pyuul

Release 0.2.0

Gabriele Orlando

Aug 30, 2022

CONTENTS

pyUUL is a Python library designed to process 3D-structures of macromolecules, such as PDBs, translating them into fully differentiable data structures. These data structures can be used as input for any modern neural network architecture. Currently, the user can choose between three different types of data representation: voxel-based, surface point cloud and volumetric point cloud.

CHAPTER

ONE

INSTALLATION GUIDE

1.1 How do I set it up?

You can install PyUUL either from the bitbucket (https://bitbucket.org/grogdrinker/pyuul/) repository, from a pypi package (https://pypi.org/project/pyuul/) or from a conda package (https://anaconda.org/anaconda/pyuul).

1.2 Installing PyUUL with pip:

If you want to install PyUUL via Pypi, just open a terminal and type:

```
pip install pyuul
```

and all the required dependencies will be installed automatically. Congrats, you are ready to rock!

1.3 Installing PyUUL with Anaconda

PyUUL can be installed simply via conda by typing in the terminal:

```
conda install -c grogdrinker pyuul
```

again, all the required dependencies will be installed automatically.

1.4 Installing PyUUL from source:

If you want to install PyUUL from this repository, you need to install the dependencies first.

Install `pytorch >= 1.0` with the command: `conda install pytorch -c pytorch` or refer to `pytorch` website https://pytorch.org.

Install the remaining requirements with the command: `conda install scipy numpy scikit-learn`.

You can remove this environment at any time by typing: `conda remove -n pyuul --all`.

Finally, you can clone the repository with the following command:

```
git clone https://bitbucket.org/grogdrinker/pyuul/
```

CHAPTER

TWO

INTRODUCTION

2.1 What is a biological macromolecule?

If you are looking at this documentation, you probably already know what a macromolecule is. Macromolecules are an essential element of cell life and are extremely heterogeneous. They can catalyze chemical reactions (i.e., proteins), transfer information among generations (i.e., DNA) or constitute the cell itself (i.e., lipids, carbohydrates and proteins).

2.2 Machine learning in structural biology

The study of the 3D structure of these macromolecules has dramatically increased in importance during the last years. The amount of structural data available in public databases such as the protein data bank is increasing at an unprecedented rate. This has not only the effect of increasing our knowledge about structural biology, but it has also opened the door to the application of machine learning algorithms to biological structural data.

2.3 The technological gap

However, biological structures are often hard to handle. From public datasets, you can download atomic coordinates, but this type of data is not directly usable in standard machine learning algorithms. For this reason, structural bioinformatics has been lacking behind in machine learning with respect to other fields such as computer vision, where modern neural network architectures, such as 3D-convolutional neural network or transformers, are largely used.

2.4 What is PyUUL?

To overcome this problem we built PyUUL, a pytorch library that can transform biological structures in differentiable 3D objects that are suitable for machine learning algorithms developed for computer vision. This library therefore greatly increases the number of neural network architectures applicable to structural bioinformatics. Currently, the user can choose between three different types of data representation: voxel-based, surface point cloud and volumetric point cloud. If you want to learn more about PyUUL, you can read the manuscript at:

2.5 Possible applications in the scientific world

PyUUL can be used to import machine learning algorithms from computer vision to structural bioinformatics. Some of the algorithms listed below might be good candidates for future works:

- **Protein classification and domain identification** Point networks are used for point cloud (one of the representation PyUUL provides) classification and segmentation. This means this architecture might be suitable to find and classify structural sub-modules (domains) of proteins (https://openaccess.thecvf.com/content_cvpr_2017/html/Qi_PointNet_Deep_Learning_CVPR_2017_paper.html).
- **End-to-end protein structure prediction** A volumetric representation of proteins might be used as an additional prediction step in end to end protein structure prediction, adding a 3d-convolutional branch to the network and expanding the work of Mohammed AlQuraishi (https://www.biorxiv.org/content/10.1101/265231v1.full.pdf).
- **Protein structure superimposition**, algorithms of point cloud registration, such as the one described by Yang et al (https://arxiv.org/abs/2001.07715), might be used to perform alignment-free structural superimposition.

CHAPTER

THREE

QUICKSTART

This quickstart example shows how to easily build 3D representations of biological structures with PyUUL.

