HOOMD-TF

Release 1.0.2

Aug 15, 2020

Table of Contents

This plugin enables the use of TensorFlow in a HOOMD-blue simulation

HOOMD-TF can be used for a variety of tasks such as online force-matching, online machine learning in HOOMD-blue simulations, and arbitrary collective variable calculations using TensorFlow tensor operations. Because both HOOMD-blue and TensorFlow are GPU-accelerated, HOOMD-TF was designed with speed in mind, and minimizes latency with a GPU-GPU communication scheme. Of particular interest, HOOMD-TF allows for online machine learning with early termination, rather than the more tradditional batch learning approach for MD+ML.

HOOMD-TF includes several utility functions as convenient built-ins, such as:

- RDF calculation
- EDS Biasing (See this paper)
- Coarse-Grained simulation force matching

In addition to all these, the TensorFlow interface of HOOMD-TF makes implementing arbitrary ML models as easy as it is in TensorFlow, by exposing the HOOMD-blue neighbor list and particle positions to TensorFlow. This enables GPU-accelerated tensor calculations, meaning arbitrary collective variables can be treated in the TensorFlow model framework, as long as they can be expressed as tensor operations on particle positions or neighbor lists.

CHAPTER 1

Citation

Please use the following citation:

> HOOMD-TF: GPU-Accelerated, Online Machine Learning in the Hoomd-blue Molecular Dynamics Engine. R Barrett, M Chakraborty, DB Amirkulova, HA Gandhi, G Wellawatte, and AD White (2020) *Journal of Open Source Software* doi: 10.21105/joss.02367

CHAPTER 2

Tutorial

See example notebooks here to learn about what Hoomd-TF can do.

2.1 Quickstart Tutorial

Here's an example of how you use Hoomd-TF. To compute a `1 / r` pairwise potential:

```
import hoomd, hoomd.md
import hoomd.htf as htf
import tensorflow as tf

########### Graph Building Code ###########
graph = htf.graph_builder(64) # max neighbors = 64
pair_energy = graph.nlist_rinv # nlist_rinv is neighbor 1 / r
particle_energy = tf.reduce_sum(pair_energy, axis=1) # sum over neighbors
forces = graph.compute_forces(particle_energy) # compute forces
graph.save('my_model', forces)

########### Hoomd-Sim Code ###############
hoomd.context.initialize()
# this will start TensorFlow, so it goes
# in a with statement for clean exit
with htf.tfcompute('my_model') as tfcompute:
    # create a square lattice
    system = hoomd.init.create_lattice(unitcell=hoomd.lattice.sq(a=4.0),
                                       n=[3,3])
    nlist = hoomd.md.nlist.cell()
    hoomd.md.integrate.mode_standard(dt=0.005)
    hoomd.md.integrate.nve(group=hoomd.group.all())
    tfcompute.attach(nlist, r_cut=3.0)
    hoomd.run(1e3)
```

This creates a computation graph whose energy function is `2 / r` and also computes forces and virial for the simulation. The `2` is because the neighborlists in Hoomd-TF are *full* neighborlists (double counted). The Hoomd-blue

code starts a simulation of a 9 particle square lattice and simulates it for 1000 timesteps under the potential defined in our Hoomd-TF model. The general process of using Hoomd-TF is to build a TensorFlow computation graph, load the graph, and then attach the graph. See *Building the Graph* and *Using a Graph in a Simulation* for a more detailed description. Or see a complete set of Jupyter Notebook tutorials.

2.2 Contributor Code of Conduct

2.2.1 Introduction

All contributors to this project are expected to follow the code of conduct detailed in this document at all times when engaging with or discussing the HOOMD-TF project. This includes public/private mailing lists, issue tracker threads, blogs, personal websites, social media, and any other forms of communication, in-person or otherwise.

This code of conduct is expected to be followed by any and all participants, whether or not they are affiliated with the White Laboratory, whether or not they claim affiliation with HOOMD-TF, including when representing HOOMD-TF in any capacity.

This code of conduct is not exhaustive and may be subject to changes and improvements in the future. It is meant to foster an environment of collaboration and respect. In order to maintain a respectful community, please follow the *intent* of this page as much as the literal writing.

2.2.2 Specific Guidelines

We, the contributors and participants, aim to:

1. Be welcoming. Any and all who are willing and able to contribute to this project should feel welcome to participate. To this end, we encourage the use of public communication such as GitHub issue tracking as much as possible, to ensure those who join later can benefit from any useful discussion.

2. Be nice. We strive to avoid assigning ill intent and assume the best of others first. We should be patient with newcomers and avoid talking down to anyone. We should never allow personal negative feelings to turn into an attack on someone else's character or work. A negative environment is counter-productive and just not pleasant. In particular, violence, threats of violence, unwelcome personal advances, insults, and sexist or racist comments or jokes are entirely unwelcome, as is encouragement of any of these.

3. Be humble. Criticisms of code are not personal and should be taken on good faith. Any improvement is good.

4. Be patient. Everyone has different levels of experience, and a willingness to learn should be met with a willingness to teach. Not everyone is obligated to answer every question in-depth, but no one should be ignored or barred from helping because of a lack of experience.

5. Be transparent. Since this is a community project, new users and developers should be able to understand how and why it works the way it does. Contributors should be willing to answer questions about their work, and should follow the documentation style to ensure a smooth learning curve.

6. Be inclusive. Discrimination based on age, disability, ethnicity, familial status, gender identity, nationality, religion, sex, sexual orientation, or any legally-protected group will not be tolerated.

2.2.3 Reporting Issues

Issues can be reported via GitHub. When opening an issue, please be as specific as possible, and provide a way to replicate your issue so it can be addressed readily.

2.2.4 Contributing

Contributions are welcome and can be submitted by opening a pull request on our GitHub. As with reporting an issue, pull requests for contributions should be as specific as possible for smooth integration into the codebase. Please ensure that all tests pass before opening a pull request. If your PR includes new functionality, please also include unit testing for it, following the testing and documentation framework (see here). A contribution PR should include:

1. What the contribution is meant to address.
2. How to run any new unit tests (if applicable).

2.3 Installation Guide

2.3.1 Compiling

Prerequisites

The following packages are required to compile:

```
tensorflow < 2.0
hoomd-blue == 2.5.2 (for GPU)
hoomd-blue >= 2.5.2 (for CPU)
numpy
tbb-devel (only for hoomd-blue 2.8 and above)
```

tbb-devel is required for hoomd-blue 2.8 or above when using the hoomd-blue conda release. It is not automatically installed when installing hoomd-blue, so use `conda install -c conda-forge tbb-devel` to install. The Tensorflow version should be any Tensorflow 1 release. The higher versions, like 1.14, 1.15, will give lots of warnings about migrating code to Tensorflow 2.0. It is recommended you install via pip:

```
pip install tensorflow-gpu==1.15.0
```

Python and GCC requirements

If you install tensorflow with pip, as recommended, this provides a pre-built version of tensorflow which has specific GCC and Python versions. When you compile hoomd-tf, these must match what is found by cmake. So if your version of tensorflow used gcc-7x, then you must have gcc-7x available on your machine. The cmake script in hoomd-tf will check for this and tell you if they do not match.

Simple Compiling

Install hoomd-blue and Tensorflow by your preferred method. If you want to install hoomd-blue without GPU support, you can just use the conda release via `conda install -c conda-forge hoomd==2.5.2`. You should then similarly use the CPU version of Tensorflow (`pip install tensorflow==1.15`). If you would like GPU support, compile hoomd-blue using their instructions. Remember that pip is recommended for installing Tensorflow. Here are steps **after** installing hoomd-blue and tensorflow

```
git clone https://github.com/ur-whitelab/hoomd-tf
cd hoomd-tf && mkdir build && cd build
CXX=g++ CC=gcc cmake ..
make install
```

That's it! Check your install by running `python htf/test-py/test_sanity.py`. If you have installed with GPU support, also check with `python htf/test-py/_test_gpu_sanity.py`.

