.**NumpyTupleDatasetFeatureIndexer**(*dataset*)
FeatureIndexer for NumpyTupleDataset

Parameters **dataset** (`NumpyTupleDataset`) – dataset instance

__init__(*dataset*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(dataset)	Initialize self.
`check_type_feature_index`(j)	
`create_feature_index_list`(feature_index)	
`extract_feature`(i, j)	Extracts *i*-th data's *j*-th feature
`extract_feature_by_slice`(slice_index, j)	Extracts *slice_index*-th data's *j*-th feature.
`features_length`()	Returns length of features
`postprocess`(item)	
`preprocess`(item)	

Attributes

dataset_length	
shape	

Parsers

chainer_chemistry.dataset.parsers.BaseParser	
chainer_chemistry.dataset.parsers.CSVFileParser	csv file parser
chainer_chemistry.dataset.parsers.SDFFileParser	sdf file parser

chainer_chemistry.dataset.parsers.BaseParser

class chainer_chemistry.dataset.parsers.**BaseParser**

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()	Initialize self.

chainer_chemistry.dataset.parsers.CSVFileParser

class chainer_chemistry.dataset.parsers.**CSVFileParser**(*preprocessor*, *labels=None*, *smiles_col='smiles'*, *postprocess_label=None*, *postprocess_fn=None*, *logger=None*)

csv file parser

This FileParser parses .csv file. It should contain column which contain SMILES as input, and label column which is the target to predict.

Parameters

- **preprocessor** (BasePreprocessor) – preprocessor instance
- **labels** (*str or list*) – labels column
- **smiles_col** (*str*) – smiles column
- **postprocess_label** (*Callable*) – post processing function if necessary
- **postprocess_fn** (*Callable*) – post processing function if necessary
- **logger** –

__init__(*preprocessor*, *labels=None*, *smiles_col='smiles'*, *postprocess_label=None*, *postprocess_fn=None*, *logger=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(preprocessor[, labels, smiles_col, ...])	Initialize self.
parse(filepath[, return_smiles])	parse csv file using *preprocessor*

chainer_chemistry.dataset.parsers.SDFFileParser

class chainer_chemistry.dataset.parsers.**SDFFileParser**(*preprocessor*, *labels=None*, *postprocess_label=None*, *postprocess_fn=None*, *logger=None*)

sdf file parser

Parameters

- **preprocessor** (BasePreprocessor) – preprocessor instance
- **labels** (*str or list*) – labels column
- **postprocess_label** (*Callable*) – post processing function if necessary
- **postprocess_fn** (*Callable*) – post processing function if necessary
- **logger** –

__init__(*preprocessor*, *labels=None*, *postprocess_label=None*, *postprocess_fn=None*, *logger=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(preprocessor[, labels, ...])	Initialize self.
parse(filepath[, return_smiles])	parse sdf file using *preprocessor*

Preprocessors

Base preprocessors

chainer_chemistry.dataset.preprocessors.BasePreprocessor	Base class for preprocessor
chainer_chemistry.dataset.preprocessors.MolPreprocessor	preprocessor class specified for rdkit mol instance

chainer_chemistry.dataset.preprocessors.BasePreprocessor

class chainer_chemistry.dataset.preprocessors.**BasePreprocessor**
Base class for preprocessor

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`()	Initialize self.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.MolPreprocessor

class `chainer_chemistry.dataset.preprocessors.`**`MolPreprocessor`**(*add_Hs=False*)
preprocessor class specified for rdkit mol instance

`__init__`(*add_Hs=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([add_Hs])	Initialize self.
`get_input_features`(mol)	get molecule's feature representation, descriptor.
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

Concrete preprocessors

`chainer_chemistry.dataset.preprocessors.AtomicNumberPreprocessor`	Atomic number Preprocessor
`chainer_chemistry.dataset.preprocessors.ECFPPreprocessor`	
`chainer_chemistry.dataset.preprocessors.GGNNPreprocessor`	GGNN Preprocessor
`chainer_chemistry.dataset.preprocessors.NFPPreprocessor`	NFP Preprocessor
`chainer_chemistry.dataset.preprocessors.SchNetPreprocessor`	SchNet Preprocessor
`chainer_chemistry.dataset.preprocessors.WeaveNetPreprocessor`	WeaveNet must have fixed-size atom list for now, zero_padding option is always set to True.