3.1 Importing libraries

You can use the following code to import all the necessary libraries used in this example:

```
from pyuul import VolumeMaker # the main PyUUL module
from pyuul import utils # the PyUUL utility module
import time,os,urllib # some standard python modules we are going to use
```

3.2 Collecting the data

We now need some protein structures to test. You can download them from the PDB website (https://www.rcsb.org/). In this case, we will get them using urllib, a standard and preinstalled library of python. This will allow a painless data collection with a simple copy-paste of the code. Lets fetch a couple of structures:

```
os.mkdir('exampleStructures')
urllib.request.urlretrieve('http://files.rcsb.org/download/101M.pdb', 'exampleStructures/
↪101m.pdb')
urllib.request.urlretrieve('http://files.rcsb.org/download/5BMZ.pdb', 'exampleStructures/
↪5bmz.pdb')
urllib.request.urlretrieve('http://files.rcsb.org/download/5BOX.pdb', 'exampleStructures/
↪5box.pdb')
```

3.3 Parsing the structures

Next step is to parse the information of the structures. In order to do so, we can use the utility module of PyUUL:

```
coords, atname = utils.parsePDB("exampleStructures/") # get coordinates and atom names
atoms_channel = utils.atomlistToChannels(atname) # calculates the corresponding channel␣
↪of each atom
radius = utils.atomlistToRadius(atname) # calculates the radius of each atom
```

3.4 Volumetric representations

Now we have everything we need to get the volumetric representations. We can build the main PyUUL objects on CPU with:

```
device = "cpu" # runs the volumes on CPU
VoxelsObject = VolumeMaker.Voxels(device=device,sparse=True)
PointCloudSurfaceObject = VolumeMaker.PointCloudVolume(device=device)
PointCloudVolumeObject = VolumeMaker.PointCloudSurface(device=device)
```

or on GPU with:

```
device = "cuda" # runs the volumes on GPU (you need a cuda-compatible GPU computer for␣
↪this)
VoxelsObject = VolumeMaker.Voxels(device=device,sparse=True)
PointCloudSurfaceObject = VolumeMaker.PointCloudVolume(device=device)
PointCloudVolumeObject = VolumeMaker.PointCloudSurface(device=device)
```

Next, we move everything to the right device:

```
coords = coords.to(device)
radius = radius.to(device)
atoms_channel = atoms_channel.to(device)
```

Finally, we obtain the volumetric representation of the proteins with:

```
SurfacePoitCloud = PointCloudSurfaceObject(coords, radius)
VolumePoitCloud = PointCloudVolumeObject(coords, radius)
VoxelRepresentation = VoxelsObject(coords, radius, atoms_channel)
```

CHAPTER

FOUR

PYUUL WITH SMALL MOLECULES

This guide shows how handle small molecules in PyUUL. For this purpose we will use the parser for SDF files provided by PyUUL.

4.1 Importing libraries

You can use the following code to import all the necessary libraries used in this example:

```
from pyuul import VolumeMaker # the main PyUUL module
from pyuul import utils # the PyUUL utility module
import os,urllib # some standard python modules we are going to use
```

4.2 Using the parser for small molecules (SDF files)

We now need some small molecule structures to test. You can download them from the PDB website (https://www.rcsb.org/). As in the quickstart guide, we will get them using urllib, a standard and preinstalled library of python. This will allow a painless data collection with a simple copy-paste of the code. Lets fetch a couple of structures:

```
os.mkdir('exampleStructures')
urllib.request.urlretrieve('https://files.rcsb.org/ligands/view/GTP_ideal.sdf',
↪'exampleStructures/gtp.sdf')
urllib.request.urlretrieve('https://files.rcsb.org/ligands/view/CFF_ideal.sdf',
↪'exampleStructures/cff.sdf')
urllib.request.urlretrieve('https://files.rcsb.org/ligands/view/CLR_ideal.sdf',
↪'exampleStructures/clr.sdf')
```

Next step is to parse the information of the structures. In order to do so, we can use the utility module of PyUUL. The syntax is the same of when dealing with PDBs, but in this case we use the function parseSDF:

```
coords, atname = utils.parseSDF("exampleStructures/") # get coordinates and atom names
atoms_channel = utils.atomlistToChannels(atname) # calculates the corresponding channel␣
↪of each atom
radius = utils.atomlistToRadius(atname) # calculates the radius of each atom
```

From now on, everything works the same way of PDBs. We can build the main PyUUL objects on CPU with:

```
device = "cpu" # runs the volumes on CPU. use "cuda" for GPU. Pytorch needs to be␣
↪installed with cuda support in this case.
```

