Supporting Information for libmolgrid: GPU Accelerated Molecular Gridding for Deep Learning Applications

Jocelyn Sunseri and David R. Koes*

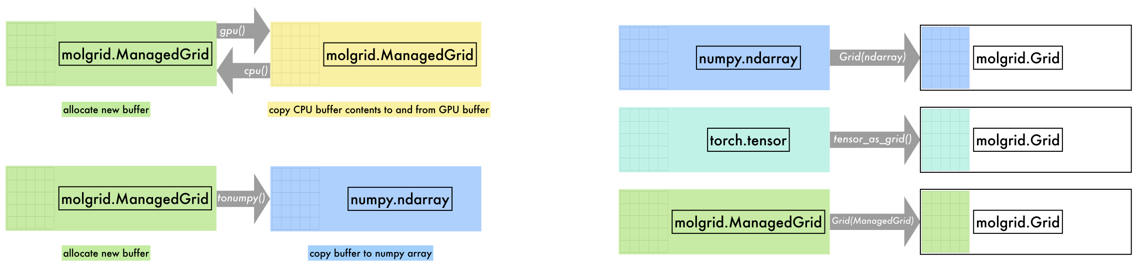
Department of Computational and Systems Biology, University of Pittsburgh, 3501 Fifth Ave., Pittsburgh, PA 15260, United States

E-mail: dkoes@pitt.edu

Grids

Figure S1 illustrates the behavior of `ManagedGrid`s (S1a) and `Grid`s (S1b). `ManagedGrid`s can migrate data between devices, and they create a copy when converting to or from other objects that have their own memory. `Grid`s do not own memory, instead serving as a view over the memory associated with another object that does; they do not create a copy of the buffer, rather they interact with the original buffer directly, and they cannot migrate it between devices.

The explicit specialization of a grid exposed in the Python `molgrid` API has a naming convention that specifies its dimensionality, underlying data type, and in the case of `Grid`s, the device where its memory buffer is located. The structure of the naming convention is `[GridClass][NumDims][DataType]["CUDA" if GridClass=="Grid" and DataLoc == "GPU"]`. Since `ManagedGrid`s can migrate their data from host to device, their names do not depend on any particular data location. For example, a 1-dimensional `ManagedGrid` of



(a) (b)

Figure S1: `ManagedGrid`s manage their own memory buffer, which can migrate data between the CPU and GPU and copy data to a NumPy array as shown in (a). `Grid`s are a view over a memory buffer owned by another object; they may be constructed from a Torch tensor, a `ManagedGrid`, or an arbitrary data buffer with a Python-exposed pointer, including a NumPy array as shown in (b).

type float is an `MGrid1f`, a 3-dimensional `Grid` of type float is a `Grid3f`, and a 5-dimensional `Grid` of type double that is a view over device data is a `Grid5dCUDA`.

Atom Typing

Table S1 provides an example of performing atom typing via a user-defined callback function. This example creates a vector atom typer using this method, but index typers are created in exactly the same manner, returning a single value rather than a vector.

ExampleProvider

Table S2 shows the options that can be set via the `ExampleProvider` constructor. Randomization is enabled with the `shuffle` option; oversampling of underrepresented classes to provide equal representation from all available classes categorized by the `Example` label is enabled with `balanced`; resampling based on a specific molecule associated with an `Example` (determined by the first filename encountered on a given metadata line) comes from `stratify_receptor` (as the name suggests, this is often used to sample equally from

Table S1: Examples of performing atom typing with a user-provided callback function.

```python
# create simple molecule, for demonstration purposes
m = pybel.readstring('smi','C')
m.addh()

# atom typing based on atomic number and valence, with a constant radius of 1.5
def mytyper(atom):
    if hasattr(atom, 'GetValence'):
        return ([atom.GetAtomicNum(),atom.GetValence()], 1.5)
    else:
        return ([atom.GetAtomicNum(),atom.GetExplicitDegree()], 1.5)

# create the typer; the explicit names may be omitted, in which case numerical names
# will be automatically created
t = molgrid.PythonCallbackVectorTyper(mytyper, 2, ["anum","valence"])
# get types for our simple molecule using our typer
types = [t.get_atom_type_vector(a.OBAtom) for a in m.atoms]
```

Table S2: Available arguments to `ExampleProvider` constructor, along with their default values.

```python
exprovider = molgrid.ExampleProvider(shuffle=False, balanced=False,
stratify_receptor=False, labelpos=0, stratify_pos=1, stratify_abs=True, stratify_min=0,
stratify_max=0, stratify_step=0, group_batch_size=1, max_group_size=0,
cache_structs=True, add_hydrogens=True, duplicate_first=False, num_copies=1,
make_vector_types=False, data_root="", recmolcache="", ligmolcache="")
```

`Example`s associated with different receptors); `labelpos` specifies the location of the binary classification label on each line of the metadata file, in terms of an index starting from 0 that numbers the entries on a line; `stratify_pos` similarly specifies the location of a regression target value that will be used to stratify `Example`s for resampling (for example a binding affinity); `stratify_abs` indicates that stratification of `Example`s based on a regression value will use the absolute value, which is useful when a negative value has a special meaning such as with a hinge loss; and `stratify_min`, `stratify_max`, and `stratify_step` are used to define the bins for numerical stratification of `Example`s.

