ATOM3D

ATOM3D Developers

Mar 02, 2023

GETTING STARTED WITH ATOM3D

1 Quick Start 1

2 ATOM3D data formats 5

3 Using ATOM3D datasets 7

4 Creating new datasets 11

5 Machine learning with ATOM3D 13

6 Contributing to ATOM3D 19

CHAPTER

ONE

QUICK START

1.1 Introduction

ATOM3D aims to facilitate the development of novel machine learning methods on three-dimensional molecular structure by providing several curated benchmark datasets across a variety of tasks in molecular and strucural biology. This package provides a set of standardized functions and methodologies for interacting with provided datasets as well as preparing new 3D molecular datasets.

1.2 Installation

1.2.1 Install using pip

```
pip install atom3d
```

1.2.2 Install from source

To install, first clone the ATOM3D repository:

```
git clone https://github.com/drorlab/atom3d
```

To install with base dependencies:

```
make requirements
```

To install with RDKit (needed for processing small molecule files, e.g. SDF/MOL2), install within conda:

```
conda create -n atom3d python=3.6 pip rdkit
conda activate atom3d
make requirements
```

1.2.3 Model-specific dependencies

The standard installation described above lets you use all the data loading and processing functions included in ATOM3D. To use the specific machine learning models, additional dependencies can be necessary. We describe these in the machine learning section.

1.3 Usage

1.3.1 Downloading datasets

All datasets can be downloaded in LMDB format from atom3d.ai, or using the Python API:

```
>>> import atom3d.datasets.datasets as da
>>> da.download_dataset('lba', PATH_TO_DATASET) # Download LBA dataset
```

See *Using ATOM3D datasets* for more details.

1.3.2 Loading datasets

```
>>> import atom3d.datasets as da
>>> dataset = da.load_dataset(PATH_TO_DATASET, 'lmdb') # Load LMDB format dataset
>>> print(len(dataset))  # Print length
>>> print(dataset[0].keys()) # Print keys stored in first structure
```

1.4 Frequently Asked Questions

1.4.1 1. What pre-curated datasets are available through ATOM3D?

ATOM3D currently contains eight datasets, spanning molecular structure, function, interaction, and design tasks:

- *Small Molecule Properties (SMP)*

  Predicting physico-chemical properties of small molecules is a common task in medicinal chemistry and materials design. SMP is based on the QM9 dataset, which contains structures and energetic, electronic, and thermodynamic properties for 134,000 stable small organic molecules, obtained from quantum-chemical calculations.

- *Protein Interface Prediction (PIP)*

  Proteins interact with each other in many scenarios—for example, our antibody proteins recognize diseases by binding to antigens. A critical problem in understanding these interactions is to identify which amino acids of two given proteins will interact upon binding. The PIP dataset contains structures from the Database of Interacting Protein Structures (DIPS), a comprehensive dataset of protein complexes mined from the PDB, and the Docking Benchmark 5 (DB5), a smaller gold standard dataset.

- *Residue Identity (RES)*

  Understanding the structural role of individual amino acids is important for engineering new proteins. We can understand this role by predicting the substitutabilities of different amino acids at

a given protein site based on the surrounding structural environment. The RES dataset consists of atomic environments extracted from nonredundant structures in the PDB.

- *Mutation Stability Prediction (MSP)*

  Identifying mutations that stabilize a protein's interactions is a key task in designing new proteins. Experimental techniques for probing these are labor intensive, motivating the development of efficient computational methods. MSP contains structures from the SKEMPI dataset of protein-protein interactions, with each mutation computationally modeled into the structure.

- *Ligand Binding Affinity (LBA)*

  Most therapeutic drugs and many molecules critical for biological signaling take the form of small molecules. Predicting the strength of the protein-small molecule interaction is a challenging but crucial task for drug discovery applications. LBA contains structures from the "refined set" of PDBBind, a curated database containing protein-ligand complexes from the PDB and their corresponding binding strengths.

- *Ligand Efficacy Prediction (LEP)*

  Many proteins switch on or off their function by changing shape. Predicting which shape a drug will favor is thus an important task in drug design. LEP contains a curated set of proteins from several families with both "active" and "inactive" state structures, with 527 small molecules with known activating or inactivating function modeled in using the program Glide.

- *Protein Structure Ranking (PSR)*

  Assessing the quality of a specific 3D protein conformation is a crucial aspect of computational protein structure prediction. PSR contains data from the Critical Assessment of Structure Prediction (CASP), a blind international competition for predicting protein structure.