Compiling with Hoomd-Blue

Use this method if you need to compile with developer flags on or other special requirements.

```
git clone --recursive https://bitbucket.org/glotzer/hoomd-blue hoomd-blue
```

We typically use v2.5.2 of hoomd-blue

```
cd hoomd-blue && git checkout tags/v2.5.2
```

Now we put our plugin in the source directory with a softlink:

```
git clone https://github.com/ur-whitelab/hoomd-tf
ln -s $HOME/hoomd-tf/htf $HOME/hoomd-blue/hoomd
```

Now compile (from hoomd-blue directory). Modify options for speed if necessary. Set build type to *DEBUG* if you need to troubleshoot.

```
mkdir build && cd build
CXX=g++ CC=gcc cmake .. -DCMAKE_BUILD_TYPE=Release \
 -DENABLE_CUDA=ON -DENABLE_MPI=OFF\
 -DBUILD_HPMC=off -DBUILD_CGCMM=off -DBUILD_MD=on\
 -DBUILD_METAL=off -DBUILD_TESTING=off -DBUILD_DEPRECATED=off -DBUILD_MPCD=OFF \
 -DCMAKE_INSTALL_PREFIX=`python -c "import site; print(site.getsitepackages()[0])"`
```

Now compile with make:

```
make
```

Option 1: Put build directory on your python path:

```
export PYTHONPATH="$PYTHONPATH:`pwd`"
```

Option 2: Install in your python site-packages

```
make install
```

Conda Environments

If you are using a conda environment, you may need to force cmake to find your python environment. This is rare, we only see it on our compute cluster which has multiple conflicting version of python and conda. The following additional flags can help with this:

```
export CMAKE_PREFIX_PATH=/path/to/environment
CXX=g++ CC=gcc cmake .. \
-DPYTHON_INCLUDE_DIR=$(python -c "from distutils.sysconfig import get_python_inc;␣
↪print(get_python_inc())") \
-DPYTHON_LIBRARY=$(python -c "import distutils.sysconfig as sysconfig;␣
↪print(sysconfig.get_config_var('LIBDIR'))") \
-DPYTHON_EXECUTABLE=$(which python) \
-DCMAKE_BUILD_TYPE=Release -DENABLE_CUDA=ON -DENABLE_MPI=OFF -DBUILD_HPMC=off -DBUILD_
↪CGCMM=off -DBUILD_MD=on \
```

(continues on next page)

(continued from previous page)

```
-DBUILD_METAL=off -DBUILD_TESTING=off -DBUILD_DEPRECATED=off -DBUILD_MPCD=OFF \
-DCMAKE_INSTALL_PREFIX=`python -c "import site; print(site.getsitepackages()[0])"`
```

Updating Compiled Code

If you are developing frequently, add the build directory to your python path instead of *make install* (only works with hoomd-blue compiled). Then if you modify C++ code, only run make (not cmake). If you modify python, just copy over py files (`htf/*py` to `build/hoomd/htf`).

MBuild Environment

If you are using mbuild, please follow these additional install steps:

```
conda install numpy cython
pip install requests networkx matplotlib scipy pandas plyplus lxml mdtraj oset
conda install -c omnia -y openmm parmed
conda install -c conda-forge --no-deps -y packmol gsd
pip install --upgrade git+https://github.com/mosdef-hub/foyer git+https://github.com/
↪mosdef-hub/mbuild
```

2.3.2 HPC Installation

These are instructions for our group's cluster (BlueHive), and not for general users. **Feeling Lucky?** Try this for quick results

```
module load cmake gcc/7.3.0 cudnn/10.0-7.5.0 anaconda3/2019.10
export PYTHONNOUSERSITE=True
conda create -n hoomd-tf python=3.7
source activate hoomd-tf
export CMAKE_PREFIX_PATH=/path/to/environment
python -m pip install tensorflow-gpu==1.15.0
conda install -c conda-forge hoomd==2.5.2
git clone https://github.com/ur-whitelab/hoomd-tf
cd hoomd-tf && mkdir build && cd build
CXX=g++ CC=gcc cmake .. \
  -DPYTHON_INCLUDE_DIR=$(python -c "from distutils.sysconfig import get_python_inc;␣
↪print(get_python_inc())") \
  -DPYTHON_LIBRARY=$(python -c "import distutils.sysconfig as sysconfig;␣
↪print(sysconfig.get_config_var('LIBDIR'))") \
  -DPYTHON_EXECUTABLE=$(which python)
make install
cd .. && python htf/test-py/test_sanity.py
```

Here are the more detailed steps. Clone the `hoomd-tf` repo and then follow these steps:

Load the modules necessary:

```
module load cmake gcc/7.3.0 cudnn/10.0-7.5.0 anaconda3/2019.10
```

Set-up virtual python environment *ONCE* to keep packages isolated.

```
conda create -n hoomd-tf python=3.7
source activate hoomd-tf
python -m pip install tensorflow-gpu==1.15.0
```

Then whenever you login and *have loaded modules*:

```
source activate hoomd-tf
```

Continue following the compling steps below to complete install. The simple approach is recommended but **use the following different cmake step**

```
export CMAKE_PREFIX_PATH=/path/to/environment
CXX=g++ CC=gcc cmake ..
```

If using the hoomd-blue compilation, **use the following different cmake step**

```
export CMAKE_PREFIX_PATH=/path/to/environment
CXX=g++ CC=gcc cmake .. \
-DPYTHON_INCLUDE_DIR=$(python -c "from distutils.sysconfig import get_python_inc;␣
↪print(get_python_inc())") \
-DPYTHON_LIBRARY=$(python -c "import distutils.sysconfig as sysconfig;␣
↪print(sysconfig.get_config_var('LIBDIR'))") \
-DPYTHON_EXECUTABLE=$(which python) \
-DCMAKE_BUILD_TYPE=Release -DENABLE_CUDA=ON -DENABLE_MPI=OFF -DBUILD_HPMC=off -DBUILD_
↪CGCMM=off -DBUILD_MD=on \
-DBUILD_METAL=off -DBUILD_TESTING=off -DBUILD_DEPRECATED=off -DBUILD_MPCD=OFF \
-DCMAKE_INSTALL_PREFIX=`python -c "import site; print(site.getsitepackages()[0])"`\
-DNVCC_FLAGS="-ccbin /software/gcc/7.3.0/bin"
```

2.3.3 Optional Dependencies

Following packages are optional: .. code:: bash

MDAnalysis

utils.run_from_trajectory uses *MDAnalysis* for trajectory parsing

2.4 Unit Tests

You can run the unit tests directly via `python htf/test-py/test_tensorflow.py`, `python htf/test-py/test_utils.py`, etc. For convenience, you can also install the `pyest` package and run the following from the root HOOMD-TF directory:

```
pytest htf/test-py/
```

2.5 Benchmarking

The `benchmark.py` script in the `htf/benchmarking` directory is a convenience script for checking benchmark results. Example use syntax is shown below. This will run a benchmark trial with 10000 Lennard-Jones particles with HOOMD-blue in GPU configuration, and save the results as a .txt file in the current working directory. The first argument should be an integer for the number of particles to simulate. The second should be either "gpu" or "cpu" to indicate the execution mode, and the third indicates where to save the results. Note that large systems may take

some time, as the HOOMD benchmarking utility runs five repeats of a 50,000 timestep simulation with 6,000 steps of warmup each time. In order to run GPU profiling, make sure you have compiled for GPU (see *Compiling*).

```
python htf/benchmarking/benchmark.py 10000 gpu $(pwd)
```

2.6 Building the Graph

To construct a graph, create a *graphbuilder.graph_builder* instance:

```
import hoomd.htf as htf
graph = htf.graph_builder(NN, output_forces)
```

where `NN` is the maximum number of nearest neighbors to consider (can be 0). This is an upper-bound, so choose a large number. If you are unsure, you can guess and add `check_nlist = True`. This will cause the program to halt if you choose too low. `output_forces` indicates if the graph will output forces to use in the simulation. After building the `graph`, it will have five tensors as attributes that can be used when constructing the TensorFlow graph: `nlist`, `positions`, `box`, `box_size`, and `forces`:

- `nlist` is an `N` x `NN` x 4 tensor containing the nearest neighbors. An entry of all zeros indicates that less than `NN` nearest neighbors where present for a particular particle. The 4 right-most dimensions are `x,y,z` and `w`, which is the particle type. Particle type is an integer starting at 0. Note that the `x,y,z` values are a vector originating at the particle and ending at its neighbor.
- `positions` is an `N` x 4 tensor of particle positions (x,y,z) and type.
- `forces` is an `N` x 4 tensor that is *only* available if the graph does not output forces (via `output_forces=False`).
- `box` is a 3x3 tensor containing the low box coordinate, high box coordinate, and then tilt factors. `box_size` contains just the box length in each dimension.

