gnina/libmolgrid: Versatile grid-based molecular input library optimized for machine learning

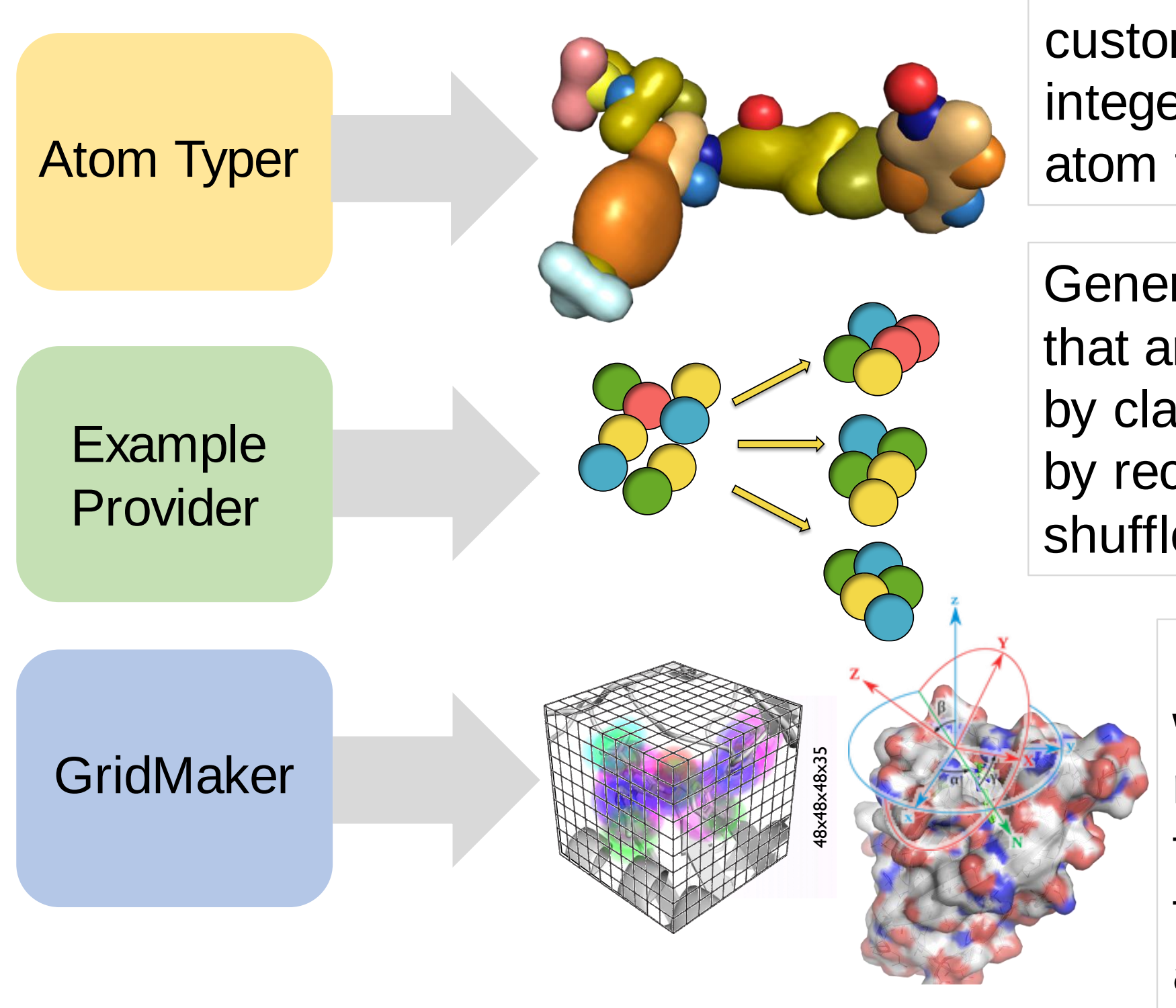
Jocelyn Sunseri[1,2] and David Ryan Koes[2]
[1]Carnegie Mellon - University of Pittsburgh Joint Program in Computational Biology,
[2]Department of Computational and Systems Biology University of Pittsburgh

Technical Details

Three-dimensional interactions between atoms are the conceptual foundation of molecular modeling. Existing deep learning frameworks specialized for molecular modeling focus on lower dimensional projections of these interactions, for example limiting their scope to graph-based or scalar distance-based representations that fundamentally fail to capture the full space of features that govern molecular behavior.

gnina/libmolgrid is an **open source** library that exposes three-dimensional grid-based molecular modeling capabilities in Python. It can interface with popular deep learning frameworks including Caffe, PyTorch, and Keras. It has built in support for gridding temporal data from molecular dynamics simulations and processing it with one of the RNN classes provided by the chosen deep learning package. The efficiency of our high resolution three-dimensional grid-based input representation is made possible by fully leveraging CUDA for gridding and gradient propagation.