- *RNA Structure Ranking (RSR)*

  Similar to proteins, RNA plays major functional roles (e.g., gene regulation) and can adopt well-defined 3D shapes. However the problem is data-poor, with only a few hundred known structures. PSR contains candidate structures for the first 21 released RNA Puzzle challenges, a blind structure prediction competition for RNA.

1.4.2 2. Do I have to use the provided train/val/test splits for ATOM3D datasets?

No, you may create your own splitting functions and apply them to any dataset. Please see *Using ATOM3D datasets* for more details.

1.4.3 3. What kind of utility functions exist in ATOM3D?

There are functions available for performing many common tasks on macromolecular structure. See the *usage examples* for some common use cases, and explore the API documentation to find specific functions.

If we are missing a function you think would be useful, please consider *contributing*!

1.4.4 4. Can I contribute datasets and models back to ATOM3D?

Yes! We are happy to accept new datasets and models! See *contributing* for details.

1.5 Reference

If you use ATOM3D in your work, please cite our preprint:

Townshend, R. J. L., Vögele, M., Suriana, P., Derry, A., Powers, A., Laloudakis, Y., Balachandar, S., Anderson, B., Eismann, S., Kondor, R., Altman, R. B., Dror, R. O. (2020). ATOM3D: Tasks On Molecules in Three Dimensions. *arXiv:2012.04035*. http://arxiv.org/abs/2012.04035.

For specific datasets, please also cite the respective original source(s) as specified in the preprint.

CHAPTER

TWO

ATOM3D DATA FORMATS

2.1 The Dataset object

ATOM3D uses a standardized data format for 3D molecular structures, designed to maximize flexibility and efficiency without losing important structural information. The main way to interact with these datasets is through the Dataset objects in `atom3d.datasets.datasets`. These are essentially PyTorch datasets in which each item consists of several key-data pairs.

Each item (molecular structure) in an ATOM3D dataset contains at minimum the following keys:

- `id` (str): unique identifier for structure
- `file_path` (str): path to file from which structure was derived
- `atoms` (DataFrame): data describing 3D coordinates and atomic information

Depending on the dataset, other important data may be provided under additional keys:

- `bonds` (DataFrame): bond data is provided for small molecule structures
- `scores`/`labels`: task-specific scores or labels for each structure

2.1.1 The `atoms` dataframe

The `atoms` dataframe contains the bulk of the data for a 3D structure. The dataframe is constructed in a hierarchical manner to enable consistent data processing across datasets.

The first column is the `ensemble`, which represents the highest level of structure, for example the PDB entry for a protein.

The second column, the `subunit`, is the subset of the ensemble representing the specific units of structure for the task/dataset. For example, for the protein structure ranking (PSR) task, each candidate structure for a given target protein would be assigned a unique subunit (here, the ensemble is the target protein).

The remainder of the columns contain the structural information. These follow the convention of the PDB file format: `model` - `chain` - `hetero` - `insertion_code` - `residue` - `segid` - `resname` - `altloc` - `occupancy` - `bfactor` - `x` - `y` - `z` - `element` - `name` - `fullname` - `serial_number`.

For small molecules and nucleic acids, many of these columns are unused.

2.1.2 The `bonds` dataframe

The `bonds` dataframe contains the covalent bonding information for a molecular structure, especially for small molecules. Each row of the dataframe represents a bond and contains three columns:

- `atom1`: index of the first atom in the bond (corresponding to the `serial_number` field in the `atoms` dataframe)
- `atom2`: index of the second atom in the bond (corresponding to the `serial_number` field in the `atoms` dataframe)
- `type`: bond type, encoded as Double (single bond = 1.0, double bond = 2.0, triple bond = 3.0, aromatic bond = 1.5).

CHAPTER

THREE

USING ATOM3D DATASETS

All datasets in ATOM3D are provided in standardized LMDB format. LMDB allows for compressed, fast, random access to your structures, all within a single database.

3.1 Downloading LMDB datasets

All datasets are hosted on Zenodo, and the links to download raw and split datasets in LMDB format can be found at atom3d.ai. Alternatively, you can use the Python API:

```
>>> import atom3d.datasets as da
>>> da.download_dataset('lba', TARGET_PATH, split=SPLIT_NAME)
```

See the ATOM3D website or the *FAQ* for more information about the datasets available.