2.6.1 Molecule Batching

It may be simpler to have positions or neighbor lists or forces arranged by molecule. For example, you may want to look at only a particular bond or subset of atoms in a molecule. To do this, you can call *graphbuilder.graph_builder.build_mol_rep()*, whose argument `MN` is the maximum number of atoms in a molecule. This will create the following new attributes: `mol_positions`, `mol_nlist`, and `mol_forces` (if your graph has `output_forces=False`). These new attributes are dimension `M x MN x ...` where `M` is the number of molecules and `MN` is the atom index within the molecule. If your molecule has fewer than `MN` atoms, extra entries will be zeros. You can defnie a molecule to be whatever you want, and atoms need not be only in one molecule. Here's an example to compute a water angle, where we assume that the oxygens are the middle atom:

```
import hoomd.htf as htf
graph = graph_builder(0)
graph.build_mol_rep(3)
# want slice for all molecules (:)
# want h1 (0), o (1), h2(2)
# positions are x,y,z,w. We only want x,y z (:3)
v1 = graph.mol_positions[:, 2, :3] - graph.mol_positions[:, 1, :3]
v2 = graph.mol_positions[:, 0, :3] - graph.mol_positions[:, 1, :3]
# compute per-molecule dot product and divide by per molecule norm
c = tf.einsum('ij,ij->i', v1, v2) / tf.norm(v1, axis=1) / tf.norm(v2 axis=1)
angles = tf.math.acos(c)
```

2.6.2 Computing Forces

If your graph is outputting forces, you may either compute forces and pass them to *graphbuilder.graph_builder.save()* or have them computed via automatic differentiation of a potential energy. Call *graphbuilder.graph_builder.compute_forces()* with the argument `energy`, which can be either a scalar or a tensor which depends on `nlist` and/or `positions`. A tensor of forces will be returned as $\sum_i(\frac{-\partial E}{\partial n_i}) - \frac{dE}{dp}$, where the sum is over the neighbor list. For example, to compute a $1/r$ potential:

```
graph = htf.graph_builder(N - 1)
#remove w since we don't care about types
nlist = graph.nlist[:, :, :3]
#get r
r = tf.norm(nlist, axis=2)
#compute 1. / r while safely treating r = 0.
# halve due to full nlist
rij_energy = 0.5 * graph.safe_div(1, r)
#sum over neighbors
energy = tf.reduce_sum(rij_energy, axis=1)
forces = graph.compute_forces(energy)
```

Notice that in the above example that we have used the *graphbuilder.graph_builder.safe_div()* method, which allows us to safely treat a $1/0$, which can arise because `nlist` contains 0s for when fewer than `NN` nearest neighbors are found.

Note: because `nlist` is a *full* neighbor list, you should divide by 2 if your energy is a sum of pairwise energies.

2.6.3 Neighbor lists

As mentioned above, *graphbuilder.graph_builder* contains a member called `nlist`, which is an `N x NN x 4` neighobr list tensor. You can ask for masked versions of this with *graphbuilder.graph_builder.masked_nlist()* where `type_i` and `type_j` are optional integers that specify the type of the origin (`type_i`) or neighobr (`type_j`). The `nlist` argument allows you to pass in your own neighbor list and `type_tensor` allows you to specify your own list of types, if different than what is given by hoomd-blue. You can also access `nlist_rinv` which gives a pre-computed `1 / r` (dimension `N x NN`).

2.6.4 Virial

The virial is computed and added to the graph if you use the *graphbuilder.graph_builder.compute_forces()* method and your energy has a non-zero derivative with respect to `nlist`. You may also explicitly pass the virial when saving, or pass `None` to remove the automatically-calculated virial.

2.6.5 Finalizing the Graph

To finalize and save your graph, you must call *graphbuilder.graph_builder.save()* with the following arguments:

- `directory`: where to save your TensorFlow model files
- `force_tensor` (optional): your computed forces, either as computed by your graph or output from *graphbuilder.graph_builder.compute_forces()*. This should be an `N x 4` tensor with the 4th column indicating per-particle potential energy.
- `virial` (optional): the virial tensor to save. The virial should be an `N x 3 x 3` tensor.

- `out_nodes` (optional): If your graph is not outputting forces, then you must provide a tensor or list of tensors which will be computed at each timestep.

2.6.6 Saving Data

Using variables is the best way to save computed quantities while running a compute graph. See the *Loading Variables* section for loading them. You can save a tensor value to a variable using *graphbuilder.graph_builder.save_tensor()*. Here is an example of computing the LJ potential and saving the system energy at each step.

```
# set-up graph
graph = htf.graph_builder(NN)
# compute LJ potential
inv_r6 = graph.nlist_rinv**6
p_energy = 4.0 / 2.0 * (inv_r6 * inv_r6 - inv_r6)
energy = tf.reduce_sum(p_energy)
# save the tensor
graph.save_tensor(energy, 'lj-energy')
forces = graph.compute_forces(energy)
# save the graph
graph.save(force_tensor=forces, model_directory=directory)
```

Often you may want a running mean of a variable, for which there is a built-in, *graphbuilder.graph_builder.running_mean()*:

```
# set-up graph to compute energy
...
# we name our variable avg-energy
graph.running_mean(energy, 'avg-energy')
# run the simulation
...
```

2.6.7 Variables and Restarts

In TensorFlow, variables are generally trainable parameters. They are required parts of your graph when doing learning. Each `save_period` (set as arg to *tensorflowcompute.tfcompute.attach()*), they are written to your model directory. Note that when a run is started, the latest values of your variables are loaded from your model directory. *If you are starting a new run but you previously ran your model, the old variable values will be loaded.* To prevent this unexpectedly loading old checkpoints, if you run *graphbuilder.graph_builder.save()*, it will move out all old checkpoints. This behavior means that if you want to restart, you should not re-run *graphbuilder.graph_builder.save()* in your restart script, *nor* should you pass `move_previous = False` as a parameter if you re-run *graphbuilder.graph_builder.save()*.

Variables are also how you save data as seen above. If you are doing training and also computing other variables, be sure to set your variables which you do not want to be affected by training optimization to be `trainable=False` when constructing them.

2.6.8 Loading Variables

You may load variables after the simulation using the following syntax:

```
variables  = htf.load_variables(model_dir, ['avg-energy'])
```

The *utils.load_variables()* is general and can be used to load trained, non-trained, or averaged variables. **It is important to name your custom variables so they can be loaded using this function.**

2.6.9 Period of out nodes

You can modify how often tensorflow is called via *tensorflowcompute.tfcompute.attach()*. You can also have more granular control of operations/tensors passed to `out_nodes` by changing the type to a list whose first element is the tensor and the second argument is the period at which it is computed. For example:

```
...graph building code...
forces = graph.compute_forces(energy)
avg_force = tf.reduce_mean(forces, axis=-1)
print_node = tf.Print(energy, [energy], summarize=1000)
graph.save(force_tensor=forces, model_directory=name, out_nodes=[[print_node, 100],␣
↪[avg_force, 25]])
```

This will print the energy every 100 steps and compute the average force every 25 steps (although it is unused). Note that these two ways of affecting period both apply. So if the above graph was attached with `tfcompute.attach(..., period=25)` then the `print_node` will be run only every 2500 steps.

2.6.10 Printing

If you would like to print out the values from nodes in your graph, you can add a print node to the `out_nodes`. For example:

```
...graph building code...
forces = graph.compute_forces(energy)
print_node = tf.Print(energy, [energy], summarize=1000)
graph.save(force_tensor=forces, model_directory=name, out_nodes=[print_node])
```

The `summarize` keyword sets the maximum number of numbers to print. Be wary of printing thousands of numbers per step.