Library Components

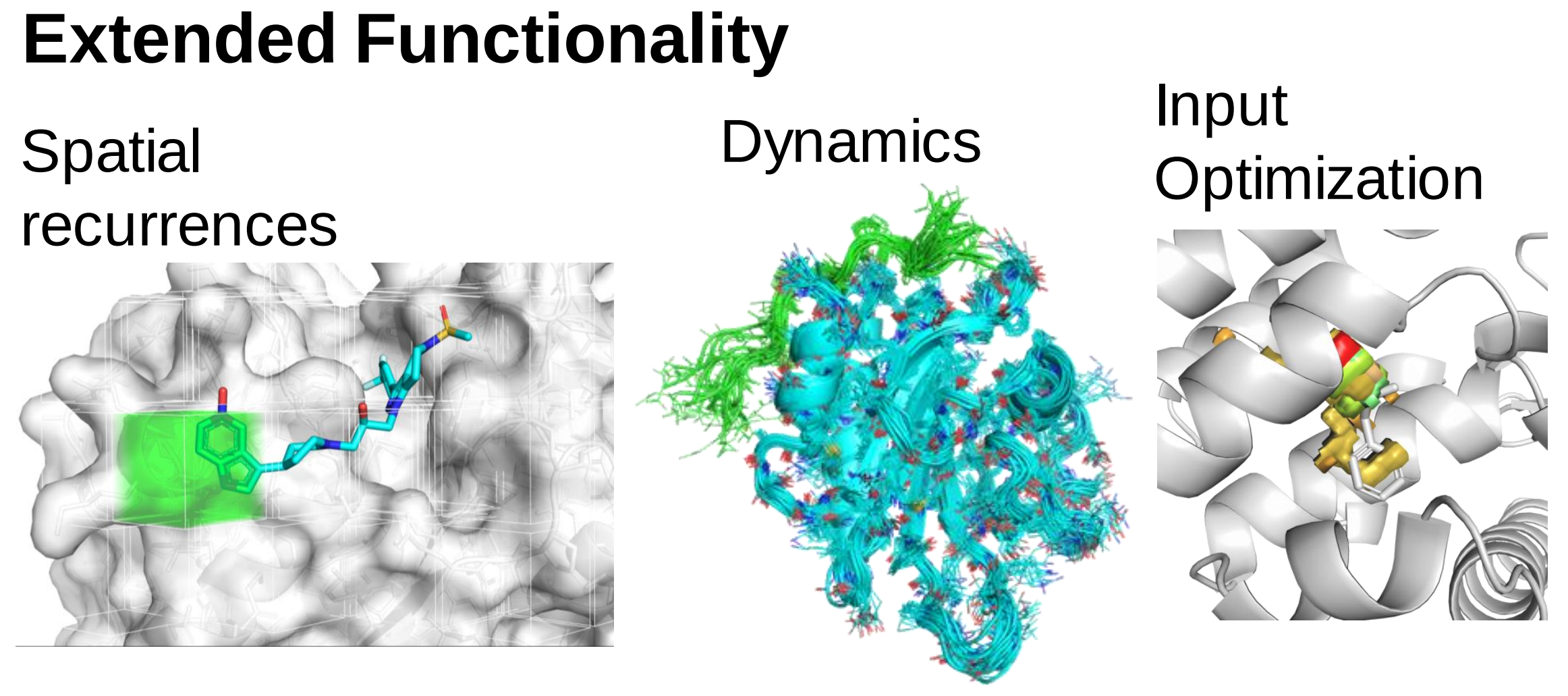


Flexible, customizable integer and vector atom typing

Generate batches that are balanced by class, stratified by receptor, shuffled, and more

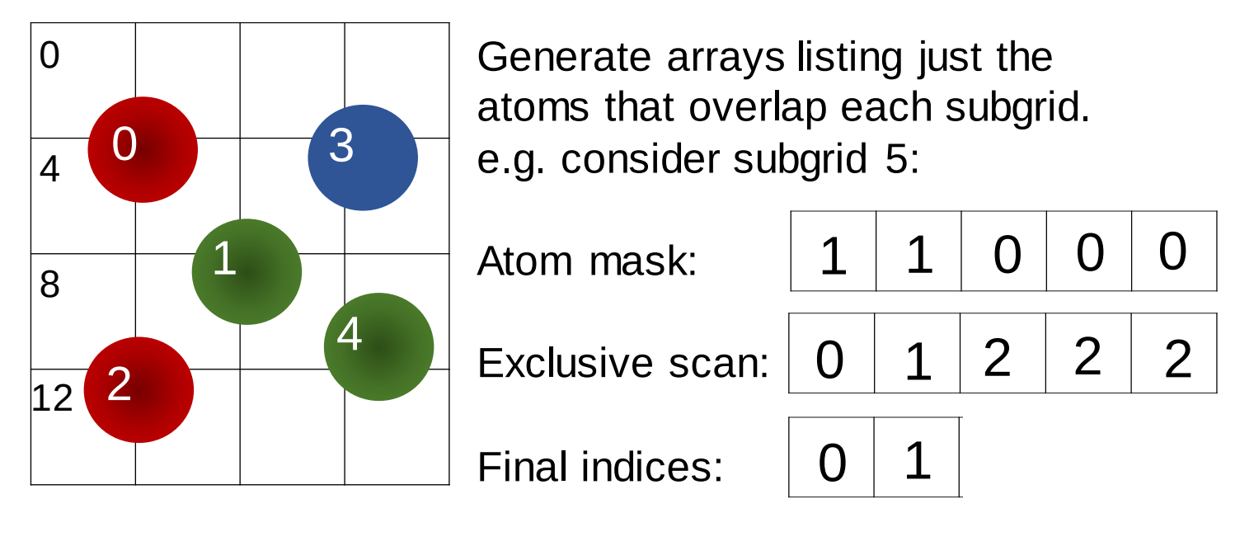
Fast gridding with optional random transformations for data augmentation & regularization