chainer_chemistry.dataset.preprocessors.AtomicNumberPreprocessor

class `chainer_chemistry.dataset.preprocessors.`**`AtomicNumberPreprocessor`**(*max_atoms=-1*, *out_size=-1*)
Atomic number Preprocessor

Parameters

- **max_atoms** (*int*) – Max number of atoms for each molecule, if the number of atoms is more than this value, this data is simply ignored. Setting negative value indicates no limit for max atoms.
- **out_size** (*int*) – It specifies the size of array returned by *get_input_features*. If the number of atoms in the molecule is less than this value, the returned arrays is padded to have fixed size. Setting negative value indicates do not pad returned array.

__init__(*max_atoms=-1*, *out_size=-1*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([max_atoms, out_size])	Initialize self.
`get_input_features`(mol)	get input features
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.ECFPPreprocessor

class chainer_chemistry.dataset.preprocessors.**ECFPPreprocessor**(*radius=2*)

__init__(*radius=2*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([radius])	Initialize self.
`get_input_features`(mol)	get molecule's feature representation, descriptor.
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.GGNNPreprocessor

class chainer_chemistry.dataset.preprocessors.**GGNNPreprocessor**(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)

GGNN Preprocessor

Parameters

- **max_atoms** (*int*) – Max number of atoms for each molecule, if the number of atoms is more than this value, this data is simply ignored. Setting negative value indicates no limit for max atoms.
- **out_size** (*int*) – It specifies the size of array returned by *get_input_features*. If the number of atoms in the molecule is less than this value, the returned arrays is padded to have fixed size. Setting negative value indicates do not pad returned array.

__init__(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([max_atoms, out_size, add_Hs])	Initialize self.
`get_input_features`(mol)	get input features
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.NFPPreprocessor

class `chainer_chemistry.dataset.preprocessors.`**NFPPreprocessor**(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)

NFP Preprocessor

Parameters

- **max_atoms** (*int*) – Max number of atoms for each molecule, if the number of atoms is more than this value, this data is simply ignored. Setting negative value indicates no limit for max atoms.
- **out_size** (*int*) – It specifies the size of array returned by *get_input_features*. If the number of atoms in the molecule is less than this value, the returned arrays is padded to have fixed size. Setting negative value indicates do not pad returned array.

__init__(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([max_atoms, out_size, add_Hs])	Initialize self.
`get_input_features`(mol)	get input features
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.SchNetPreprocessor

class `chainer_chemistry.dataset.preprocessors.`**SchNetPreprocessor**(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)

SchNet Preprocessor

Parameters

- **max_atoms** (*int*) – Max number of atoms for each molecule, if the
- **of atoms is more than this value, this data is simply** (*number*) –

- **ignored.** –
- **negative value indicates no limit for max atoms.** (*Setting*) –
- **zero_padding** (*bool*) – True
- **max_atoms** – Max number of atoms for each molecule, if the number of atoms is more than this value, this data is simply ignored. Setting negative value indicates no limit for max atoms.
- **out_size** (*int*) – It specifies the size of array returned by *get_input_features*. If the number of atoms in the molecule is less than this value, the returned arrays is padded to have fixed size. Setting negative value indicates do not pad returned array.

__init__(*max_atoms=-1*, *out_size=-1*, *add_Hs=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([max_atoms, out_size, add_Hs])	Initialize self.
`get_input_features`(mol)	get input features
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

chainer_chemistry.dataset.preprocessors.WeaveNetPreprocessor

class chainer_chemistry.dataset.preprocessors.**WeaveNetPreprocessor**(*max_atoms=20*, *add_Hs=True*, *use_fixed_atom_feature=False*, *atom_list=None*, *include_unknown_atom=False*)

WeaveNet must have fixed-size atom list for now, zero_padding option is always set to True.