(continues on next page)

(continued from previous page)

```
VoxelsObject = VolumeMaker.Voxels(device=device,sparse=True)
PointCloudSurfaceObject = VolumeMaker.PointCloudVolume(device=device)
PointCloudVolumeObject = VolumeMaker.PointCloudSurface(device=device)
```

We move everything to the right device:

```
coords = coords.to(device)
radius = radius.to(device)
atoms_channel = atoms_channel.to(device)
```

Finally, we obtain the volumetric representations with:

```
SurfacePoitCloud = PointCloudSurfaceObject(coords, radius)
VolumePoitCloud = PointCloudVolumeObject(coords, radius)
VoxelRepresentation = VoxelsObject(coords, radius, atoms_channel)
```

CHAPTER

FIVE

PYUUL WITHOUT USING THE PARSERS

PyUUL allows the user to manually define the structure, bypassing the parsers. This might be useful in order to deal with exotic atoms that are not supported by the parsers, but also in order to use PyUUL as an intermediate step in an end-to-end neural network.

As usual, we first need to import PyUUL modules:

```
from pyuul import VolumeMaker # the main PyUUL module
from pyuul import utils # the PyUUL utility module
import torch
```

We then generate the main PyUUL python objects:

```
device = "cpu"
VoxelsObject = VolumeMaker.Voxels(device=device,sparse=True)
PointCloudVolumeObject = VolumeMaker.PointCloudVolume(device=device)
PointCloudSurfaceObject = VolumeMaker.PointCloudSurface(device=device)
```

Point clouds object only require the coordinates and radius tensors.

```
device = "cpu"
coordinates = torch.rand((5,10,3),device=device) #5 molecules of 10 atoms each
radius = torch.rand((5,10),device=device) #radius of each atom

SurfacePoitCloud = PointCloudSurfaceObject(coordinates, radius)
VolumePoitCloud = PointCloudVolumeObject(coordinates, radius)
```

For voxelized representation, we also need channels

```
channels = torch.randint(size=(5,10),low=0,high=5,device=device)
voxelRepresentation = VoxelsObject(coordinates, radius, channels)
```

Volumetric representations are differentiable, so autograd gradient of coordinates is not killed by this operation:

```
coordinates.requires_grad = True
channels = torch.randint(size=(5,10),low=0,high=5,device=device)
voxelRepresentation = VoxelsObject(coordinates, radius, channels).to_dense() #sparse␣
↪backward is not implemented in pytorch

toy_loss = voxelRepresentation.sum()
toy_loss.backward() # the gradient is propagated to coordinates
```

CHAPTER

SIX

TUTORIALS

6.1 Alpha Helices identification

This tutorial shows how to build a neural network that takes voxelized proteins as input. The tutorial replicates the example done in the paper.

CHAPTER

SEVEN

CONTACTS

For bug reports, features addition and technical questions please contact gabriele.orlando@kuleuven.be

For any other questions contact joost.Schymkowitz@kuleuven.be or frederic.rousseau@kuleuven.be

CHAPTER

EIGHT

PYUUL

8.1 VolumeMaker module

class VolumeMaker.**PointCloudSurface**(*device='cpu'*)

Bases: Module

__init__(*device='cpu'*)

Constructor for the CloudPointSurface class, which builds the main PyUUL object for cloud surface.

Parameters

device (*torch.device*) – The device on which the model should run. E.g. torch.device("cuda") or torch.device("cpu:0")

forward(*coords*, *radius*, *maxpoints=5000*, *external_radius_factor=1.4*)

Function to calculate the surface cloud point representation of macromolecules

Parameters

- **coords** (*torch.Tensor*) – Coordinates of the atoms. Shape (batch, numberOfAtoms, 3). Can be calculated from a PDB file using utils.parsePDB
- **radius** (*torch.Tensor*) – Radius of the atoms. Shape (batch, numberOfAtoms). Can be calculated from a PDB file using utils.parsePDB and utils.atomlistToRadius
- **maxpoints** (*int*) – number of points per macromolecule in the batch
- **external_radius_factor=1.4** – multiplicative factor of the radius in order ot define the place to sample the points around each atom. The higher this value is, the smoother the surface will be

Returns

surfacePointCloud – surface point cloud representation of the macromolecules in the batch. Shape (batch, channels, numberOfAtoms, 3)

Return type

torch.Tensor

training: **bool**

class VolumeMaker.**PointCloudVolume**(*device='cpu'*)

Bases: Module

__init__(*device='cpu'*)

Constructor for the CloudPointSurface class, which builds the main PyUUL object for volumetric point cloud.