2.6.11 Optional: Keras Layers for Model Building

Currently HOOMD-TF supports Keras layers in model building. We do not yet support Keras `Model.compile()` or `Model.fit()`. This example shows how to set up a neural network model using Keras layers.

```
import tensorflow as tf
from tensorflow.keras import layers
import hoomd.htf as htf

NN = 64
N_hidden_nodes = 5
graph = htf.graph_builder(NN, output_forces=False)
r_inv = graph.nlist_rinv
input_tensor = tf.reshape(r_inv, shape=(-1,1), name='r_inv')
#we don't need to explicitly make a keras.Model object, just layers
input_layer = layers.Input(tensor=input_tensor)
hidden_layer = layers.Dense(N_hidden_nodes)(input_layer)
output_layer = layers.Dense(1, input_shape=(N_hidden_nodes,))(hidden_layer)
#do not call Model.compile, just use the output in the TensorFlow graph
nn_energies = tf.reshape(output_layer, [-1, NN])
calculated_energies = tf.reduce_sum(nn_energies, axis=1, name='calculated_energies')
calculated_forces = graph.compute_forces(calculated_energies)
#cost and optimizer must also be set through TensorFlow, not Keras
cost = tf.losses.mean_squared_error(calculated_forces, graph.forces)
```

(continues on next page)

(continued from previous page)

```
optimizer = tf.train.AdamOptimizer(0.001).minimize(cost)
#save using graph.save, not Keras Model.compile
graph.save(model_directory='/tmp/keras_model/', out_nodes=[ optimizer])
```

The model can then be loaded and trained as normal. Note that `keras.models.Model.fit()` is not currently supported. You must train using *`tensorflowcompute.tfcompute`* as explained in the next section.

2.6.12 Complete Examples

The directory *htf/models* contains some example scripts.

2.6.13 Lennard-Jones with 1 Particle Type

```
graph = hoomd.htf.graph_builder(NN)
#use convenience rinv
r_inv = graph.nlist_rinv
p_energy = 4.0 / 2.0 * (r_inv**12 - r_inv**6)
#sum over pairwise energy
energy = tf.reduce_sum(p_energy, axis=1)
forces = graph.compute_forces(energy)
graph.save(force_tensor=forces, model_directory='/tmp/lj-model')
```

2.7 Using a Graph in a Simulation

You may use a saved TensorFlow model via:

```
import hoomd, hoomd.md
import hoomd.htf as htf

...hoomd initialization code...
with htf.tfcompute(model_dir) as tfcompute:

    nlist = hoomd.md.nlist.cell()
    tfcompute.attach(nlist, r_cut=3)

    ...other hoomd code...

    hoomd.run(...)
```

where `model_dir` is the directory where the TensorFlow model was saved, `nlist` is a hoomd neighbor list object and `r_cut` is the maximum distance for to consider particles as being neighbors. `nlist` is optional and is not required if your graph doesn't use the `nlist` object (you passed `NN = 0` when building your graph).

2.7.1 Logging

The default logging level of Tensorflow is relatively noisy. You can reduce the amount of logged statements via

2.7.2 Batching

If you used per-molecule positions or nlist in your graph, you can either rely on hoomd-tf to find your molecules by traversing the bonds in your system (default) or you can specify what are molecules in your system. They are passed via `attach(..., mol_indices=[[..]])`. The `mol_indices` are a, possibly ragged, 2D python list where each element in the list is a list of atom indices for a molecule. For example, `[[0,1], [1]]` means that there are two molecules with the first containing atoms 0 and 1 and the second containing atom 1. Note that the molecules can be different size and atoms can exist in multiple molecules.

If you do not call *graphbuilder.graph_builder.build_mol_rep()* while building your graph, you can optionally split your batches to be smaller than the entire system. This is set via the `batch_size` integer argument to `tfcompute.tfcompute.attach()`. This can help for high-memory simulations where you cannot spare the GPU memory to have each tensor be the size of your system.

2.7.3 Bootstrapping Variables

If you have trained variables previously and would like to load them into the current TensorFlow graph, you can use the `bootstrap` and `bootstrap_map` arguments. `bootstrap` should be a checkpoint file path or model directory path (latest checkpoint is used) containing variables which can be loaded into your tfcompute graph. Your model will be built, then all variables will be initialized, and then your bootstrap checkpoint will be loaded and no variables will be reloaded even if there exists a checkpoint in the model directory (to prevent overwriting your bootstrap variables). `bootstrap_map` is an optional additional argument that will have keys that are variable names in the `bootstrap` checkpoint file and values that are names in the tfcompute graph. This can be used when your variable names do not match up. Here are two example demonstrating with and without a `bootstrap_map`:

Here's an example that creates some variables that could be trained offline without Hoomd. In this example, they just use their initial values.

```
import tensorflow as tf

#make some variables
v = tf.Variable(8.0, name='epsilon')
s = tf.Variable(2.0, name='sigma')

#initialize and save them
saver = tf.train.Saver()
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    saver.save(sess, '/tmp/bootstrap/model')
```

We load them in the hoomd run script:

```
with hoomd.htf.tfcompute(model_dir,
    bootstrap='/tmp/bootstrap/model') as tfcompute:
    ...
```

Here's how we would load them in the hoomd run script if we want to change the names of the variables:

```
# here the pretrained variable parameters will replace variables with a different name
with hoomd.htf.tfcompute(model_dir,
    bootstrap='/tmp/bootstrap/model',
    bootstrap_map={'lj-epsilon':'epsilon', 'lj-sigma':'sigma'}) as tfcompute:
    ...
```

2.7.4 Bootstrapping Variables from Other Models

Here's an example of bootstrapping where you train with Hoomd-TF and then load the variables into a different model:

```
# build_models.py
import tensorflow as tf
import hoomd.htf as htf

def make_train_graph(NN, directory):
    # build a model that fits the energy to a linear term
    graph = htf.graph_builder(NN, output_forces=False)
    # get r
    nlist = graph.nlist[:, :, :3]
    r = graph.safe_norm(nlist, axis=2)
    # build energy model
    m = tf.Variable(1.0, name='m')
    b = tf.Variable(0.0, name='b')
    predicted_particle_energy = tf.reduce_sum(m * r + b, axis=1)
    # get energy from hoomd
    particle_energy = graph.forces[:, 3]
    # make them match
    loss = tf.losses.mean_squared_error(particle_energy, predicted_particle_energy)
    optimize = tf.train.AdamOptimizer(1e-3).minimize(loss)
    graph.save(model_directory=directory, out_nodes=[optimize])

def make_force_graph(NN, directory):
    # this model applies the variables learned in the example above
    # to compute forces
    graph = htf.graph_builder(NN)
    # get r
    nlist = graph.nlist[:, :, :3]
    r = graph.safe_norm(nlist, axis=2)
    # build energy model
    m = tf.Variable(1.0, name='m')
    b = tf.Variable(0.0, name='b')
    predicted_particle_energy = tf.reduce_sum(m * r + b, axis=1)
    forces = graph.compute_forces(predicted_particle_energy)
    graph.save(force_tensor=forces, model_directory=directory)
make_train_graph(64, 16, '/tmp/training')
make_force_graph(64, 16, '/tmp/inference')
```

Here is how we run the training model:

```
#run_train.py
import hoomd, hoomd.md
import hoomd.htf as htf


hoomd.context.initialize()

with htf.tfcompute('/tmp/training') as tfcompute:
    rcut = 3.0
    system = hoomd.init.create_lattice(unitcell=hoomd.lattice.sq(a=2.0),
                                       n=[8,8])
    nlist = hoomd.md.nlist.cell(check_period = 1)
    lj = hoomd.md.pair.lj(rcut, nlist)
    lj.pair_coeff.set('A', 'A', epsilon=1.0, sigma=1.0)
    hoomd.md.integrate.mode_standard(dt=0.005)
```

(continues on next page)

(continued from previous page)

```
    hoomd.md.integrate.nve(
        group=hoomd.group.all()).randomize_velocities(kT=0.2, seed=42)

    tfcompute.attach(nlist, r_cut=rcut)
    hoomd.run(100)
```

Load the variables trained in the training run into the model which computes forces:

```
#run_inference.py
import hoomd, hoomd.md
import hoomd.htf as htf

hoomd.context.initialize()
with htf.tfcompute('/tmp/inference',
        bootstrap='/tmp/training') as tfcompute:
    rcut = 3.0
    system = hoomd.init.create_lattice(unitcell=hoomd.lattice.sq(a=2.0),
                                       n=[8,8])
    nlist = hoomd.md.nlist.cell(check_period = 1)
    #notice we no longer compute forces with hoomd
    hoomd.md.integrate.mode_standard(dt=0.005)
    hoomd.md.integrate.nve(
        group=hoomd.group.all()).randomize_velocities(kT=0.2, seed=42)

    tfcompute.attach(nlist, r_cut=rcut)
    hoomd.run(100)
```

2.8 Utilities

There are a few convenience functions for plotting potential energies of pairwise potentials and constructing CG mappings.