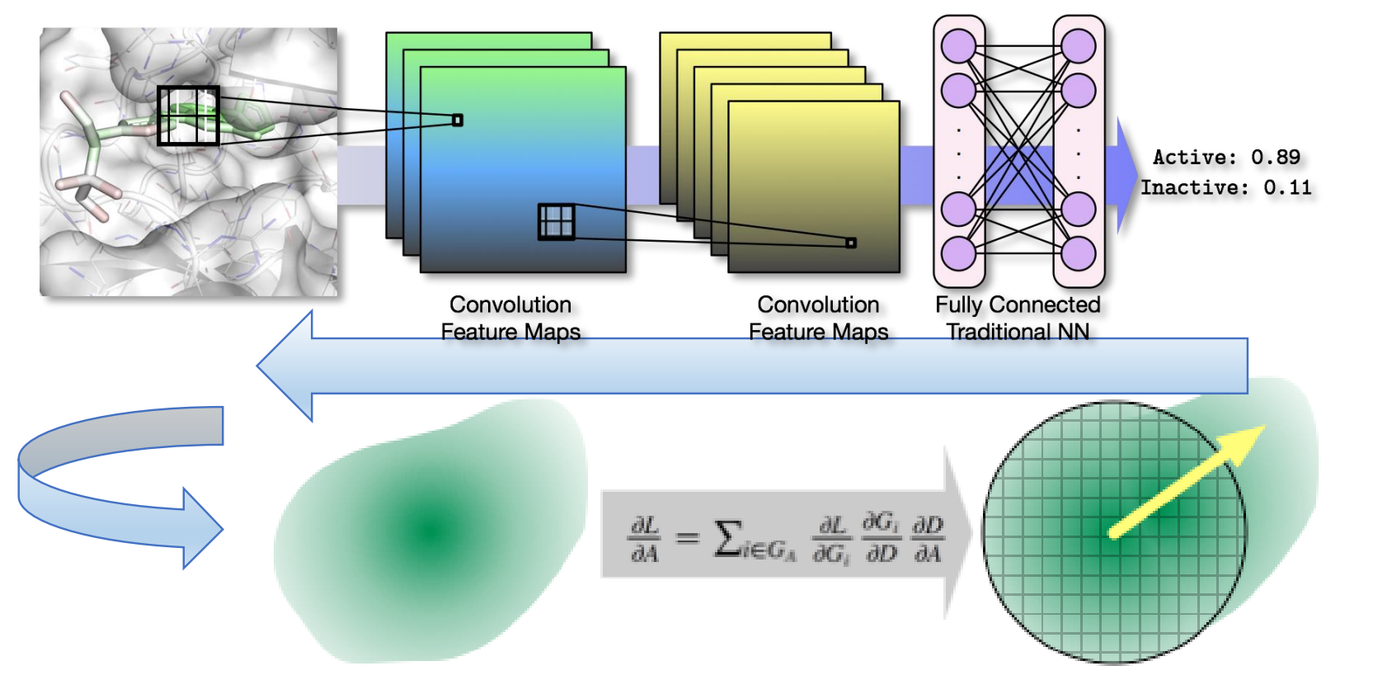
Extended Functionality



These features are fully supported in the Caffe-based interface, and can be readily implemented by users who prefer other supported frameworks via manipulations of generated grids. They include (1) decomposition of input into spatial recurrences, (2) processing of temporally sequential data such as molecular dynamics trajectory frames, and (3) optimizing input grids using a trained network to generate novel ligand density.

GPU Gridding and Gradients


0
4
8
12
0
1
2
3
4
Generate arrays listing just the atoms that overlap each subgrid. e.g. consider subgrid 5:
Atom mask: 1 1 0 0 0
Exclusive scan: 0 1 2 2 2
Final indices: 0 1


Convolution Feature Maps
Convolution Feature Maps
Fully Connected Traditional NN
Active: 0.89
Inactive: 0.11
$\frac{\partial L}{\partial A} = \sum_{i \in G_A} \frac{\partial L}{\partial G_i}\frac{\partial G_i}{\partial D}\frac{\partial D}{\partial A}$

We use a two-step parallel approach to rapidly generate four dimensional grids of atom density suitable for use as a neural network input. First, subgrids of spatially adjacent threads parallelize over the atom list to compute the reduced set of atoms that possibly overlap their region. Then they parallelize over grid points, using the greatly reduced atom list to optimize this atom density calculation. The accumulation of atomic gradients also uses parallel reductions to leverage the performance of the GPU.

Caffe

Caffe is the original deep learning framework with which we designed our library to be compatible, and all extended features are currently available when using Caffe. This includes several pre-trained and validated models and support for LSTM processing of MD trajectories and sub-grid spatial recurrences.