Parameters

- **max_atoms** (*int*) – Max number of atoms for each molecule, if the number of atoms is more than this value, this data is simply ignored. Setting negative value indicates no limit for max atoms.
- **add_Hs** (*bool*) – If True, implicit Hs are added.
- **use_fixed_atom_feature** (*bool*) – If True, atom feature is extracted used in original paper. If it is False, atomic number is used instead.
- **atom_list** (*list*) – list of atoms to extract feature. If None, default *ATOM* is used as *atom_list*
- **include_unknown_atom** (*bool*) – If False, when the *mol* includes atom which is not in *atom_list*, it will raise *MolFeatureExtractionError*. If True, even the atom is not in *atom_list*, *atom_type* is set as “unknown” atom.

__init__(*max_atoms=20*, *add_Hs=True*, *use_fixed_atom_feature=False*, *atom_list=None*, *include_unknown_atom=False*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([max_atoms, add_Hs, ...])	Initialize self.
`get_input_features`(mol)	get input features for WeaveNet
`get_label`(mol[, label_names])	Extracts label information from a molecule.
`prepare_smiles_and_mol`(mol)	Prepare *smiles* and *mol* used in following preprocessing.
`process`(filepath)	

Utilities

`chainer_chemistry.dataset.preprocessors.MolFeatureExtractionError`	
`chainer_chemistry.dataset.preprocessors.type_check_num_atoms`	Check number of atoms in *mol* does not exceed *num_max_atoms*
`chainer_chemistry.dataset.preprocessors.construct_atomic_number_array`	Returns atomic numbers of atoms consisting a molecule.
`chainer_chemistry.dataset.preprocessors.construct_adj_matrix`	Returns the adjacent matrix of the given molecule.

chainer_chemistry.dataset.preprocessors.MolFeatureExtractionError

exception `chainer_chemistry.dataset.preprocessors.`**`MolFeatureExtractionError`**

chainer_chemistry.dataset.preprocessors.type_check_num_atoms

`chainer_chemistry.dataset.preprocessors.`**`type_check_num_atoms`**(*mol*, *num_max_atoms=-1*)

Check number of atoms in *mol* does not exceed *num_max_atoms*

If number of atoms in *mol* exceeds the number *num_max_atoms*, it will raise *MolFeatureExtractionError* exception.

Parameters

- **mol** (*Mol*) –
- **num_max_atoms** (*int*) – If negative value is set, not check number of atoms.

chainer_chemistry.dataset.preprocessors.construct_atomic_number_array

`chainer_chemistry.dataset.preprocessors.`**`construct_atomic_number_array`**(*mol*, *out_size=-1*)

Returns atomic numbers of atoms consisting a molecule.

Parameters

- **mol** (*rdkit.Chem.Mol*) – Input molecule.

- **out_size** (*int*) – The size of returned array. If this option is negative, it does not take any effect. Otherwise, it must be larger than the number of atoms in the input molecules. In that case, the tail of the array is padded with zeros.

Returns

an array consisting of atomic numbers of atoms in the molecule.

Return type numpy.ndarray

chainer_chemistry.dataset.preprocessors.construct_adj_matrix

`chainer_chemistry.dataset.preprocessors.`**`construct_adj_matrix`**(*mol*, *out_size=-1*, *self_connection=True*)

Returns the adjacent matrix of the given molecule.

This function returns the adjacent matrix of the given molecule. Contrary to the specification of `rdkit.Chem.rdmolops.GetAdjacencyMatrix()`, The diagonal entries of the returned matrix are all-one.

Parameters

- **mol** (*rdkit.Chem.Mol*) – Input molecule.
- **out_size** (*int*) – The size of the returned matrix. If this option is negative, it does not take any effect. Otherwise, it must be larger than the number of atoms in the input molecules. In that case, the adjacent matrix is expanded and zeros are padded to right columns and bottom rows.
- **self_connection** (*bool*) – Add self connection or not. If True, diagonal element of adjacency matrix is filled with 1.

Returns

the adjcent matrix of the input molecule. If `out_size` is non-negative, the returned its size is equal to that value. Otherwise, it is equal to the number of atoms in the the molecule.