2.8.1 RDF

To compute an RDF, use *graphbuilder.graph_builder.compute_rdf()*:

```
# set-up graph to compute energy
...
rdf = graph.compute_rdf([1,10], 'rdf', nbins=200)
graph.running_mean(rdf, 'avg-rdf')
# run the simulation
...
variables  = htf.load_variables(model_dir, ['avg-rdf'])
print(variables)
```

2.8.2 Pairwise Potential and Forces

To compute a pairwise potential, use *utils.compute_pairwise_potential()*:

```
...
r = numpy.arange(1, 10, 1)
potential, forces = htf.compute_pairwise_potential('/path/to/model', r, potential_
↪tensor)
...
```

2.8.3 Biasing with EDS

To apply Experiment Directed Simulation biasing to a system, use *utils.eds_bias()*:

```
eds_alpha = htf.eds_bias(cv, set_point=3.0, period=100)
eds_energy = eds_alpha * cv
eds_forces = graph.compute_forces(eds_energy)
graph.save('eds-graph', eds_forces)
```

Here, `htf.eds_bias(cv, set_point, period, learning_rate, cv_scale, name)` computes the lagrange multiplier/eds coupling that are used to bias the simulation. It may be useful to also take the average of `eds_alpha` so that you can use it in a subsequent simulation:

```
avg_alpha = graph.running_mean(eds_alpha, name='avg-eds-alpha')
.....
# after simulation
vars = htf.load_variables('model/directory', ['avg-eds-alpha'])
print(vars['avg-eds-alpha'])
```

2.8.4 Trajectory Parsing

To process information from a trajectory, use *utils.run_from_trajectory()*:

2.8.5 Coarse-Graining

Find Molecules

To go from atom index to particle index, use the *utils.find_molecules()*:

```
# The method takes in a hoomd system as an argument.
...
molecule_mapping_index = hoomd.htf.find_molecules(system)
...
```

Sparse Mapping

The *utils.sparse_mapping()* method creates the necessary indices and values for defining a sparse tensor in tensorflow that is a mass-weighted $M \times N$ mapping operator where M is the number of coarse-grained particles and N is the number of atoms in the system. In the following example,`mapping_per_molecule` is a list of $k \times n$ matrices where k is the number of coarse-grained sites for each molecule and n is the number of atoms in the corresponding molecule. There should be one matrix per molecule. Since the example is for a 1 bead mapping per molecule the shape is $1 \times n$. The ordering of the atoms should follow the output from the find_molecules method. The variable `molecule_mapping_index` is the output from *utils.find_molecules()*.

```
#The example is shown for 1 coarse-grained site per molecule.
...
molecule_mapping_matrix = numpy.ones([1, len(molecule_mapping_index[0])], dtype=np.
↪int)
mapping_per_molecule = [molecule_mapping_matrix for _ in molecule_mapping_index]
cg_mapping = htf.sparse_mapping(mapping_per_molecule, \
                    molecule_mapping_index, system = system)
...
```

Center of Mass

utils.center_of_mass() maps the given positions according to the specified mapping operator to coarse-grain site positions, while considering periodic boundary conditions. The coarse grain site position is placed at the center of mass of its constituent atoms.

```
...
mapped_position = htf.center_of_mass(graph.positions[:,:3], cg_mapping, system)
#cg_mapping is the output from the sparse_matrix(...) method and indicates how each␣
↪molecule is mapped.
...
```

Compute Mapped Neighbor List

utils.compute_nlist() returns the neighbor list for a set of mapped coarse-grained particles. In the following example, `mapped_positions` is the mapped particle positions obeying the periodic boundary condition, as returned by *utils.center_of_mass()*, `rcut` is the cutoff radius and `NN` is the number of nearest neighbors to be considered for the coarse-grained system.

```
...
mapped_nlist= htf.compute_nlist(mapped_positions, rcut, NN, system)
...
```

2.8.6 Tensorboard

You can visualize your models with tensorboard. First, add `write_tensorboard=True` to the `htf.tfcompute.tfcompute` constructor. This will add a new directory called `tensorboard` to your model directory.

After running, you can launch tensorboard like so:

```
tensorboard --logdir=/path/to/model/tensorboard
```

and then visit `http://localhost:6006` to view the graph.

Saving Scalars in Tensorboard

If you would like to save a scalar over time, like total energy or training loss, you can use the Tensorboard functionality. Add scalars to the Tensorboard summary during the build step:

```
tf.summary.scalar('total-energy', tf.reduce_sum(particle_energy))
```

and then add the `write_tensorboard=True` flag during the `htf.tfcompute.tfcompute` initialization. The period of tensorboard writes is controlled by the `save_period` flag to the `htf.tfcompute.tfcompute.attach()` command. See the Tensorboard section below for how to view the resulting scalars.

Viewing when TF is running on remote server

If you are running on a server, before launching tensorboard use this ssh command to login:

```
ssh -L 6006:[remote ip or hostname]:6006 username@remote
```

and then you can view after launching on the server via your local web browser.

Viewing when TF is running in container

If you are running docker, you can make this port available a few different ways. The first is to get the IP address of your docker container (google how to do this if not default), which is typically `172.0.0.1`, and then visit `http://172.0.0.1:6006` or equivalent if you have a different IP address for your container.

The second option is to use port forwarding. You can add a port forward flag, `-p 6006:6006`, when running the container which will forward traffic from your container's 6006 port to the host's 6006 port. Again, then you can visit `http://localhost:6006` (linux) or `http://127.0.0.1:6006` (windows).

The last method, which usually works when all others fail, is to have all the container's traffic be on the host. You can do this by adding the flag `--net=host` to the run command of the container. Then you can visit `http://localhost:6006`.

2.8.7 Interactive Mode

Experimental, but you can trace your graph in realtime in a simulation. Add both the `write_tensorboard=True` to the constructor and the `_debug_mode=True` flag to `attach` command. You then open another shell and connect by following the online instructions for interactive debugging from Tensorboard.

2.9 Docker Image for Development

To use the included docker image:

```
docker build -t hoomd-tf .
```

To run the container:

```
docker run --rm -it --cap-add=SYS_PTRACE --security-opt seccomp=unconfined \
 -v /insert/path/to/htf/:/srv/hoomd-blue/htf hoomd-tf bash
```

The `cap--add` and `security-opt` flags are optional and allow `gdb` debugging. Install `gdb` and `python3-dbg` packages to use `gdb` with the package.

Once in the container:

```
cd /srv/hoomd-blue && mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Debug\
    -DENABLE_CUDA=OFF -DENABLE_MPI=OFF -DBUILD_HPMC=off\
     -DBUILD_CGCMM=off -DBUILD_MD=on -DBUILD_METAL=off \
```

(continues on next page)

(continued from previous page)

```
    -DBUILD_TESTING=off -DBUILD_DEPRECATED=off -DBUILD_MPCD=OFF
make -j2
```

2.10 Change Log

2.10.1 v1.0.2 (2020-8-14)

Bug fixes

- EDS Bias was not using mean for computing offset
- EDS Bias was not resetting statistics correctly
- Removed stale test-cc directory that caused cmake errors

2.10.2 v1.0.1 (2020-7-27)

Bug fixes

- Prevented CPU overflow when nlist is too small (and added unit test)
- Adding check on mapping validity

2.10.3 v1.0 (2020-7-20)

JOSS Review

Hoomd-TF has been published as a peer-reviewed article in the Journal of Open Source Software (JOSS)

New Features

- Added pre-built molecular features
- Added MDAnalysis style selection for defining mapping operators

Enhancements

- Docs can now be built without HTF install
- mol batching performance is much better
- Simplified variable saving
- More example notebooks and reduced file sizes of example trajectories
- Supports dynamic boxes
- Better EDS bias naming
- Prevents accidentally computing forces from positions, instead of nlist
- Added guards against compiler mismatch
- Added sanity tests to prevent unsupported CPU/GPU modes
- Added benchmarking script
- Added check for nlist overflows on GPU
- Added check for mismatch on saving variables/tensors

- Fixed all compiler warnings
- Added Dockerfile for containerized use

Bug Fixes

- Empty tensorboard summaries no longer crash
- Prevented import issues with name clashes between packages and classes

2.10.4 v0.6 (2020-02-21)

Enhancements

- Migrated documentation to sphinx
- Added Jupyter notebook examples
- Various documentation improvements
- Added CUDA 10 Support

2.10.5 v0.5 (2019-10-17)

Bug Fixes

- Types are now correctly translated to TF

2.10.6 v0.4 (2019-09-25)

New Features

- Added experiment directed simulation biasing to *htf*.