Sample Code

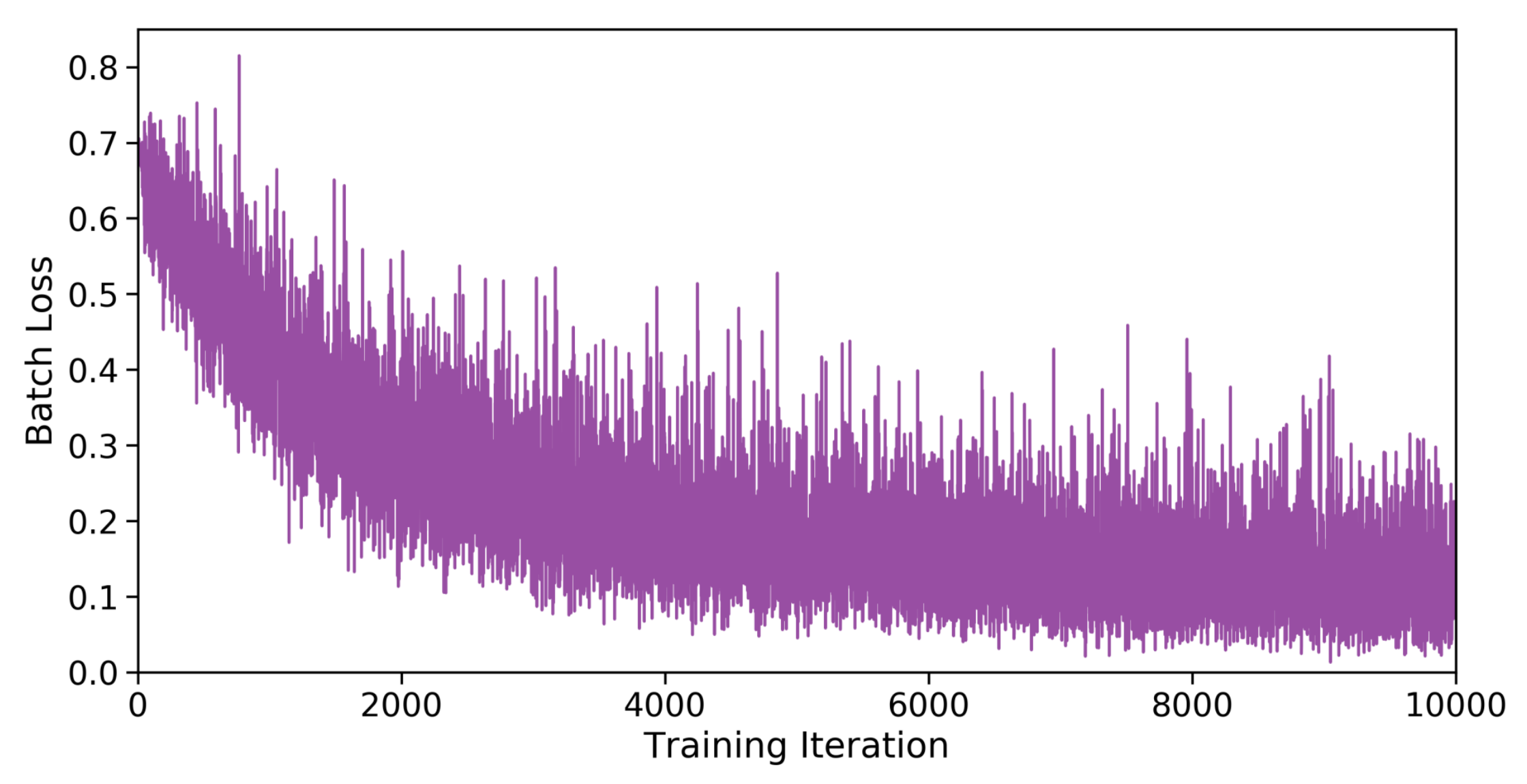
```
layer {
 name: "data"
 type: "MolGridData"
 top: "data"
 top: "label"
 molgrid_data_param {
  source: "small.types"
  batch_size: 50
  dimension: 23.5
  resolution: 0.5
  shuffle: true
  balanced: true
  random_rotation: true
  ligmap: "lig"
  recmap: "rec"
  root_folder: "data/structs"
 }
}
layer {
 name: "unit1_pool"
 type: "Pooling"
 bottom: "data"
 top: "unit1_pool"
 pooling_param {
  pool: MAX
  kernel_size: 2
  stride: 2
 }
}
layer {
 name: "unit1_conv"
 type: "Convolution"
 bottom: "unit1_pool"
 top: "unit1_conv"
 convolution_param {
  num_output: 32
  pad: 1
  kernel_size: 3
  stride: 1
  weight_filler {
   type: "xavier"
  }
 }
}
```

```
layer {
 name: "unit1_func"
 type: "ReLU"
 bottom: "unit1_conv"
 top: "unit1_conv"
}

layer {
 name: "unit2_pool"
 type: "Pooling"
 bottom: "unit1_conv"
 top: "unit2_pool"
 pooling_param {
  pool: MAX
  kernel_size: 2
  stride: 2
 }
}

layer {
 name: "unit2_conv"
 type: "Convolution"
 bottom: "unit2_pool"
 top: "unit2_conv"
 convolution_param {
  num_output: 64
  pad: 1
  kernel_size: 3
  stride: 1
  weight_filler {
   type: "xavier"
  }
 }
}

layer {
 name: "unit2_func"
 type: "ReLU"
 bottom: "unit2_conv"
 top: "unit2_conv"
}
```

```
layer {
 name: "unit3_pool"
 type: "Pooling"
 bottom: "unit2_conv"
 top: "unit3_pool"
 pooling_param {
  pool: MAX
  kernel_size: 2
  stride: 2
 }
}

layer {
 name: "unit3_conv"
 type: "Convolution"
 bottom: "unit3_pool"
 top: "unit3_conv"
 convolution_param {
  num_output: 128
  pad: 1
  kernel_size: 3
  stride: 1
  weight_filler {
   type: "xavier"
  }
 }
}

layer {
 name: "unit3_func"
 type: "ReLU"
 bottom: "unit3_conv"
 top: "unit3_conv"
}
```

```
layer {
 name: "pose_output"
 type: "InnerProduct"
 bottom: "unit3_conv"
 top: "pose_output"
 inner_product_param {
  num_output: 2
  weight_filler {
   type: "xavier"
  }
 }
}

layer {
 name: "loss"
 type: "SoftmaxWithLoss"
 bottom: "pose_output"
 bottom: "label"
 top: "loss"
}

layer {
 name: "output"
 type: "Softmax"
 bottom: "pose_output"
 top: "output"
}
```

Caffe Training



PyTorch

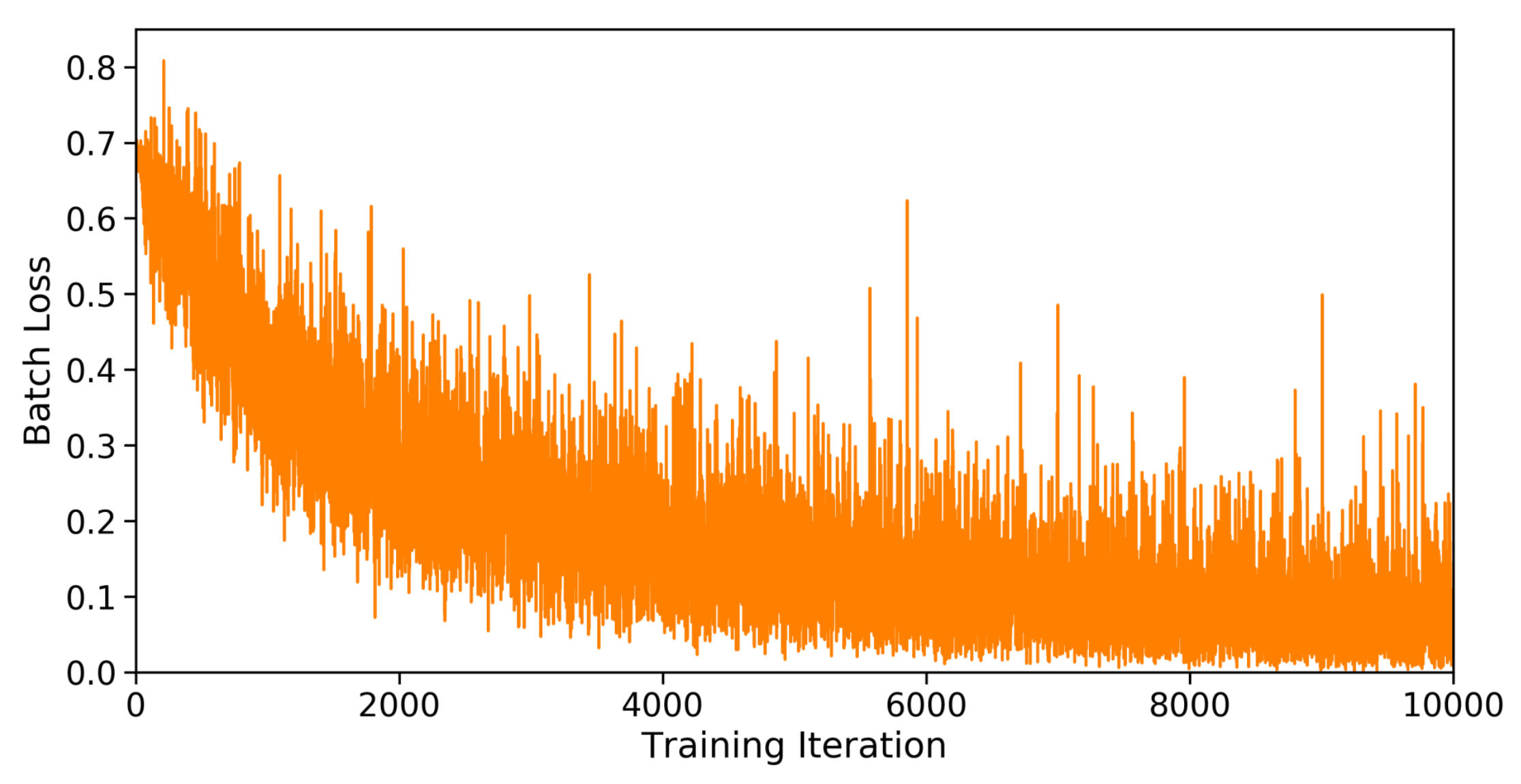
PyTorch support includes the ability to generate appropriate three-dimensional grid-based inputs for molecular modeling classification, regression, and generative modeling. The user can access the generated grids to extend their applications to temporal and recurrent spatial modeling.