Return type numpy.ndarray

1.5.2 Datasets

Dataset implementations

chainer_chemistry.datasets.NumpyTupleDataset	Dataset of a tuple of datasets.

chainer_chemistry.datasets.NumpyTupleDataset

class `chainer_chemistry.datasets.`**`NumpyTupleDataset`**(**datasets*)

Dataset of a tuple of datasets.

It combines multiple datasets into one dataset. Each example is represented by a tuple whose `i`-th item corresponds to the i-th dataset. And each `i`-th dataset is expected to be an instance of numpy.ndarray.

Parameters datasets – Underlying datasets. The `i`-th one is used for the `i`-th item of each example. All datasets must have the same length.

`__init__`(**datasets*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(*datasets)	Initialize self.
`get_datasets`()	
`load`(filepath)	
`save`(filepath, numpy_tuple_dataset)	save the dataset to filepath in npz format

Attributes

`features`	Extract features according to the specified index.

Dataset loaders

`chainer_chemistry.datasets.tox21.get_tox21`	Downloads, caches and preprocesses Tox21 dataset.
`chainer_chemistry.datasets.qm9.get_qm9`	Downloads, caches and preprocesses QM9 dataset.

chainer_chemistry.datasets.tox21.get_tox21

`chainer_chemistry.datasets.tox21.`**`get_tox21`**(*preprocessor=None*, *labels=None*, *return_smiles=False*)

Downloads, caches and preprocesses Tox21 dataset.

Parameters

- **`preprocesssor`** (`BasePreprocessor`) – Preprocessor. This should be chosen based on the network to be trained. If it is None, default *AtomicNumberPreprocessor* is used.
- **`labels`** (*`str or list`*) – List of target labels.
- **`return_smiles`** (*`bool`*) – If set to True, smiles array is also returned.

Returns The 3-tuple consisting of train, validation and test datasets, respectively. Each dataset is composed of *features*, which depends on *preprocess_method*.

chainer_chemistry.datasets.qm9.get_qm9

`chainer_chemistry.datasets.qm9.`**`get_qm9`**(*preprocessor=None*, *labels=None*, *return_smiles=False*)

Downloads, caches and preprocesses QM9 dataset.

Parameters

- **`preprocessor`** (`BasePreprocessor`) – Preprocessor. This should be chosen based on the network to be trained. If it is None, default *AtomicNumberPreprocessor* is used.
- **`labels`** (*`str or list`*) – List of target labels.
- **`return_smiles`** (*`bool`*) – If set to `True`, smiles array is also returned.

Returns dataset, which is composed of *features*, which depends on *preprocess_method*.

1.5.3 Functions

Function implementations

chainer_chemistry.functions.matmul	Computes the matrix multiplication of two arrays.

chainer_chemistry.functions.matmul

chainer_chemistry.functions.**matmul**(*a*, *b*, *transa=False*, *transb=False*)

Computes the matrix multiplication of two arrays.

Parameters

- **a** (*Variable*) – The left operand of the matrix multiplication. If a and b are both 1-D arrays, matmul returns a dot product of vector *a* and vector *b*. If 2-D arrays, matmul returns matrix product of a and b. If arrays' dimension is larger than 2, they are treated as a stack of matrices residing in the last two indexes. matmul returns a stack of each two arrays. a and b must have the same dimension.
- **b** (*Variable*) – The right operand of the matrix multiplication. Its array is treated as a matrix in the same way as a's array.
- **transa** (*bool*) – If True, each matrices in a will be transposed. If a.ndim == 1, do nothing.
- **transb** (*bool*) – If True, each matrices in b will be transposed. If b.ndim == 1, do nothing.

Returns The result of the matrix multiplication.

Return type Variable

Example

```
>>> a = np.array([[1, 0], [0, 1]], 'f')
>>> b = np.array([[4, 1], [2, 2]], 'f')
>>> F.matmul(a, b).data
array([[ 4.,  1.],
       [ 2.,  2.]], dtype=float32)
```

1.5.4 Links

Link implementations

chainer_chemistry.links.EmbedAtomID	Embeddning specialized to atoms.
chainer_chemistry.links.GraphLinear	Graph Linear layer.

chainer_chemistry.links.EmbedAtomID

class chainer_chemistry.links.**EmbedAtomID**(*out_size*, *in_size=117*, *initialW=None*, *ignore_label=None*)

Embeddning specialized to atoms.