3.2 Loading datasets in Python

After downloading or *creating* an LMDB dataset, it is easy to load it into Python using ATOM3D. You can also load a dataset from a directory of files in any supported structural data format (currently PDB, SDF, XYZ, and Rosetta silent files). The resulting object acts just like a PyTorch Dataset, with a length and entries that can be accessed or iterated on. See *ATOM3D data formats* for details on the format of each data entry.

```
import atom3d.datasets as da

dataset = da.load_dataset(PATH_TO_INPUT_DIR, {'lmdb', 'pdb','silent','sdf','xyz','xyz-
↪gdb'})
print(len(dataset))  # Print length
print(dataset[0].keys()) # Print keys stored in first structure
```

3.3 Filtering datasets

By default, datasets contain all atoms in each molecular structure. However, many applications may require filtering the structure to remove undesirable atoms or focus on a specific region of the structure. This is easy to do by defining filter functions which operate on the `atoms` dataframe of a dataset item, returning a filtered version of the same dataframe. Several such filter functions are predefined in `atom3d.filters`.

For example, to remove non-standard residues from all proteins in a dataset:

```
from atom3d.filters import filters
from atom3d.data.example import load_example_dataset

dataset = load_example_dataset()
for struct in dataset:
  struct['atoms'] = filters.standard_residue_filter(struct['atoms'])
```

It is also possible to combine multiple filters with `atom3d.filters.filters.compose()`. For example, to use only the first chain of a protein *and* remove non-standard residues:

```
>>> from atom3d.data.example import load_example_dataset
>>> from atom3d.filters import filters
>>> dataset = load_example_dataset()
>>> struct = dataset[0]
>>> filter1 = filters.single_chain_filter
>>> filter2 = filters.standard_residue_filter
>>> filter_fn = filters.compose(filter1, filter2)
>>> struct['atoms'].shape
(5220, 20)
>>> struct['atoms'] = filter_fn(struct['atoms'])
>>> struct['atoms'].shape
(2568, 20)
```

These functions can also be readily extended by defining wrappers that return a filter function for a particular application. For example, the function `atom3d.filters.sequence.form_seq_filter_against()` creates a filter function that removes structures with greater than some sequence identity to any structure in a specified dataset (e.g. to filter train examples that are too similar to the test set).

```
from atom3d.filters.sequence import form_seq_filter_against
from atom3d.datasets.datasets import LMDBDataset

train_dataset = LMDBDataset(TRAIN_PATH)
test_dataset = LMDBDataset(TEST_PATH)

filter_fn = form_seq_filter_against(test_dataset, 0.3)

for struct in train_dataset:
  struct['atoms'] = filter_fn(struct['atoms']) # returns empty dataframe if a match␣
↪is found in test set
```

To automatically apply a filter to a dataset on the fly as each example is loaded, you can convert it to a transform function and pass it to any Dataset using the `transform` argument.

```
from atom3d.filters import filters
from atom3d.datasets.datasets import LMDBDataset

filter_fn = filters.standard_residue_filter
transform_fn = filters.filter_to_transform(filter_fn) # convert filter function to␣
↪transform function
dataset = LMDBDataset(PATH, transform=transform_fn) # load dataset and apply transform
```

3.4 Splitting datasets

For most machine learning applications, the datasets will need to be split into train/validation/test subsets. Because the desired splitting methodology varies depending on the molecule type and the application, the standard way to split datasets in ATOM3D is by using pre-computed sets of indices into the dataset. These indices can be computed arbitrarily using any splitting function that takes in a dataset and returns indices to include in the train, validation, and test sets.

The `atom3d.splits.splits.split()` function then takes in a dataset and the split indices and returns the three corresponding sub-datasets (in the same format as the original dataset):

```
import atom3d.splits.splits as spl

train_dataset, val_dataset, test_dataset = spl.split(dataset, indices_train, indices_
↪val, indices_test)
```

3.4.1 Using standard splitting criteria

ATOM3D provides splitting functions for many commonly used splitting methodologies in the `atom3d.splits.splits` module.

- **Split randomly**

  The simplest splitting method is to split the dataset at random.