Sample Code

```
import molgrid
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.nn import init
import os

molgrid.set_random_seed(0)
torch.manual_seed(0)
np.random.seed(0)

class Net(nn.Module):
  def __init__(self, dims):
    super(Net, self).__init__()
    self.pool0 = nn.MaxPool3d(2)
    self.conv1 = nn.Conv3d(dims[0], 32, kernel_size=3, padding=1)
    self.pool1 = nn.MaxPool3d(2)
    self.conv2 = nn.Conv3d(32, 64, kernel_size=3, padding=1)
    self.pool2 = nn.MaxPool3d(2)
    self.conv3 = nn.Conv3d(64, 128, kernel_size=3, padding=1)

    self.last_layer_size =
      dims[1]//8 * dims[2]//8 * dims[3]//8 * 128
    self.fc1 = nn.Linear(self.last_layer_size, 2)

  def forward(self, x):
    x = self.pool0(x)
    x = F.relu(self.conv1(x))
    x = self.pool1(x)
    x = F.relu(self.conv2(x))
    x = self.pool2(x)
    x = F.relu(self.conv3(x))
    x = x.view(-1, self.last_layer_size)
    x = self.fc1(x)
    return x

 def weights_init(m):
   if isinstance(m, nn.Conv3d) or isinstance(m, nn.Linear):
    init.xavier_uniform_(m.weight.data)
```

```
batch_size = 50
datadir = os.path.dirname(__file__)+'/data'
fname = datadir+"/small.types"

e = molgrid.ExampleProvider(
  data_root=datadir+"/structs",
  balanced=True,shuffle=True)
e.populate(fname)

gmaker = molgrid.GridMaker()
dims = gmaker.grid_dimensions(e.type_size())
tensor_shape = (batch_size,)+dims

model = Net(dims).to('cuda')
model.apply(weights_init)

optimizer = optim.SGD(model.parameters(),
  lr=0.01, momentum=0.9)

input_tensor = torch.zeros(tensor_shape,
  dtype=torch.float32, device='cuda')
float_labels = torch.zeros(batch_size,
  dtype=torch.float32)

losses = []
for iteration in range(10000):
  #load data
  batch = e.next_batch(batch_size)
  gmaker.forward(batch, input_tensor,
    random_translation=0, random_rotation=True)
  batch.extract_label(0, float_labels)
  labels = float_labels.long().to('cuda')

  optimizer.zero_grad()
  output = model(input_tensor)
  loss = F.cross_entropy(output,labels)
  loss.backward()
  optimizer.step()
  losses.append(float(loss))
```

PyTorch Training



Keras

Keras support includes the ability to generate appropriate three-dimensional grid-based inputs for molecular modeling classification, regression, and generative modeling. The user can access the generated grids to extend their applications to temporal and recurrent spatial modeling.