Enhancements

- Added box dimension to computation graph (*graph.box* and *graph.box_size*)
- Can now wrap position derived distances with *graph.wrap_vector*
- Made it possible to specify period for *out_nodes*

Bug Fixes

- Fixed dangling list element in *rev_mol_indices*

2.10.7 v0.3 (2019-07-03)

Enhancements

- Batching by molecule now has a atom id to mol id/atom id look-up (*rev_mol_indices*)
- Version string is visible in package
- Example models now take an argument specifying where to save them
- When batching, atom sorting is automatically disabled
- *compute_pairwise_potential* now outputs force as well as potential

Bug Fixes

- Computing nlist in TF now correctly sorts when requested
- Fixed Mac OS specific issues for compiling against existing HOOMD-blue install
- Running mean computation variables are now marked as untrainable

2.10.8 v0.2 (2019-06-03)

New Features

- Added attach *batch_size* argument enabling batching of TF calls
- Can now batch by molecule, enabling selection/exclusion of molecules
- Added XLA option to improve TF speed
- Now possible to compile the plugin after hoomd-blue install
- Changed name of package to htf instead of tensorflow_plugin

Enhancements

- Changed output logging to only output TF items to the tf_manager.log and
- Log-level is now consistent with hoomd
- Added C++ unit tests skeleton in the same format as HOOMD-blue. Compile with -DBUILD_TESTING=ON to use.
- Switched to hoomd-blue cuda error codes
- Added MPI tests with domain decomposition
- Improved style consistency with hoomd-blue
- Cmake now checks for TF and hoomd versions while building hoomd-tf.

2.10.9 v0.1 (2019-04-22)

- Made Python packages actual dependencies.
- Switched to using hoomd-blue cuda error codes.
- Removed TaskLock from C++ code.
- Documentation updates
- Included license.
- User can now use specific hoomd forces in the hoomd2tf force mode.
- Added the ability to create a custom nlist.
- Made unit tests stricter and fixed some cuda synchronization bugs.
- Fixed TF GPU Compiling bug.
- Fixed ordering/masking error in mapping nlist and type of neighbor particles in nlist.
- Fixed a bug which caused a seg fault in nonlist settings.

2.11 Known Issues

The following is a list of known issues. To report another issue, please use the issue tracker.

2.11.1 Using Positions

Hoomd re-orders positions to improve performance. If you are using CG mappings that rely on ordering of positions, be sure to disable this:

```
c = hoomd.context.initialize()
c.sorter.disable()
```

2.11.2 Exploding Gradients

There is a bug in norms (https://github.com/tensorflow/tensorflow/issues/12071) that somtimes prevents optimizers to work well with TensorFlow norms. Note that this is only necessary if you're summing up gradients, like what is commonly done in computing gradients in optimizers. This isn't usually an issue for just computing forces. There are three ways to deal with this:

Small Training Rates

When Training something like a Lennard-Jones potential or other `1/r` potential, high gradients are possible. You can prevent expoding gradients by using small learning rates and ensuring variables are initialized so that energies are finite.

Safe Norm

There is a workaround (*graphbuilder.graph_builder.safe_norm()*) in Hoomd-TF. There is almost no performance penalty, so it is fine to replace `tf.norm` with *graphbuilder.graph_builder.safe_norm()* throughout. This method adds a small amount to all the norms though, so if you rely on some norms being zero it will not work well.

Clipping Gradients

Another approach is to clip gradients instead of using safe_norm:

```
optimizer = tf.train.AdamOptimizer(1e-4)
gvs = optimizer.compute_gradients(cost)
capped_gvs = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in gvs]
train_op = optimizer.apply_gradients(capped_gvs)
```

Neighbor Lists

Using a max-size neighbor list is non-ideal, especially in CG simulations where density is non-uniform.

2.12 htf

Details

HOOMD-blue plus TensorFlow for online machine learning in MD simulations. Supports GPU and CPU execution.

Modules

2.12.1 graph_builder

Details

class graphbuilder.**graph_builder**(*nneighbor_cutoff*, *output_forces=True*, *check_nlist=False*)

Build the TensorFlow graph that will be used during the HOOMD run.

Parameters

- **nneighbor_cutoff** (*int*) – The maximum number of neigbhors to consider (can be 0)
- **output_forces** (*bool*) – True if your graph will compute forces to be used in TensorFlow
- **check_nlist** (*bool*) – True will raise error if neighbor list overflows (nneighbor_cutoff too low)

build_mol_rep(*MN*)

This creates `mol_forces`, `mol_positions`, and `mol_nlist` which have dimensions mol_number x MN x 4 (`mol_forces`, `mol_positions`) and ? x MN x NN x 4 (`mol_nlist`) tensors batched by molecule, where MN is the number of molecules. MN is determined at run time. The MN must be chosen to be large enough to encompass all molecules. If your molecule is 6 atoms and you chose MN=18, then the extra entries will be zeros. Note that your input should be 0 based, but subsequent tensorflow data will be 1 based, since 0 means no atom. The specification of what is a molecule will be passed at runtime, so that it can be dynamic if desired.

To convert a mol_quantity to a per-particle quantity, call `scatter_mol_quanitity(mol_quantity)`

Parameters **MN** (*int*) – The number of molecules

Returns None

compute_forces(*energy*, *virial=None*, *positions=False*, *nlist=None*)

Computes pairwise or position-dependent forces (field) given a potential energy function that computes per-particle or overall energy

Parameters

- **energy** (*tensor*) – The potential energy
- **virial** (*bool*) – Defaults to `None`. Virial contribution will be computed if the graph outputs forces. Can be set manually instead. Note that the virial term that depends on positions is not computed.
- **positions** (*tensor*) – Defaults to `False`. Particle positions tensor to use for force calculations. If set to `True`, uses `self.positions`. If set to `False` (default), no position dependent forces will be computed. Only pairwise forces from neighbor list will be applied. If set to a tensor, that tensor will be used instead of `self.positions`.
- **nlist** (*tensor*) – Defaults to `None`. Particle-wise neighbor list to use for force calculations. If not specified, uses `self.nlist`.

Returns The TF force tensor. Note that the virial part will be stored as the class attribute `virial` and will be saved automatically.

compute_rdf(*r_range*, *name*, *nbins=100*, *type_i=None*, *type_j=None*, *nlist=None*, *positions=None*)

Computes the pairwise radial distribution function, and appends the histogram tensor to the graph's `out_nodes`.

Parameters

- **bins** – The bins to use for the RDF
- **name** – The name of the tensor containing rdf. The name will be concatenated with '-r' to create a tensor containing the r values of the rdf.
- **type_i** – Use this to select the first particle type.
- **type_j** – Use this to select the second particle type.
- **nlist** – Neighbor list to use for RDF calculation. By default it will use `self.nlist`.
- **positions** – Defaults to `self.positions`. This tensor is only used to get the origin particle's type. So if you're making your own, just make sure column 4 has the type index.

Returns Historgram tensor of the RDF (not normalized).

masked_nlist(*type_i=None*, *type_j=None*, *nlist=None*, *type_tensor=None*)

Returns a neighbor list masked by the given particle type(s).