- **Split by sequence identity (proteins)**
- **Split by scaffold (small molecules)**
- **Split by year**

3.4.2 Defining your own splitting criteria

- **Split by cluster/group membership**
- **Split by cluster/group size**

3.5 Examples

The following examples illustrate some useful functionalities of ATOM3D using a small mock dataset.

```
>>> from atom3d.data.example import load_example_dataset
>>> dataset = load_example_dataset()
```

1. **Get coordinates of all atoms in a structure.**

```
>>> import atom3d.util.formats as fo
>>> struct = dataset[0] # get first structure in dataset
>>> atoms_df = struct['atoms'] # load atom data for structure
>>> coords = fo.get_coordinates_from_df(atoms_df)
>>> coords.shape
(2568, 3)
```

2. **Get protein sequences from a structure.**

```
>>> import atom3d.protein.sequence as seq
>>> struct = dataset[0] # get first structure in dataset
>>> atoms_df = struct['atoms'] # load atom data for structure
>>> chain_sequences = seq.get_chain_sequences(atoms_df)
>>> chain_sequences # Contains sequences for all chains/monomers, identified by tuple␣
↪of (ensemble, subunit, structure, model, chain)
[(('11as.pdb', '0', '11as.pdb', '1', 'A'), 'AYIAKQRQISFVKS...PAAVRESVPSLLN')]
```

3. **Extract all atoms within 5 Angstroms of a ligand**

In this example, the ligand is assumed to be stored as a subunit in the atoms dataframe, under the label "LIG". In practice, the ligand could be stored in different way (e.g. in a separate dataframe or under a different label), depending on how the dataset was constructed.

```
>>> from atom3d.filters.filters import distance_filter
>>> import atom3d.util.formats as fo
>>> struct = dataset[0] # get first structure in dataset
>>> atoms_df = struct['atoms'] # load atom data for structure
>>> lig_coords = fo.get_coordinates_from_df(atoms_df[atoms_df['subunit']=='LIG']) #␣
↪get coords of ligand
>>> df_filtered = distance_filter(atoms_df, lig_coords, dist=5.0)
```

CHAPTER

FOUR

CREATING NEW DATASETS

In addition to the eight pre-curated datasets, you can also create your own datasets in the same standardized LMDB format. Currently, we support creating LMDB datasets from a set of PDB files, SDF files, silent files, or xyz files.

4.1 Create a dataset from input files

Assuming a directory containing all of the files you wish to process, you can create a new LMDB dataset from the command line:

```
python -m atom3d.datasets PATH_TO_INPUT_DIR PATH_TO_LMDB_OUTPUT --filetype
↪{pdb,silent,xyz,xyz-gdb}
```

You can also load the dataset first in Python before writing it to LMDB format using the `atom3d.datasets.datasets` module:

```
import atom3d.datasets.datasets as da

# Load dataset from directory of PDB files
dataset = da.load_dataset(PATH_TO_INPUT_DIR, 'pdb')
# Create LMDB dataset from PDB dataset
da.make_lmdb_dataset(dataset, PATH_TO_LMDB_OUTPUT)
```

4.2 Modify a dataset (add labels etc.)

To modify a dataset you can load it in Python and define the modification via the `transform` option. The most common modification is adding labels, which are usually provided separate from PDB or SDF files. In the following example, we assume that they are saved in CSV files with the same names as the corresponding PDB files.

```
import os
import pandas as pd
import atom3d.datasets.datasets as da

def add_label(item):
    # Remove the file ending ".pdb" from the ID
    name = item['id'][:-4]
    # Get label data
    label_file = os.path.join(PATH_TO_LABELS_DIR, name+'.csv')
    # Add label data in form of a data frame
    item['label'] = pd.read_csv(label_file)
    return item
```

(continues on next page)

(continued from previous page)

```
# Load dataset from directory of PDB files
dataset = da.load_dataset(PATH_TO_INPUT_DIR, 'pdb', transform=add_label)

# Create LMDB dataset from PDB dataset
da.make_lmdb_dataset(dataset, PATH_TO_LMDB_OUTPUT)
```

You can flexibly use the *transform* option to modify any aspect of a dataset. For example, if you want to shift all structures in x direction, use the following function:

```
def my_transformation(item):
    item['atoms']['x'] += 3
    return item
```