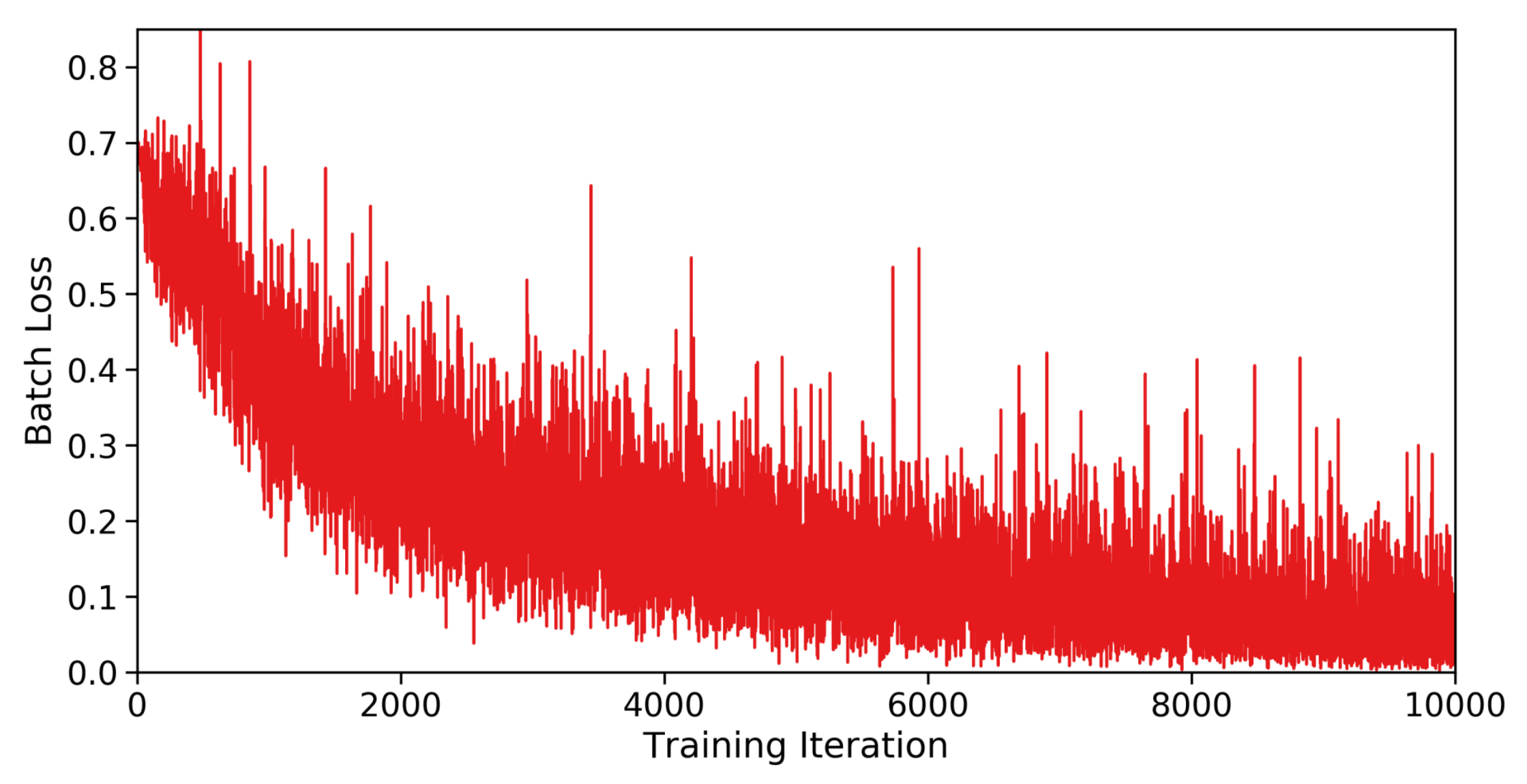
Sample Code

```
import molgrid
import numpy as np
import tensorflow as tf
import tensorflow.keras as keras
import keras.layers
import os

molgrid.set_random_seed(0)
np.random.seed(0)


def create_model(dims):
  """ Creates a 3D CNN by defining and applying layers """

  input_layer = keras.layers.Input(shape=dims)
  pool0 = keras.layers.MaxPooling3D(
    data_format="channels_first")(input_layer)
  conv1 = keras.layers.Conv3D(filters=32, kernel_size=3,
    data_format="channels_first", activation="relu")(pool0)
  pool1 = keras.layers.MaxPooling3D(
    data_format="channels_first")(conv1)
  conv2 = keras.layers.Conv3D(filters=64, kernel_size=3
    data_format="channels_first", activation="relu")(pool1)
  pool2 = keras.layers.MaxPooling3D(
    data_format="channels_first")(conv2)
  conv3 = keras.layers.Conv3D(filters=128, kernel_size=3,
    data_format="channels_first", activation="relu")(pool2)

  flatten = keras.layers.Flatten(
    data_format="channels_first")(conv3)
  fc1 = keras.layers.Dense(2,activation='softmax')(flatten)

  # Define and return model
  model = keras.models.Model(inputs=input_layer, outputs=fc1)
  model.compile(optimizer=
    keras.optimizers.SGD(lr=0.01, momentum=0.9),
    loss="sparse_categorical_crossentropy")

  return model
```

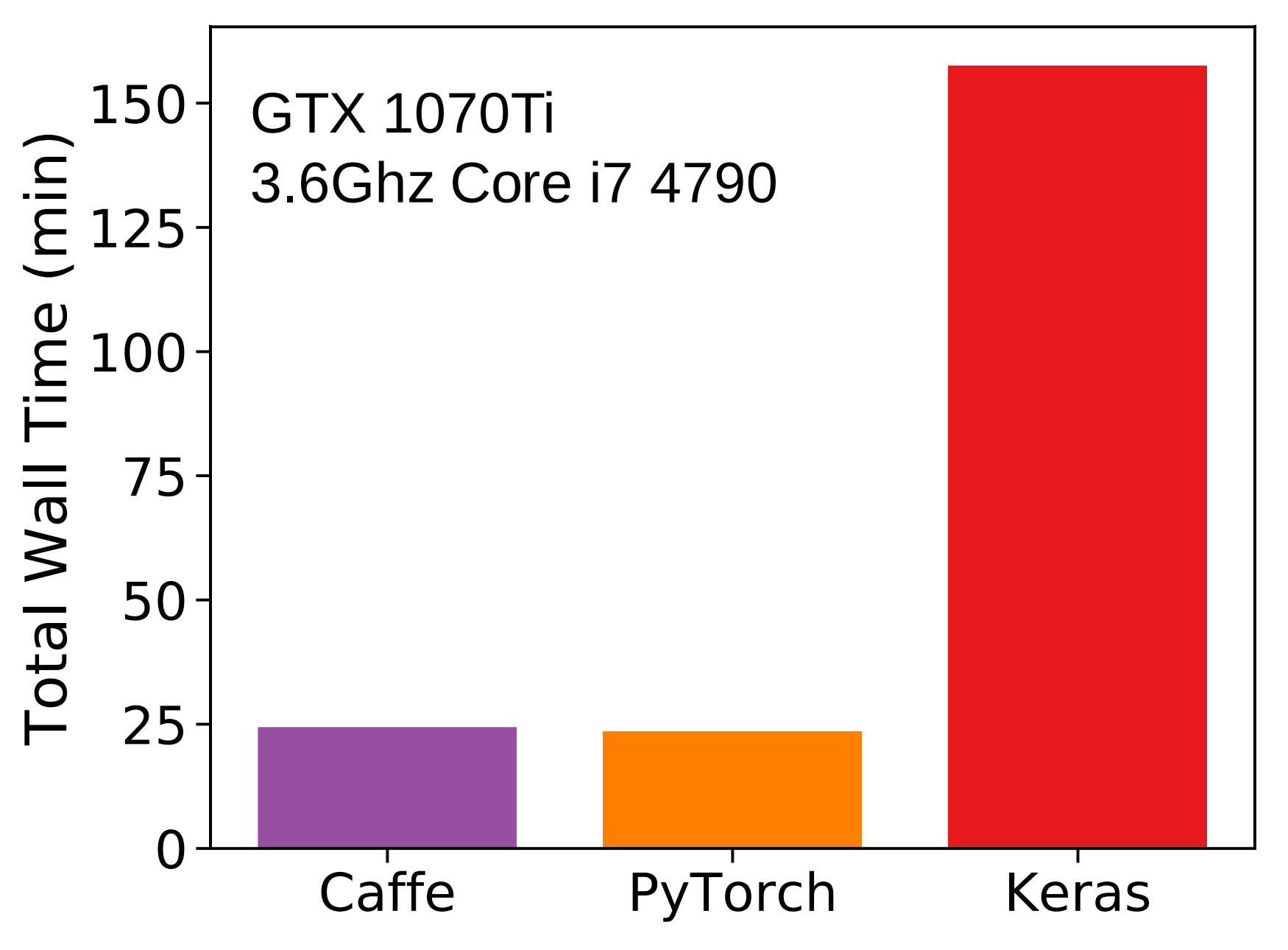
```
batch_size = 50
datadir = os.path.dirname(__file__)+'/data'
fname = datadir+"/small.types"

e = molgrid.ExampleProvider(
  data_root=datadir+"/structs",
  balanced=True,shuffle=True)
e.populate(fname)

gmaker = molgrid.GridMaker()
dims = gmaker.grid_dimensions(e.type_size())
tensor_shape = (batch_size,)+dims

model = create_model(dims)

labels = molgrid.MGrid1f(batch_size)
input_tensor = molgrid.MGrid5f(*tensor_shape)


losses = []
for iteration in range(10000):
  #load data
  batch = e.next_batch(batch_size)
  gmaker.forward(batch, input_tensor,
    random_translation=0, random_rotation=True)
  batch.extract_label(0, labels)

  loss = model.train_on_batch(input_tensor.tonumpy(),
                labels.tonumpy())
  losses.append(float(loss))
```