Additional options provide customization for interpreting examples and optimizations for data I/O. When using a recurrent network for processing a sequence of data, such as the case of training with molecular dynamics frames, `group_batch_size` specifies the number of frames to propagate gradients through for truncated backpropagation through time and `max_group_size` indicates the total number of `Example`s associated with the largest `Example` group (e.g. the maximum number of frames). `add_hydrogens` will result in protonation of parsed molecules with OpenBabel. `duplicate_first` will clone the first `CoordinateSet` in an `Example` to be separately paired with each of the subsequent `CoordinateSet`s in that `Example` (e.g., a single receptor structure is replicated to match different ligand poses). `num_copies` emits the same example multiple times (this allows the same structure to be presented to the neural network using multiple transformations in a single batch). `make_vector_types` will represent types as a one-hot vector rather than a single index. `cache_structs` will keep coordinates in memory to reduce training time. `data_root` allows the user to specify a shared parent directory for molecular data files, which then allows the metadata file to specify the filenames as relative paths. Finally, `recmolcache` and `ligmolcache` are binary files that store an efficient representation of all receptor and ligand files to be used for training, with each structure stored only once. These are created using the `create_caches2.py` script from `https://github.com/gnina/scripts`. Caches combine many small files into one memory mapped file resulting in a substantial I/O performance improvement and reduction in mem-

Table S3: Available arguments to the `GridMaker` constructor, along with their default values.

```
gmaker = molgrid.GridMaker(resolution=0.5, dimension=23.5, binary=False,
radius_type_indexed=False, radius_scale=1.0, gaussian_radius_multiple=1.0)
```

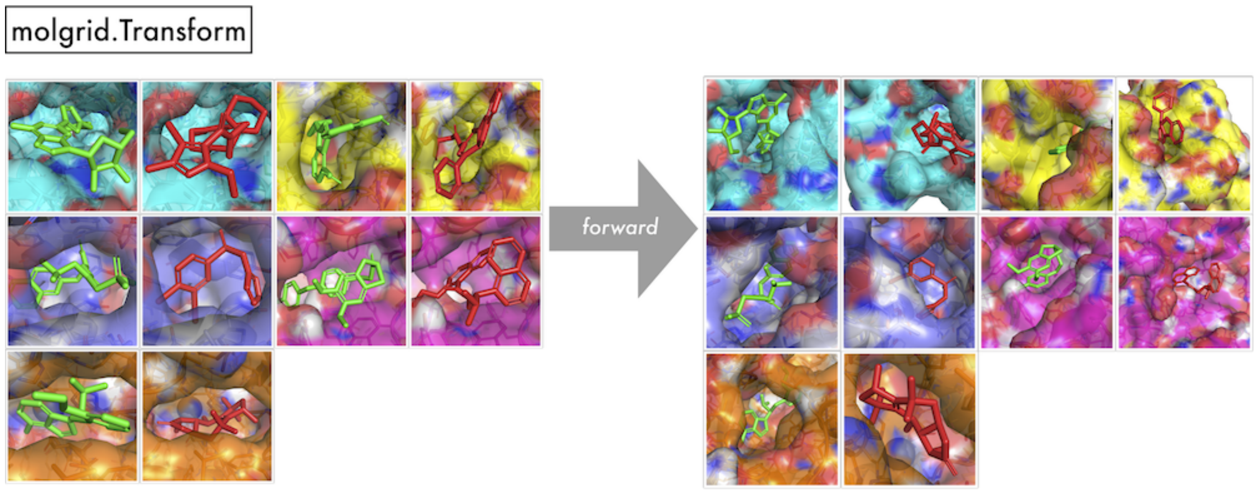
ory usage during training.

GridMaker

Table S3 shows the available arguments to the `GridMaker` constructor. `GridMaker` options include the grid resolution; dimension along each side of the cube; whether to constrain atom density values to be a binary indicator of overlapping an atom, rather than the default of a Gaussian to a multiple of the atomic radius (call this grm) and then decaying to 0 quadratically at $\frac{1+2grm^2}{2grm}$; whether to index the atomic radius array by type id (for vector types); a real-valued pre-multiplier on atomic radii, which can be used to change the size of atoms; and, if using real-valued atomic densities (rather than the alternative binary densities), the multiple of the atomic radius to which the Gaussian component of the density extends.