4.3 Split a dataset

Once you have processed your dataset, you probably want to split it in training, validation, and test sets. In the following example, we assume that we want to split the dataset generated above according to a predefined split and that the IDs for the structures that belong in each dataset are defined in the files *train.txt*, *valid.txt* and *test.txt*.

```
import atom3d.splits.splits as spl

# Load split values
tr_values = pd.read_csv('train.txt',header=None)[0].tolist()
va_values = pd.read_csv('valid.txt',header=None)[0].tolist()
te_values = pd.read_csv('test.txt',header=None)[0].tolist()

# Create splits
split_ds = spl.split_by_group(dataset,
                              value_fn = lambda x: x['id'],
                              train_values = tr_values,
                              val_values   = va_values,
                              test_values  = te_values)

# Create split LMDB datasets
for s, split_name in enumerate(['training','validation','test']):
    # Create the output directory if it does not exist yet
    split_dir = os.path.join(PATH_TO_LMDB_OUTPUT, split_name)
    os.makedirs(split_dir, exist_ok=True)
    # Create LMDB dataset for the current split
    da.make_lmdb_dataset(split_ds[s], split_dir)
```

There are many ways to split datasets and we provide functions for many of them in the the `atom3d.splits` module. They are described in more detail here.

CHAPTER

FIVE

MACHINE LEARNING WITH ATOM3D

ATOM3D makes it easy to train any machine learning model on 3D biomolecular structure data. All ATOM3D datasets are built on top of PyTorch Datasets, making it simple to create dataloaders that work with almost any model architecture out of the box. We provide dataloaders for all pre-curated datasets, as well as some base model architectures for three major classes of deep learning methods for 3D molecular learning: graph neural networks (GNNs), 3D convolutional neural networks (3DCNNs), and equivariant neural networks (ENNs). Please see our paper for more details on the specific choice of architecture for each task.

5.1 Preparing data for training

The structures in ATOM3D Datasets are represented in dataframe format (see *ATOM3D data formats*), which need to be processed into numeric tensors before they are ready for training. One way of doing this is to define a dataloader that yields dataframes, which are then converted into tensors (e.g. graphs or voxelized cubes) after they are loaded. However, this makes automatic batching somewhat complicated, requiring custom functions for collating dataframes into minibatches.

To avoid this, ATOM3D enables the conversion to tensors to happen as the data retrieved from the dataset (i.e. the items returned by a Dataset's `getitem` method contain tensors, rather than dataframes). Dataloading and minibatching is then simple to do using Pytorch's `DataLoader` class (or the Pytorch-Geometric equivalent for graphs).

The conversion itself happens through the *transform* function, which is passed to the Dataset class on instantiation. Transform functions for converting dataframes to graphs (e.g. for GNNs) or voxelized cubes (e.g. for 3D CNNs) are provided in `atom3d.util.transforms`, but any arbitrary transformation function can be defined similarly.

5.2 Base models

For general use, we provide base versions of each model type. These models may be useful as proof-of-concept testing on a new dataset, to provide a strong baseline for benchmarking a specially engineered model architecture, or as a template for the design of new model architectures. The base models provided are the following:

- **GCN** (`atom3d.models.gnn.GCN`)

  A simple GNN consisting of five layers of graph convolutions as defined by Kipf and Welling (2017). Each GCN layer is followed by batch normalization and a ReLU nonlinearity. These layers will learn an embedding vector for each node in the network, but it is often necessary to reduce this to a single vector for classification or regression. We provide two ways to do this: (1) global mean pooling over all nodes (default), or (2) extract the embedding of a single node in the graph supplied by the `select_idx` argument.

  This network and all other GNNs are implemented using the pytorch-geometric library. This package must be installed separately, and data passed into the model should be in the format used by pytorch-geometric. For converting Dataset items to graphs, see `atom3d.util.graph`.

- **CNN3D** (`atom3d.models.cnn.CNN3D`)

  A simple convolutional network consisting of six layers of 3D convolutions, each followed by batch normalization, ReLU activation, and optionally dropout. This network uses strided convolutions for downsampling. The desired input and output dimensionality must be specified when instantiating the model.

  The input data is expected to be a voxelized cube with several feature channels, represented as a tensor with 5 dimensions: (batch_size, in_dimension, box_size, box_size, box_size). For converting Dataset items to voxelized tensors, see `atom3d.util.cnn`.

- **ENN** (`atom3d.models.enn.ENN`)

  This network and all ENNs based on it are implemented using an adapted version of the Cormorant package. Make sure to install it from the Dror Lab fork.

- **FeedForward** (`atom3d.models.ff.FeedForward`)

  A basic feed-forward neural network (multi-layer perceptron), with tunable number of hidden layers and layer dimensions. In many cases, the 3D learning methods above are most useful as feature extractors to transform a molecular structure to a single vector, or embedding. For classification and regression tasks, it is then necessary to transform this vector into the desired output dimensionality. Although any method could be used instead, this feed-forward network is simple but flexible, and easy to plug into any machine learning pipeline.