This is a chain in the sense of Chainer that converts an atom, represented by a sequence of molecule IDs, to a sequence of embedding vectors of molecules. The operation is done in a minibatch manner, as most chains do.

The forward propagation of link consists of ID embedding, which converts the input x into vector embedding h where its shape represents (minibatch, atom, channel)

See also:

`chainer.links.EmbedID`

`__init__`(*out_size*, *in_size=117*, *initialW=None*, *ignore_label=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(out_size[, in_size, initialW, . . .])	Initialize self.
`add_param`(name[, shape, dtype, initializer])	Registers a parameter to the link.
`add_persistent`(name, value)	Registers a persistent value to the link.
`addgrads`(link)	Accumulates gradient values from given link.
`children`()	Returns a generator of all child links.
`cleargrads`()	Clears all gradient arrays.
`copy`()	Copies the link hierarchy to new one.
`copyparams`(link)	Copies all parameters from given link.
`disable_update`()	Disables update rules of all parameters under the link hierarchy.
`enable_update`()	Enables update rules of all parameters under the link hierarchy.
`init_scope`()	Creates an initialization scope.
`links`([skipself])	Returns a generator of all links under the hierarchy.
`namedlinks`([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
`namedparams`([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
`params`([include_uninit])	Returns a generator of all parameters under the link hierarchy.
`register_persistent`(name)	Registers an attribute of a given name as a persistent value.
`serialize`(serializer)	Serializes the link object.
`to_cpu`()	Copies parameter variables and persistent values to CPU.
`to_gpu`([device])	Copies parameter variables and persistent values to GPU.
`zerograds`()	Initializes all gradient arrays by zero.

Attributes

`ignore_label`	
`update_enabled`	`True` if at least one parameter has an update rule enabled.
`within_init_scope`	True if the current code is inside of an initialization scope.

Continued on next page

Table 29 – continued from previous page

xp	Array module for this link.

chainer_chemistry.links.GraphLinear

class chainer_chemistry.links.**GraphLinear**(*in_size*, *out_size=None*, *nobias=False*, *initialW=None*, *initial_bias=None*)

Graph Linear layer.

This function assumes its input is 3-dimensional. Differently from chainer.functions.linear, it applies an affine transformation to the third axis of input *x*.

See also:

chainer.links.Linear

__init__(*in_size*, *out_size=None*, *nobias=False*, *initialW=None*, *initial_bias=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(in_size[, out_size, nobias, ...])	Initialize self.
add_param(name[, shape, dtype, initializer])	Registers a parameter to the link.
add_persistent(name, value)	Registers a persistent value to the link.
addgrads(link)	Accumulates gradient values from given link.
children()	Returns a generator of all child links.
cleargrads()	Clears all gradient arrays.
copy()	Copies the link hierarchy to new one.
copyparams(link)	Copies all parameters from given link.
disable_update()	Disables update rules of all parameters under the link hierarchy.
enable_update()	Enables update rules of all parameters under the link hierarchy.
init_scope()	Creates an initialization scope.
links([skipself])	Returns a generator of all links under the hierarchy.
namedlinks([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
namedparams([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
params([include_uninit])	Returns a generator of all parameters under the link hierarchy.
register_persistent(name)	Registers an attribute of a given name as a persistent value.
serialize(serializer)	Serializes the link object.
to_cpu()	Copies parameter variables and persistent values to CPU.
to_gpu([device])	Copies parameter variables and persistent values to GPU.
zerograds()	Initializes all gradient arrays by zero.

Attributes

update_enabled	True if at least one parameter has an update rule enabled.
within_init_scope	True if the current code is inside of an initialization scope.
xp	Array module for this link.