Parameters
device (*torch.device*) – The device on which the model should run. E.g. torch.device("cuda") or torch.device("cpu:0")

forward(*coords*, *radius*, *maxpoints=5000*)

Function to calculate the volumetric cloud point representation of macromolecules

Parameters

- **coords** (*torch.Tensor*) – Coordinates of the atoms. Shape (batch, numberOfAtoms, 3). Can be calculated from a PDB file using utils.parsePDB
- **radius** (*torch.Tensor*) – Radius of the atoms. Shape (batch, numberOfAtoms). Can be calculated from a PDB file using utils.parsePDB and utils.atomlistToRadius
- **maxpoints** (*int*) – number of points per macromolecule in the batch

Returns
PointCloudVolume – volume point cloud representation of the macromolecules in the batch. Shape (batch, channels, numberOfAtoms, 3)

Return type
torch.Tensor

training: **bool**

class VolumeMaker.**Voxels**(*device=device(type='cpu')*, *sparse=True*)

Bases: Module

__init__(*device=device(type='cpu')*, *sparse=True*)

Constructor for the Voxels class, which builds the main PyUUL object.

Parameters

- **device** (*torch.device*) – The device on which the model should run. E.g. torch.device("cuda") or torch.device("cpu:0")
- **sparse** (*bool*) – Use sparse tensors calculation when possible

forward(*coords*, *radius*, *channels*, *numberchannels=None*, *resolution=1*, *cubes_around_atoms_dim=5*, *steepness=10*, *function='sigmoid'*)

Voxels representation of the macromolecules

Parameters

- **coords** (*torch.Tensor*) – Coordinates of the atoms. Shape (batch, numberOfAtoms, 3). Can be calculated from a PDB file using utils.parsePDB
- **radius** (*torch.Tensor*) – Radius of the atoms. Shape (batch, numberOfAtoms). Can be calculated from a PDB file using utils.parsePDB and utils.atomlistToRadius
- **channels** (*torch.LongTensor*) – channels of the atoms. Atoms of the same type shold belong to the same channel. Shape (batch, numberOfAtoms). Can be calculated from a PDB file using utils.parsePDB and utils.atomlistToChannels
- **numberchannels** (*int or None*) – maximum number of channels. if None, max(atNameHashing) + 1 is used
- **cubes_around_atoms_dim** (*int*) – maximum distance in number of voxels for which the contribution to occupancy is taken into consideration. Every atom that is farer than cubes_around_atoms_dim voxels from the center of a voxel does no give any contribution to the relative voxel occupancy

- **resolution** (*float*) – side in A of a voxel. The lower this value is the higher the resolution of the final representation will be
- **steepness** (*float or int*) – steepness of the sigmoid occupancy function.
- **function** (*"sigmoid" or "gaussian"*) – occupancy function to use. Can be sigmoid (every atom has a sigmoid shaped occupancy function) or gaussian (based on Li et al. 2014)

Returns

volume – voxel representation of the macromolecules in the batch. Shape (batch, channels, x,y,z), where x,y,z are the size of the 3D volume in which the macromolecules have been represented

Return type

torch.Tensor

training: **bool**

8.2 utils module

utils.**atomlistToChannels**(*atomNames*, *hashing='Element_Hashing'*, *device='cpu'*)

function to get channels from atom names (obtained parsing the pdb files with the parsePDB function)

Parameters

- **atomNames** (*list*) – atom names obtained parsing the pdb files with the parsePDB function
- **hashing** (*"TPL_Hashing" or "Element_Hashing" or dict*) – define which atoms are grouped together. You can use two default hashings or build your own hashing:

  TPL_Hashing: uses the hashing of torch protein library (https://github.com/lupoglaz/TorchProteinLibrary) Element_Hashing: groups atoms in accordnce with the element only: C -> 0, N -> 1, O ->2, P ->3, S- >4, H ->5, everything else ->6