5.3 Model-specific installation instructions

The standard installation described in the introduction lets you use all the data loading and processing functions included in ATOM3D. To use the specific machine learning models, additional dependencies can be necessary. Here we describe which ones you need for each type of model.

5.3.1 3DCNN

No additional dependencies are needed to use 3D convolutional neural networks.

5.3.2 GNN

The graph neural networks in ATOM3D are based on PyTorch Geometric

In order to use it, you should install ATOM3D in a dedicated environment, defining the correct version of the CUDA toolkit (here: 10.2):

```
conda create --name geometric -c pytorch -c rdkit pip rdkit pytorch=1.5␣
↪cudatoolkit=10.2
conda activate geometric
pip install atom3d
```

Then install PyTorch Geometric by running:

```
pip install torch-scatter==latest+${CUDA} -f https://pytorch-geometric.com/whl/torch-
↪1.5.0.html
pip install torch-sparse==latest+${CUDA} -f https://pytorch-geometric.com/whl/torch-1.
↪5.0.html
pip install torch-cluster==latest+${CUDA} -f https://pytorch-geometric.com/whl/torch-
↪1.5.0.html
```

(continues on next page)

(continued from previous page)

```
pip install torch-spline-conv==latest+${CUDA} -f https://pytorch-geometric.com/whl/
↪torch-1.5.0.html
pip install torch-geometric
```

where *${CUDA}* should be replaced by either *cpu*, *cu92*, *cu101* or *cu102* depending on your PyTorch installation (*cu102* if installed as above).

If you do not know your CUDA version, you can find out via:

```
nvcc --version
```

5.3.3 ENN

The equivariant networks in ATOM3D are based on Cormorant.

In order to use it, you should install ATOM3D in a dedicated environment, defining the correct version of the CUDA toolkit (here: 10.1):

```
conda create --name cormorant python=3.7 pip scipy pytorch cudatoolkit=10.1 -c pytorch
conda activate cormorant
pip install atom3d
```

If you do not know your CUDA version, you can find out via:

```
nvcc --version
```

The Cormorant fork used for this project can be cloned directly from the git repo using:

```
cd ~
git clone https://github.com/drorlab/cormorant.git
```

You can currently only install it in development mode by going to the directory with setup.py and running:

```
cd cormorant
python setup.py develop
```

5.4 Task-specific models and dataloaders

Many datasets and tasks require specific model architectures (e.g. paired or multi-headed networks), and thus require custom-built dataloaders to process the data in the correct manner. We provide custom dataloaders and models for each pre-curated dataset in the `atom3d.datasets` module. A brief description of each is provided below; for more details and motivation please see the ATOM3D paper.

- **SMP** (`atom3d.datasets.smp.models`)
- **PIP** (`atom3d.datasets.pip.models`)
- **RES** (`atom3d.datasets.res.models`)
- **MSP** (`atom3d.datasets.msp.models`)
- **LBA** (`atom3d.datasets.lba.models`)
- **LEP** (`atom3d.datasets.lep.models`)
- **PSR** (`atom3d.datasets.psr.models`)

- **RSR** (`atom3d.datasets.rsr.models`)

5.5 Examples

1. **Train base GCN model on a protein dataset, with default parameters.**

In this example, the dataset contains labels for each example under the `label` key. These are assumed to be binary labels applied to the entire graph, rather than to a specific node.

The underlying dataset contains dataframes in the `atoms` field, as with all ATOM3D Datasets, but for training we must convert these to tensors representing each graph. This is done via the *transform* function, which enables automatic batching via `DataLoader` objects (either standard Pytorch or Pytorch-Geometric).