1.5.5 Models

Model implementations

chainer_chemistry.models.NFP	Neural Finger Print (NFP)
chainer_chemistry.models.GGNN	Gated Graph Neural Networks (GGNN)
chainer_chemistry.models.MLP	Basic implementation for MLP
chainer_chemistry.models.SchNet	**param out_dim** dimension of output feature vector :type out_dim: int :param hidden_dim: dimension of feature vector associated to each atom :type hidden_dim: int :param n_layers: number of layers :type n_layers: int :param readout_hidden_dim: dimension of feature vector associated to each molecule :type readout_hidden_dim: int :param n_atom_types: number of types of atoms :type n_atom_types: int :param concat_hidden: If set to True, readout is executed in each layer and the result is concatenated :type concat_hidden: bool
chainer_chemistry.models.WeaveNet	WeaveNet implementation

chainer_chemistry.models.NFP

class chainer_chemistry.models.**NFP**(*out_dim*, *hidden_dim=16*, *n_layers=4*, *max_degree=6*, *n_atom_types=117*, *concat_hidden=False*)

Neural Finger Print (NFP)

Parameters

- **out_dim** (*int*) – dimension of output feature vector
- **hidden_dim** (*int*) – dimension of feature vector associated to each atom
- **max_degree** (*int*) – max degree of atoms when molecules are regarded as graphs
- **n_atom_types** (*int*) – number of types of atoms
- **n_layer** (*int*) – number of layers

__init__(*out_dim*, *hidden_dim=16*, *n_layers=4*, *max_degree=6*, *n_atom_types=117*, *concat_hidden=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(out_dim[, hidden_dim, n_layers, . . .])	Initialize self.
`add_link`(name, link)	Registers a child link to this chain.
`add_param`(name[, shape, dtype, initializer])	Registers a parameter to the link.
`add_persistent`(name, value)	Registers a persistent value to the link.
`addgrads`(link)	Accumulates gradient values from given link.
`children`()	Returns a generator of all child links.
`cleargrads`()	Clears all gradient arrays.
`copy`()	Copies the link hierarchy to new one.
`copyparams`(link)	Copies all parameters from given link.
`disable_update`()	Disables update rules of all parameters under the link hierarchy.
`enable_update`()	Enables update rules of all parameters under the link hierarchy.
`init_scope`()	Creates an initialization scope.
`links`([skipself])	Returns a generator of all links under the hierarchy.
`namedlinks`([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
`namedparams`([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
`params`([include_uninit])	Returns a generator of all parameters under the link hierarchy.
`register_persistent`(name)	Registers an attribute of a given name as a persistent value.
`serialize`(serializer)	Serializes the link object.
`to_cpu`()	Copies parameter variables and persistent values to CPU.
`to_gpu`([device])	Copies parameter variables and persistent values to GPU.
`zerograds`()	Initializes all gradient arrays by zero.

Attributes

`update_enabled`	`True` if at least one parameter has an update rule enabled.
`within_init_scope`	True if the current code is inside of an initialization scope.
`xp`	Array module for this link.

chainer_chemistry.models.GGNN

class `chainer_chemistry.models.`**`GGNN`**(*out_dim*, *hidden_dim=16*, *n_layers=4*, *n_atom_types=117*, *concat_hidden=False*, *weight_tying=True*)

Gated Graph Neural Networks (GGNN)

See: Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015). Gated graph sequence neural networks. arXiv:1511.05493

Parameters

- **out_dim** (*int*) – dimension of output feature vector
- **hidden_dim** (*int*) – dimension of feature vector associated to each atom
- **n_layers** (*int*) – number of layers
- **n_atom_types** (*int*) – number of types of atoms
- **concat_hidden** (*bool*) – If set to True, readout is executed in each layer and the result is concatenated
- **weight_tying** (*bool*) – enable weight_tying or not