Parameters

- **type_i** – Use this to select the first particle type.
- **type_j** – Use this to select a second particle type (optional).
- **nlist** – Neighbor list to mask. By default it will use `self.nlist`.
- **type_tensor** – An N x 1 tensor containing the type(s) of the nlist origin. If None, particle types from `self.positions` will be used.

Returns The masked neighbor list tensor.

nlist_rinv

Returns an N x NN tensor of 1 / r for each neighbor

running_mean(*tensor*, *name*, *batch_reduction='mean'*)

Computes running mean of the given tensor

Parameters

- **tensor** (*tensor*) – The tensor for which you're computing running mean
- **name** (*str*) – The name of the variable in which the running mean will be stored
- **batch_reduction** (*str*) – If the hoomd data is batched by atom index, how should the component tensor values be reduced? Options are 'mean' and 'sum'. A sum means that tensor values are summed across the batch and then a mean is taking between batches. This makes sense for looking at a system property like pressure. A mean gives a mean across the batch. This would make sense for a per-particle property.

Returns A variable containing the running mean

static safe_div(*numerator*, *denominator*, *delta=3e-06*, ***kwargs*)

Use this method to avoid nan forces if doing 1/r or equivalent force calculations. There are some numerical instabilities that can occur during learning when gradients are propagated. The delta is problem specific.

Parameters

- **numerator** (*tensor*) – The numerator.
- **denominator** (*tensor*) – The denominator.
- **delta** – Tolerance for magnitude that triggers safe division.

Returns The safe division op (TensorFlow operation)

static safe_norm(*tensor*, *delta=1e-07*, ***kwargs*)

There are some numerical instabilities that can occur during learning when gradients are propagated. The delta is problem specific. NOTE: delta of safe_div must be > sqrt(3) * (safe_norm delta) See *this TensorFlow issue <https://github.com/tensorflow/tensorflow/issues/12071>*.

Parameters

- **tensor** – the tensor over which to take the norm
- **delta** – small value to add so near-zero is treated without too much accuracy loss.

Returns The safe norm op (TensorFlow operation)

save(*model_directory*, *force_tensor=None*, *virial=None*, *out_nodes=None*, *move_previous=True*)

Save the graph model to specified directory.

Parameters

- **model_directory** – Multiple files will be saved, including a dictionary with information specific to hoomd-tf and TF model files.
- **force_tensor** – The forces that should be sent to hoomd
- **virial** – The virial which should be sent to hoomd. If None and you called compute_forces, then the virial computed from that function will be saved.
- **out_nodes** – Any additional TF graph nodes that should be executed. For example, optimizers, printers, etc. Each element of the list can itself be a list where the first element is the node and the second element is the period indicating how often to execute it.

Returns None

save_tensor(*tensor*, *name*, *save_period=1*)

Saves a tensor to a variable

Parameters

- **tensor** (*tensor*) – The tensor to save
- **name** (*str*) – The name of the variable which will be saved
- **save_period** (*int*) – How often to save the variable

Returns None

wrap_vector(*r*)

Computes the minimum image version of the given vector.

Parameters **r** (*tensor*) – The vector to wrap around the HOOMD box.

Returns The wrapped vector as a TF tensor

2.12.2 tfarraycomm

Overview

class tfarraycomm.**tf_array_comm**(*nparray*, *hoomd_context=<sphinx.ext.autodoc.importer._MockObject object>*)

Set up HOOMD context and interface with C++ memory.

Parameters

- **nparray** – numpy array to share with the C++ process
- **hoomd_context** – HOOMD execution configuration (defaults to `hoomd.context.exec_conf`)

getArray()

Convert C++ array into a numpy array.

Returns the converted numpy array

receive()

Receives the array from the C++ class instance.

send()

Sends the array to the C++ class instance.

2.12.3 tfcompute

Overview

Details

class tensorflowcompute.**tfcompute**(*tf_model_directory*, *log_filename='tf_manager.log'*, *device=None*, *bootstrap=None*, *bootstrap_map=None*, *_debug_mode=False*, *_mock_mode=False*, *write_tensorboard=False*, *use_xla=False*)

TensorFlow Computations for HTF.

Parameters

- **tf_model_directory** – Directory in which to save the TensorFlow model files.
- **log_filename** – Name to use for the TensorFlow log file.
- **device** – Device (GPU) on which to execute, if a specific one is desired.
- **bootstrap** – If set to a directory, will search for and load a previously-saved model file
- **bootstrap_map** – A dictionary to be used when bootstrapping, pairing old models' tensor variable names with new ones. Key is new name, value is older model's.
- **_deubug_mode** – Set this to True to see more debug messages.
- **_mock_mode** – Set this to True to run a "fake" calculation of forces that would be passed to the HOOMD simulation without applying them.
- **write_tensorboard** – If True, a tensorboard file will be written in the tf_model_directory.
- **use_xla** – If True, enables the accelerated linear algebra library in TensorFlow, which can be useful for large and complicated tensor operations.

attach(*nlist=None*, *r_cut=0*, *save_period=1000*, *period=1*, *feed_dict=None*, *mol_indices=None*, *batch_size=None*)

Attaches the TensorFlow instance to HOOMD. The main method of this class, this method sets up TensorFlow and gets HOOMD ready to interact with it.

Parameters

- **nlist** – The HOOMD neighbor list that will be used as the TensorFlow input.
- **r_cut** – Cutoff radius for neighbor listing.
- **save_period** – How often to save the TensorFlow data. Period here is measured by how many times the TensorFLow model is updated. See period.
- **period** – How many HOOMD steps should pass before updating the TensorFlow model. In combination with save_period, determines how many timesteps pass before TensorFlow saves its data (slow). For example, with a save_period of 200, a period of 4, TensorFlow will write to the tf_model_directory every 800 simulation steps.
- **feed_dict** – The dictionary keyed by tensor names and filled with corresponding values. See feed_dict in __init__.
- **mol_indices** – Molecule indices for each atom, identifying which molecule each atom belongs to.
- **batch_size** – The size of batches if we are using batching. Cannot be used if molecule-wise batching is active.

finish_update(*batch_index*, *batch_frac*)

Allow TF to read output and we wait for it to finish.

Parameters

- **batch_index** – index of batch to be processed
- **batch_frac** – fractional batch index, i.e. `batch_frac = batch_index / len(input)`

get_forces_array()

Retrieve forces array as numpy array

get_nlist_array()

Retrieve neighbor list array as numpy array

get_positions_array()

Retrieve positions array as numpy array

get_virial_array()

Retrieve virial array as numpy array

rcut()

Define the cutoff radius used in the neighbor list. Adapted from hoomd/md/pair.py

scalar4_vec_to_np(*array*)

Convert from scalar4 dtype to numpy array :param array: the scalar4 array to be processed

set_reference_forces(**forces*)

Sets the HOOMD reference forces to be used by TensorFlow. See C++ comments in TensorFlowCompute.h

shutdown_tf()

Shut down the TensorFlow instance.

2.12.4 tfmanager

Overview

class tfmanager.**TFManager**(*graph_info*, *device*, *q*, *positions_buffer*, *nlist_buffer*, *forces_buffer*, *box_buffer*, *virial_buffer*, *log_filename*, *dtype*, *debug*, *write_tensorboard*, *use_feed*, *bootstrap*, *primary*, *bootstrap_map*, *save_period*, *use_xla*, *mol_indices*)

TFManager manages TensorFlow training and coordinates communication with HOOMD. Sets up the TensorFlow graph, sets placeholder variable values, updates and saves the TensorFlow graph, passes output forces back to HOOMD, and writes TensorBoard files.