We are assuming a binary classification problem, and using the GCN as a feature extractor. Therefore, we need a model to transform from the feature representation to the output prediction (a single value). This example uses a simple feed-forward neural network with one hidden layer.

```
# pytorch imports
import torch
import torch.nn as nn
from torch_geometric.data import Data, DataLoader

# atom3d imports
import atom3d.datasets.datasets as da
import atom3d.util.graph as gr
import atom3d.util.transforms as tr
from atom3d.models.gnn import GCN
from atom3d.models.mlp import MLP

# define training hyperparameters
learning_rate=1e-4
epochs = 5
feat_dim = 128
out_dim = 1

# Load dataset (with transform to convert dataframes to graphs) and␣
↪initialize dataloader
dataset = da.load_dataset('data/test_lmdb', 'lmdb', transform=tr.
↪GraphTransform(atom_key='atoms', label_key='label'))
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)

# get number of input features from first graph
for batch in dataloader:
    in_dim = batch.num_features
    break

# GCN feature extraction module
feat_model = GCN(in_dim, feat_dim)
# Feed-forward output module
out_model = MLP(feat_dim, [64], out_dim)

# define optimizer and criterion
params = [x for x in feat_model.parameters()] + [x for x in out_model.
↪parameters()]
optimizer = torch.optim.Adam(params, lr=1e-4)
criterion = nn.BCEWithLogitsLoss()
```

(continues on next page)

(continued from previous page)

```
# Training loop
for epoch in range(epochs):
    for batch in dataloader:
        # labels need to be float for BCE loss
        labels = batch.y.float()
        # calculate 128-dim features
        feats = feat_model(batch.x, batch.edge_index, batch.edge_attr, batch.
↪batch)
        # calculate predictions
        out = out_model(feats)
        # compute loss and backprop
        loss = criterion(out.view(-1), labels)
        loss.backward()
        optimizer.step()
    print('Epoch {}: train loss {}'.format(epoch, loss))
```

2. **Train an equivariant model on ligand binding affinity.**

We provide several example training scripts (+ model implementations and dataloaders) in a dedicated folder in the GitHub repository. One particularly educative example is training of an equivariant neural network on ligand binding affinity data.

CHAPTER

SIX

CONTRIBUTING TO ATOM3D

You can support the mission of ATOM3D in several different ways: by reporting bugs and flaws, by requesting new features that would benefit your work, by contributing a dataset, by contributing a new class of model, or by writing or improving code to handle them.

Machine learning on three-dimensional molecular structure is only at its beginning. The datasets and methods here are only a snapshot of the current state of the field and we see ATOM3D as a platform to curate and share them with the community. See below for the most common ways to help us in this endeavour. Finally, we are always happy hear about your machine learning success stories.

6.1 Report a bug or request a feature

ATOM3D is open-source and available on Github. Please submit issues or requests using the issue tracker.

6.2 Add a dataset

We distribute datasets in a JSON-encoded LMDB format which is described in the corresponding documentation. The ATOM3D library provides many tools to prepare such datasets from commonly used datatypes in the field (PDB, SDF, etc.). Ideally, you would provide the code to prepare the datasets along with them. See the dataset-specific folders in `atom3d.datasets` for examples.

Once you have prepared the dataset, please contact us so we can add it to ATOM3D.

6.3 Add a model

We are still figuring out the best way to organize the various models.

Please contact us if you have a model that you would like to be added to ATOM3D.

6.4 Add new functionality

We welcome any kind of contributions to improve or expand the ATOM3D code. In particular, we are interested in readers for new data formats and new ways to split, filter, or transform datasets. ATOM3D is maintained on Github so you can fork it and create a pull request. Please make sure to properly test your contribution before the request. For large or complicated contributions, please get in contact so we can coordinate them with you. We explain two of the most common and straightforward cases (file readers and splits) below:

6.4.1 New file readers

To add the possibility to read a new type of datasets, add a class to the file `atom3d/datasets/datasets.py`. This class should inherit the PyTorch Dataset class and be compatible with the internal *ATOM3D data format*. Use one of the existing classes like `LMDBDataset(Dataset)`, `PDBDataset(Dataset)` or `SDFDataset(Dataset)` as a template.

6.4.2 New splitting functions

To add a new way to split a dataset, you can build on the generic split functions in `atom3d.splits.splits`. For example:

- The function `split_by_group` assigns elements to splits based on pre-defined lists of values. A "value function", defining the value in the data entry to which the values in the list correspond, has to be provided as an argument.
- Similarly, `split_by_group_size` assigns elements so that the largest groups (i.e. most common examples) are in the training set and the smallest groups (i.e. less common examples) are in the test set.

You can create more specific split functions from them by defining the value function, as we do in the following examples (which are already part of ATOM3D):

```
from functools import partial
split_by_year = partial(split_by_group, value_fn=lambda x: x['year'])
split_by_scaffold = partial(split_by_group_size, value_fn=lambda x: x[
↪'scaffold'])
```