Inputs must reside in CPU memory, resulting in GPU→CPU→GPU data transfer

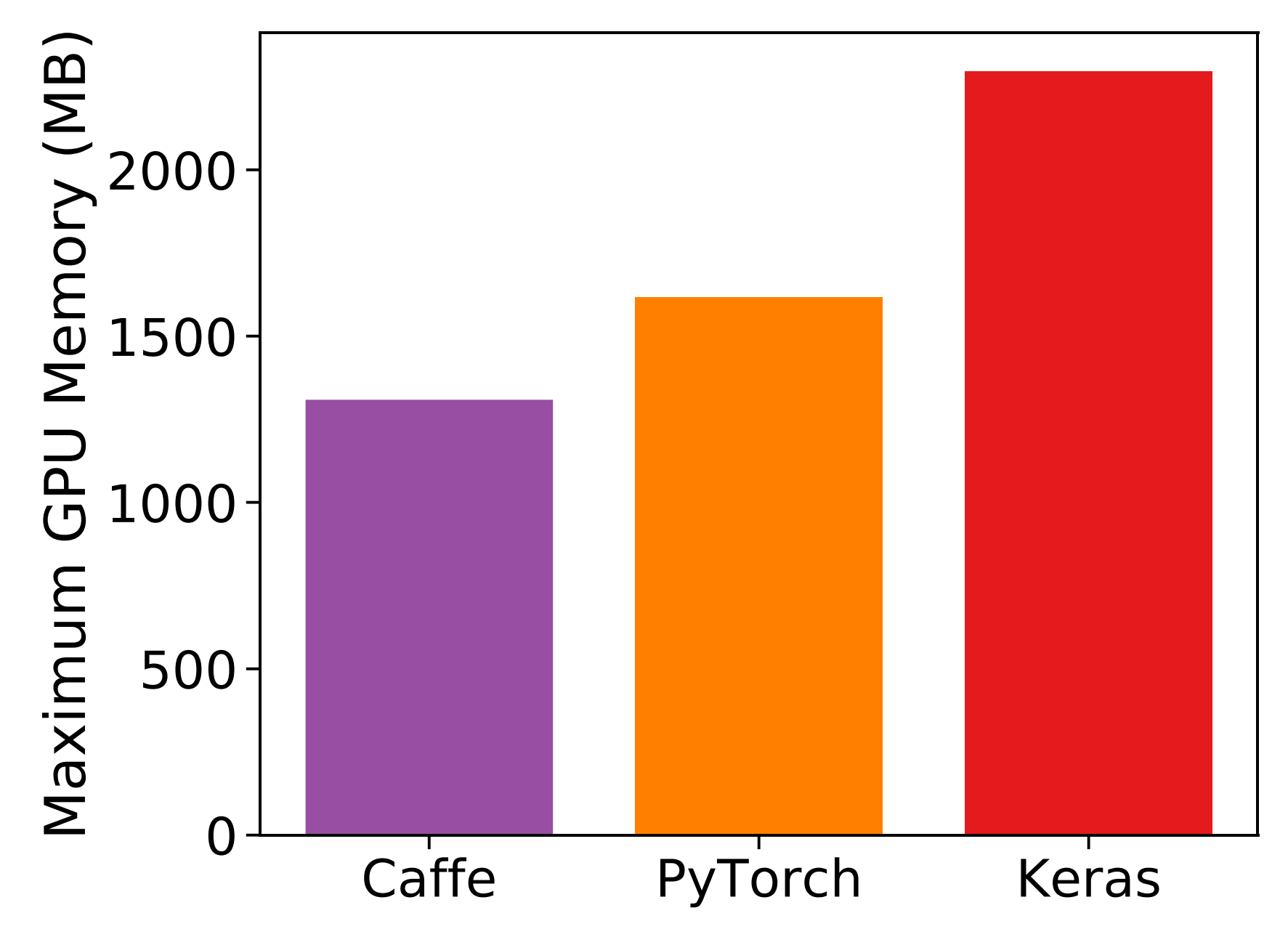
Keras Training