Parameters

- **graph_info** – The structure of the TensorFlow graph, passed as a dict. See tfcompute.py
- **device** – Which device to run on. See tfcompute.py
- **q** – Threading queue that is used during execution of TensorFlow.
- **positions_buffer** – Buffer where particle positions are stored
- **nlist_buffer** – Address of the neighbor list tensor
- **box_buffer** – Address of the box tensor
- **forces_buffer** – Address of the forces tensor
- **virial_buffer** – Address of the virial tensor
- **log_filename** – Name of the file to output tensorflow logs
- **dtype** – Data type for tensor values, e.g. int32, float32, etc
- **debug** – True to run TensorFlow in debug mode
- **write_tensorboard** – Whether to output a tensorboard file
- **use_feed** – Whether or not to use a feed dictionary of tensor values
- **bootstrap** – Location of previously-trained model files to load, otherwise None
- **primary** – Whether this is the 'primary' instance of TFManager. Only one instance writes logs and saves model files.
- **bootstrap_map** – A dictionary to be used when bootstrapping, pairing old models' tensor variable names with new ones. Key is new name, value is older model's.
- **save_period** – How often to save the TensorFlow data. Period here is measured by how many times the TensorFLow model is updated. See tfcompute.py.
- **use_xla** – If True, enables the accelerated linear algebra library in TensorFlow, which can be useful for large and complicated tensor operations.
- **mol_indices** – The molecucule indices, if doing mol batch (None otherwise)

start_loop()

start_loop method of tfmanager. Prepares GPU for execution, gathers trainable variables, sets up model saving, loads pre-trained variables, sets up tensorboard if requested and parses feed_dicts.

2.12.5 utils

Overview

This module contains various utility functions that are used throughout the `htf` module. These are imported automatically when `htf` is imported.

Details

`utils.`**`center_of_mass`**(*positions*, *mapping*, *box_size*, *name='center-of-mass'*)
Comptue mapping of the given positions (N x 3) and mapping (M x N) considering PBC. Returns mapped particles. :param positions: The tensor of particle positions :param mapping: The coarse-grain mapping used to produce the particles in system :param box_size: A list contain the size of the box [Lx, Ly, Lz] :param name: The name of the op to add to the TF graph

`utils.`**`compute_nlist`**(*positions*, *r_cut*, *NN*, *box_size*, *sorted=False*)
Compute particle pairwise neighbor lists.

Parameters

- **`positions`** – Positions of the particles
- **`r_cut`** – Cutoff radius (HOOMD units)
- **`NN`** – Maximum number of neighbors per particle
- **`box_size`** – A list contain the size of the box [Lx, Ly, Lz]
- **`sorted`** – Whether to sort neighbor lists by distance

Returns An [N X NN X 4] tensor containing neighbor lists of all particles and index

`utils.`**`compute_pairwise_potential`**(*model_directory*, *r*, *potential_tensor_name*, *checkpoint=-1*, *feed_dict={}*)
Compute the pairwise potential at r for the given model.

Parameters

- **`model_directory`** – The model directory
- **`r`** – A 1D grid of points at which to compute the potential.
- **`potential_tensor_name`** – The tensor containing potential energy.
- **`checkpoint`** – Which checkpoint to load. Default is -1, which loads latest checkpoint. An integer indicates loading from the model directory. If you pass a string, it is interpreted as a path.
- **`feed_dict`** – Allows you to add any other placeholder values that need to be added to compute potential in your model

Returns A tuple of 1D arrays. First is the potentials corresponding to the pairwise distances in r, second is the forces.

`utils.`**`eds_bias`**(*cv*, *set_point*, *period*, *learning_rate=1*, *cv_scale=1*, *name=None*)
This method computes and returns the Lagrange multiplier/EDS coupling constant (alpha) to be used as the EDS bias in the simulation.

Parameters

- **`cv`** – The collective variable which is biased in the simulation

- **set_point** – The set point value of the collective variable. This is a constant value which is pre-determined by the user and unique to each cv.
- **period** – Time steps over which the coupling constant is updated. HOOMD time units are used. If period=100 alpha will be updated each 100 time steps.
- **learninig_rate** – Learninig_rate in the EDS method.
- **cv_scale** – Used to adjust the units of the bias to HOOMD units.
- **name** – Name used as prefix on variables.

Returns Alpha, the EDS coupling constant.

utils.**find_molecules**(*system*)

Given a hoomd system, return a mapping from molecule index to particle index. This is a slow function and should only be called once.

Parameters **system** – The molecular system in HOOMD.

Returns A list of length L (number of molecules) whose elements are lists of atom indices

utils.**force_matching**(*mapped_forces*, *calculated_cg_forces*, *learning_rate=0.001*)

This will minimize the difference between the mapped forces and calculated CG forces using the Adam oprimizer.

Parameters

- **mapped_forces** (*tensor*) – A tensor with shape M x 3 where M is number of CG beads in the system. These are forces mapped from an all atom system.
- **calculated_cg_forces** (*tensor*) – A tensor with shape M x 3 where M is number of CG beads in the system. These are CG forces estimated using a function or a basis set.
- **learning_rate** (*float*) – The learning_rate for optimization

Returns optimizer, cost

utils.**load_variables**(*model_directory*, *names*, *checkpoint=-1*, *feed_dict={}*)

Adds variables from `model_directory` to the TF graph loaded from a checkpoint, optionally other than the most recent one, or setting values with a feed dict.

Parameters

- **model_directory** – Directory from which to load model variables.
- **names** – names of TensorFlow variables to load
- **checkpoint** – checkpoint number of the trained model to load from. Default value is -1 for most recently saved model.
- **feed_dict** – optionally, use a feed dictionary to populate the model

utils.**matrix_mapping**(*molecule*, *beads_distribution*)

This will create a M x N mass weighted mapping matrix where M is the number of atoms in the molecule and N is the number of mapping beads.

Parameters

- **molecule** – This is atom selection in the molecule (MDAnalysis Atoms object).
- **beads_distribution** – This is a list of beads distribution lists, Note that

each list should contain the atoms as strings just like how they appear in the topology file.

Returns An array of M x N.

utils.**mol_angle**(*mol_positions*, *type_i*, *type_j*, *type_k*)
This method calculates the bond angle given three atoms batched by molecule

mol_positions Positions tensor of atoms batched by molecules. Can be created by calling build_mol_rep() method in graphbuilder

type_i Index of the first atom (int type)

type_j Index of the second atom (int type)

type_k Index of the third atom (int type)

angles Tensor containing bond angles

utils.**mol_bond_distance**(*mol_positions*, *type_i*, *type_j*)
This method calculates the bond distance given two atoms batched by molecule

mol_positions Positions tensor of atoms batched by molecules. Can be created by calling build_mol_rep() method in graphbuilder

type_i Index of the first atom (int type)

type_j Index of the second atom (int type)

v_ij Tensor containing bond distances

utils.**mol_dihedral**(*mol_positions*, *type_i*, *type_j*, *type_k*, *type_l*)
This method calculates the dihedral angle given four atoms batched by molecule

mol_positions Positions tensor of atoms batched by molecules. Can be created by calling build_mol_rep() method in graphbuilder

type_i Index of the first atom (int type)

type_j Index of the second atom (int type)

type_k Index of the third atom (int type)

type_l Index of the fourth atom (int type)

dihedrals Tensor containing dihedral angles

utils.**run_from_trajectory**(*model_directory*, *universe*, *selection='all'*, *r_cut=10.0*, *period=10*, *feed_dict={}*)
This will process information from a trajectory and run the user defined model on the nlist computed from the trajectory.

Parameters

- **model_directory** (*string*) – The model directory
- **universe** – The MDAnalysis universe
- **selection** (*string*) – The atom groups to extract from universe
- **r_cut** (*float*) – The cutoff raduis to use in neighbor list calculations
- **period** (*int*) – Frequency of reading the trajectory frames
- **feed_dict** (*dict*) – Allows you to add any other placeholder values that need to be added to compute potential in your model

utils.**sparse_mapping**(*molecule_mapping*, *molecule_mapping_index*, *system=None*)

This will create the necessary indices and values for defining a sparse tensor in tensorflow that is a mass-weighted M x N mapping operator.

Parameters

- **molecule_mapping** – This is a list of L x M matrices, where M is the number of atoms in the molecule and L is the number of coarse-grain sites that should come out of the mapping. There should be one matrix per molecule. The ordering of the atoms should follow what is defined in the output from find_molecules
- **molecule_mapping_index** – This is the output from find_molecules. A list of length L (number of molecules) whose elements are lists of atom indices
- **system** – The hoomd system. This is used to get mass values for the mapping, if you would like to weight by mass of the atoms.

Returns A sparse tensorflow tensor of dimension N x N, where N is number of atoms

- search

Python Module Index

Index

T

U

W