Transform

In order to provide some more detail about specialized use of `molgrid::Transform`, Figure S2 shows the behavior of `Transform::forward`, taking an input `Example` and returning a transformed version of that `Example` in `transformed_example`. Usage examples for the `Transformer` constructors are shown in Table S4.



```python
for data in batch:
    t = molgrid.Transform(center=(0,0,0), random_translate=2.0, random_rotation=True)
    t.forward(data, transformed_data, dotranslate=True)
    # do something with transformed_data
```

Figure S2: An illustration of `molgrid::Transform` usage, applying a distinct random rotation and translation to each of ten input examples. These transformations can also be applied separately to individual coordinate sets. Transformations to grids being generated via a `molgrid::GridMaker` can be generated automatically by specifying `random_rotation=True` or `random_translation=True` when calling `Gridmaker::Forward`.

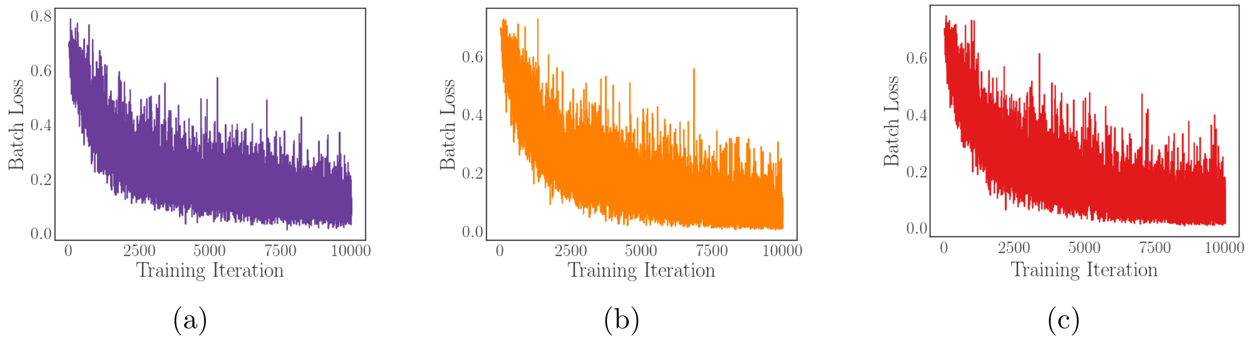


Figure S3: Loss per iteration while training a simple model, with input gridding and transformations performed on-the-fly with `libmolgrid` and neural network implementation performed with (a) Caffe, (b) PyTorch, and (c) Keras with a Tensorflow backend.

Table S4: Available `Transform` constructors.

```python
# Usage 1: specify a center, maximum distance for random translation,
# and whether to randomly rotate
transform1 = molgrid.Transform(center=molgrid.float3(0.0,0.0,0.0), random_translate=0.0,
random_rotation=False)

qt = molgrid.Quaternion(1.0, 0.0, 0.0, 0.0)
center = molgrid.float3(0.0, 0.0, 0.0)
translate = molgrid.float3(0.0, 0.0, 0.0)
# Usage 2: specify a particular rotation, to be performed around the molecule's center
transform2 = molgrid.Transform(qt)
# Usage 3: specify a particular rotation and the center around which it will be performed
transform3 = molgrid.Transform(qt, center)
# Usage 4: specify a particular rotation and center, along with a specific translation
transform4 = molgrid.Transform(qt, center, translate)
```

Results

Figure S3 shows successful training of a basic feed-forward network on a toy dataset using each of these three deep learning frameworks to perform binary classification of active versus inactive binding modes. Timing calculations for the main text performance figures were performed using GNU `time`, while memory utilization was obtained with `nvidia-smi -q -i 1 -d MEMORY -l 1`. The Caffe data was obtained using `caffe train` with the model at `https://github.com/gnina/models/blob/master/affinity/affinity.model` with the affinity layers removed; the PyTorch data was obtained using `https://gnina.github.io/libmolgrid/tutorials/train_basic_CNN_with_PyTorch.html`, run for 10,000 iterations; and the Keras data was obtained using `https://gnina.github.io/libmolgrid/tutorials/train_basic_CNN_with_Tensorflow.html`, run for 10,000 iterations. The metadata file for training is at `https://github.com/gnina/libmolgrid/blob/master/test/data/small.types`, using structures found at `https://github.com/gnina/libmolgrid/tree/master/test/data/structs`. The Cartesian reduction example can be found at `https://gnina.github.io/libmolgrid/tutorials/train_simple_cartesian_reduction.html`.