__init__(*out_dim*, *hidden_dim=16*, *n_layers=4*, *n_atom_types=117*, *concat_hidden=False*, *weight_tying=True*)
Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`(out_dim[, hidden_dim, n_layers, . . .])	Initialize self.
`add_link`(name, link)	Registers a child link to this chain.
`add_param`(name[, shape, dtype, initializer])	Registers a parameter to the link.
`add_persistent`(name, value)	Registers a persistent value to the link.
`addgrads`(link)	Accumulates gradient values from given link.
`children`()	Returns a generator of all child links.
`cleargrads`()	Clears all gradient arrays.
`copy`()	Copies the link hierarchy to new one.
`copyparams`(link)	Copies all parameters from given link.
`disable_update`()	Disables update rules of all parameters under the link hierarchy.
`enable_update`()	Enables update rules of all parameters under the link hierarchy.
`init_scope`()	Creates an initialization scope.
`links`([skipself])	Returns a generator of all links under the hierarchy.
`namedlinks`([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
`namedparams`([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
`params`([include_uninit])	Returns a generator of all parameters under the link hierarchy.
`readout`(h, h0[, step])	
`register_persistent`(name)	Registers an attribute of a given name as a persistent value.
`serialize`(serializer)	Serializes the link object.
`to_cpu`()	Copies parameter variables and persistent values to CPU.
`to_gpu`([device])	Copies parameter variables and persistent values to GPU.
`update`(h, adj[, step])	
`zerograds`()	Initializes all gradient arrays by zero.

Attributes

NUM_EDGE_TYPE	
update_enabled	True if at least one parameter has an update rule enabled.
within_init_scope	True if the current code is inside of an initialization scope.
xp	Array module for this link.

chainer_chemistry.models.MLP

class chainer_chemistry.models.**MLP**(*out_dim*, *hidden_dim=16*, *n_layers=2*, *activation=<function relu>*)

Basic implementation for MLP

Parameters

- **out_dim** (*int*) – dimension of output feature vector
- **hidden_dim** (*int*) – dimension of feature vector associated to each atom
- **n_layers** (*int*) – number of layers
- **activation** (*chainer.functions*) – activation function

__init__(*out_dim*, *hidden_dim=16*, *n_layers=2*, *activation=<function relu>*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(out_dim[, hidden_dim, n_layers, ...])	Initialize self.
add_link(name, link)	Registers a child link to this chain.
add_param(name[, shape, dtype, initializer])	Registers a parameter to the link.
add_persistent(name, value)	Registers a persistent value to the link.
addgrads(link)	Accumulates gradient values from given link.
children()	Returns a generator of all child links.
cleargrads()	Clears all gradient arrays.
copy()	Copies the link hierarchy to new one.
copyparams(link)	Copies all parameters from given link.
disable_update()	Disables update rules of all parameters under the link hierarchy.
enable_update()	Enables update rules of all parameters under the link hierarchy.
init_scope()	Creates an initialization scope.
links([skipself])	Returns a generator of all links under the hierarchy.
namedlinks([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
namedparams([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
params([include_uninit])	Returns a generator of all parameters under the link hierarchy.
register_persistent(name)	Registers an attribute of a given name as a persistent value.
serialize(serializer)	Serializes the link object.

Continued on next page

Table 37 – continued from previous page

to_cpu()	Copies parameter variables and persistent values to CPU.
to_gpu([device])	Copies parameter variables and persistent values to GPU.
zerograds()	Initializes all gradient arrays by zero.

Attributes

update_enabled	True if at least one parameter has an update rule enabled.
within_init_scope	True if the current code is inside of an initialization scope.
xp	Array module for this link.

chainer_chemistry.models.SchNet

class chainer_chemistry.models.**SchNet**(*out_dim=1*, *hidden_dim=64*, *n_layers=3*, *readout_hidden_dim=32*, *n_atom_types=117*, *concat_hidden=False*)

Parameters

- **out_dim** (*int*) – dimension of output feature vector
- **hidden_dim** (*int*) – dimension of feature vector associated to each atom
- **n_layers** (*int*) – number of layers
- **readout_hidden_dim** (*int*) – dimension of feature vector associated to each molecule
- **n_atom_types** (*int*) – number of types of atoms
- **concat_hidden** (*bool*) – If set to True, readout is executed in each layer and the result is concatenated

__init__(*out_dim=1*, *hidden_dim=64*, *n_layers=3*, *readout_hidden_dim=32*, *n_atom_types=117*, *concat_hidden=False*)

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__([out_dim, hidden_dim, n_layers, . . .])	Initialize self.
add_link(name, link)	Registers a child link to this chain.
add_param(name[, shape, dtype, initializer])	Registers a parameter to the link.
add_persistent(name, value)	Registers a persistent value to the link.
addgrads(link)	Accumulates gradient values from given link.
children()	Returns a generator of all child links.
cleargrads()	Clears all gradient arrays.
copy()	Copies the link hierarchy to new one.
copyparams(link)	Copies all parameters from given link.
disable_update()	Disables update rules of all parameters under the link hierarchy.