  Alternatively, if you are not happy with the default hashings, you can build a dictionary of dictionaries that defines the channel of every atom type in the pdb. the first dictionary has the residue tag (three letters amino acid code) as key (3 letters compound name for hetero atoms, as written in the PDB file) every residue key is associated to a dictionary, which the atom tags (as written in the PDB files) as keys and the channel (int) as value

  for example, you can define the channels just based on the atom element as following: { 'CYS': {'N': 1, 'O': 2, 'C': 0, 'SG': 3, 'CB': 0, 'CA': 0}, # channels for cysteine atoms 'GLY': {'N': 1, 'O': 2, 'C': 0, 'CA': 0}, # channels for glycine atom ... 'GOL': {'O1':2,'O2':2,'O3':2,'C1':0,'C2':0,'C3':0}, # channels for glycerol atom ... }

  The default encoding is the one that assigns a different channel to each element

  other encodings can be found in sources/hashings.py
- **device** (*torch.device*) – The device on which the model should run. E.g. torch.device("cuda") or torch.device("cpu:0")

Returns

- **coords** (*torch.Tensor*) – coordinates of the atoms in the pdb file(s). Shape (batch, numberOfAtoms, 3)
- **channels** (*torch.tensor*) – the channel of every atom. Shape (batch,numberOfAtoms)

utils.**atomlistToRadius**(*atomList*, *hashing='FoldX_radius'*, *device='cpu'*)

function to get radius from atom names (obtained parsing the pdb files with the parsePDB function)

Parameters

- **atomNames** (*list*) – atom names obtained parsing the pdb files with the parsePDB function
- **hashing** (*FoldX_radius or dict*) – "FoldX_radius" provides the radius used by the FoldX force field

  Alternatively, if you are not happy with the foldX radius, you can build a dictionary of dictionaries that defines the radius of every atom type in the pdb. The first dictionary has the residue tag (three letters amino acid code) as key (3 letters compound name for hetero atoms, as written in the PDB file) every residue key is associated to a dictionary, which the atom tags (as written in the PDB files) as keys and the radius (float) as value

  for example, you can define the radius as following: { 'CYS': {'N': 1.45, 'O': 1.37, 'C': 1.7, 'SG': 1.7, 'CB': 1.7, 'CA': 1.7}, # radius for cysteine atoms 'GLY': {'N': 1.45, 'O': 1.37, 'C': 1.7, 'CA': 1.7}, # radius for glycine atoms ... 'GOL': {'O1':1.37,'O2':1.37,'O3':1.37,'C1':1.7,'C2':1.7,'C3':1.7}, # radius for glycerol atoms ... }

  The default radius are the ones defined in FoldX

  Radius default dictionary can be found in sources/hashings.py
- **device** (*torch.device*) – The device on which the model should run. E.g. torch.device("cuda") or torch.device("cpu:0")

Returns

- **coords** (*torch.Tensor*) – coordinates of the atoms in the pdb file(s). Shape (batch, numberOfAtoms, 3)
- **radius** (*torch.tensor*) – The radius of every atom. Shape (batch,numberOfAtoms)

utils.**parsePDB**(*PDBFile*, *keep_only_chains=None*, *keep_hetatm=True*, *bb_only=False*)

function to parse pdb files. It can be used to parse a single file or all the pdb files in a folder. In case a folder is given, the coordinates are gonna be padded

Parameters

- **PDBFile** (*str*) – path of the PDB file or of the folder containing multiple PDB files
- **bb_only** (*bool*) – if True ignores all the atoms but backbone N, C and CA
- **keep_only_chains** (*str or None*) – ignores all the chain but the one given. If None it keeps all chains
- **keep_hetatm** (*bool*) – if False it ignores heteroatoms

Returns

- **coords** (*torch.Tensor*) – coordinates of the atoms in the pdb file(s). Shape (batch, numberOfAtoms, 3)
- **atomNames** (*list*) – a list of the atom identifier. It encodes atom type, residue type, residue position and chain

utils.**parseSDF**(*SDFFile*)

function to parse pdb files. It can be used to parse a single file or all the pdb files in a folder. In case a folder is given, the coordinates are gonna be padded

Parameters

SDFFile (*str*) – path of the PDB file or of the folder containing multiple PDB files

Returns

- **coords** (*torch.Tensor*) – coordinates of the atoms in the pdb file(s). Shape (batch, numberOfAtoms, 3)
- **atomNames** (*list*) – a list of the atom identifier. It encodes atom type, residue type, residue position and chain

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

F

M

P

T

U

V