Continued on next page

Table 39 – continued from previous page

`enable_update`()	Enables update rules of all parameters under the link hierarchy.
`init_scope`()	Creates an initialization scope.
`links`([skipself])	Returns a generator of all links under the hierarchy.
`namedlinks`([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
`namedparams`([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
`params`([include_uninit])	Returns a generator of all parameters under the link hierarchy.
`register_persistent`(name)	Registers an attribute of a given name as a persistent value.
`serialize`(serializer)	Serializes the link object.
`to_cpu`()	Copies parameter variables and persistent values to CPU.
`to_gpu`([device])	Copies parameter variables and persistent values to GPU.
`zerograds`()	Initializes all gradient arrays by zero.

Attributes

`update_enabled`	`True` if at least one parameter has an update rule enabled.
`within_init_scope`	True if the current code is inside of an initialization scope.
`xp`	Array module for this link.

chainer_chemistry.models.WeaveNet

class `chainer_chemistry.models.`**`WeaveNet`**(*weave_channels=None*, *hidden_dim=16*, *n_atom=20*, *n_sub_layer=1*, *n_atom_types=117*, *readout_mode='sum'*)

WeaveNet implementation

Parameters

- **`weave_channels`** (*`list`*) – list of int, output dimension for each weave module
- **`hidden_dim`** (*`int`*) – hidden dim
- **`n_atom`** (*`int`*) – number of atom of input array
- **`n_sub_layer`** (*`int`*) – number of layer for each *AtomToPair*, *PairToAtom* layer
- **`n_atom_types`** (*`int`*) – number of atom id
- **`readout_mode`** (*`str`*) – 'sum' or 'max' or 'summax'

`__init__`(*weave_channels=None*, *hidden_dim=16*, *n_atom=20*, *n_sub_layer=1*, *n_atom_types=117*, *readout_mode='sum'*)

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__`([weave_channels, hidden_dim, ...])	Initialize self.
`add_link`(name, link)	Registers a child link to this chain.
`add_param`(name[, shape, dtype, initializer])	Registers a parameter to the link.
`add_persistent`(name, value)	Registers a persistent value to the link.
`addgrads`(link)	Accumulates gradient values from given link.
`children`()	Returns a generator of all child links.
`cleargrads`()	Clears all gradient arrays.
`copy`()	Copies the link hierarchy to new one.
`copyparams`(link)	Copies all parameters from given link.
`disable_update`()	Disables update rules of all parameters under the link hierarchy.
`enable_update`()	Enables update rules of all parameters under the link hierarchy.
`init_scope`()	Creates an initialization scope.
`links`([skipself])	Returns a generator of all links under the hierarchy.
`namedlinks`([skipself])	Returns a generator of all (path, link) pairs under the hierarchy.
`namedparams`([include_uninit])	Returns a generator of all (path, param) pairs under the hierarchy.
`params`([include_uninit])	Returns a generator of all parameters under the link hierarchy.
`register_persistent`(name)	Registers an attribute of a given name as a persistent value.
`serialize`(serializer)	Serializes the link object.
`to_cpu`()	Copies parameter variables and persistent values to CPU.
`to_gpu`([device])	Copies parameter variables and persistent values to GPU.
`zerograds`()	Initializes all gradient arrays by zero.

Attributes

`update_enabled`	`True` if at least one parameter has an update rule enabled.
`within_init_scope`	True if the current code is inside of an initialization scope.
`xp`	Array module for this link.

1.5.6 Utilities

Index

Symbols

A

B

C

E

G

M

N

S

T

W