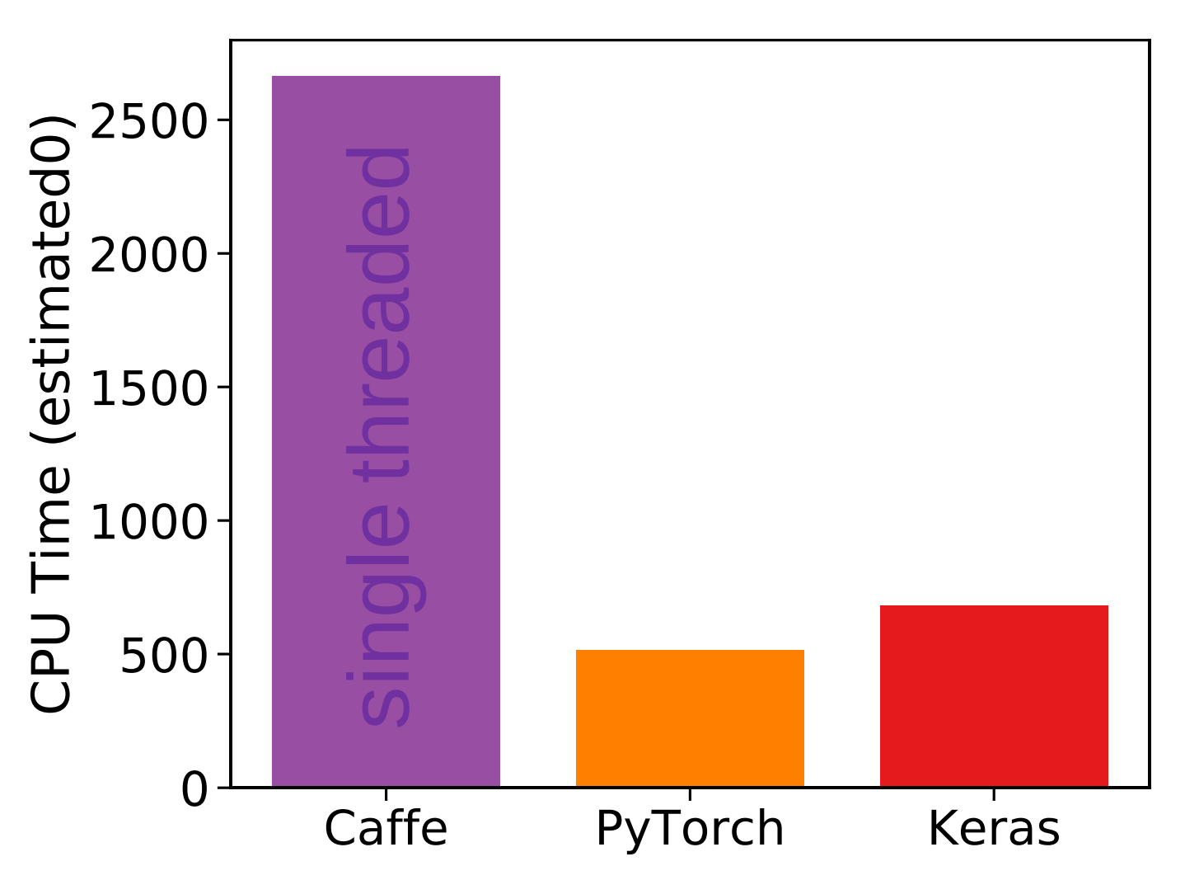
GPU Performance



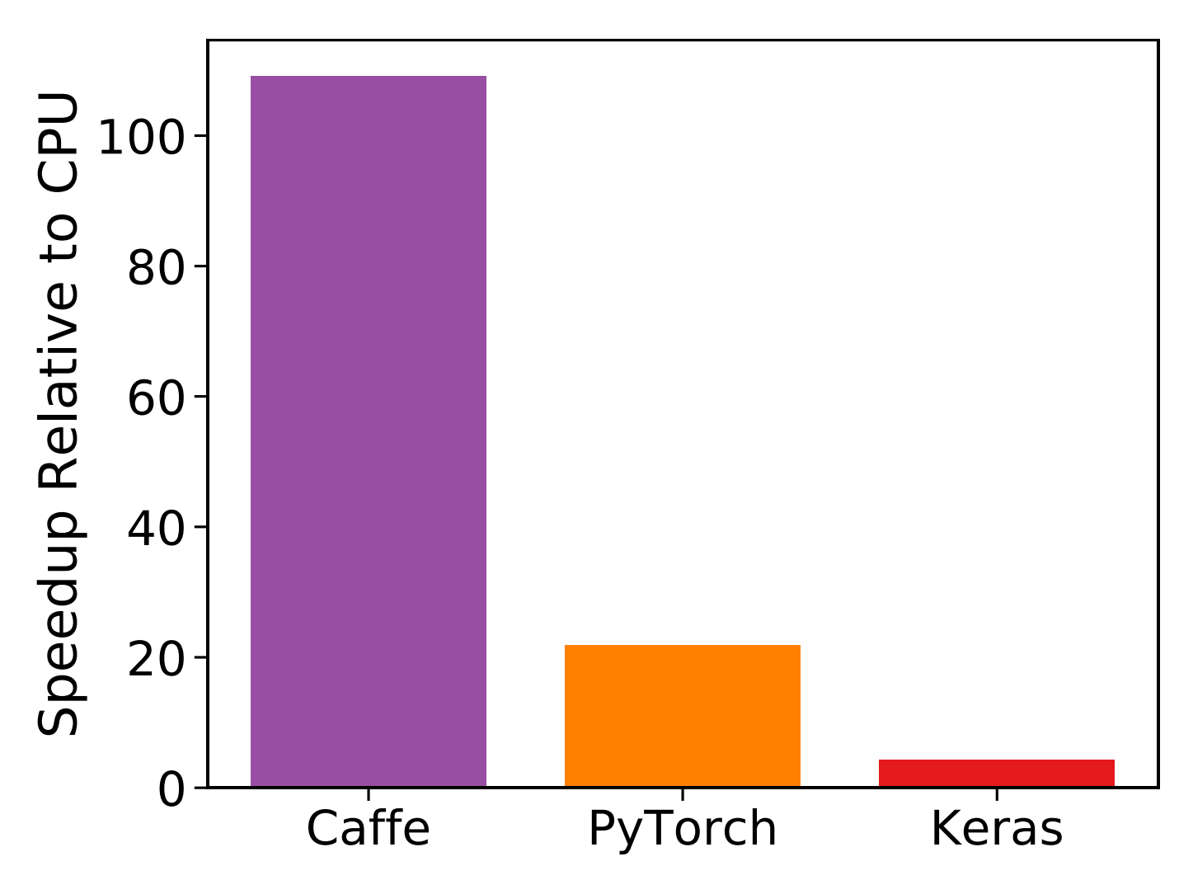
GPU Memory Utilization



CPU Performance



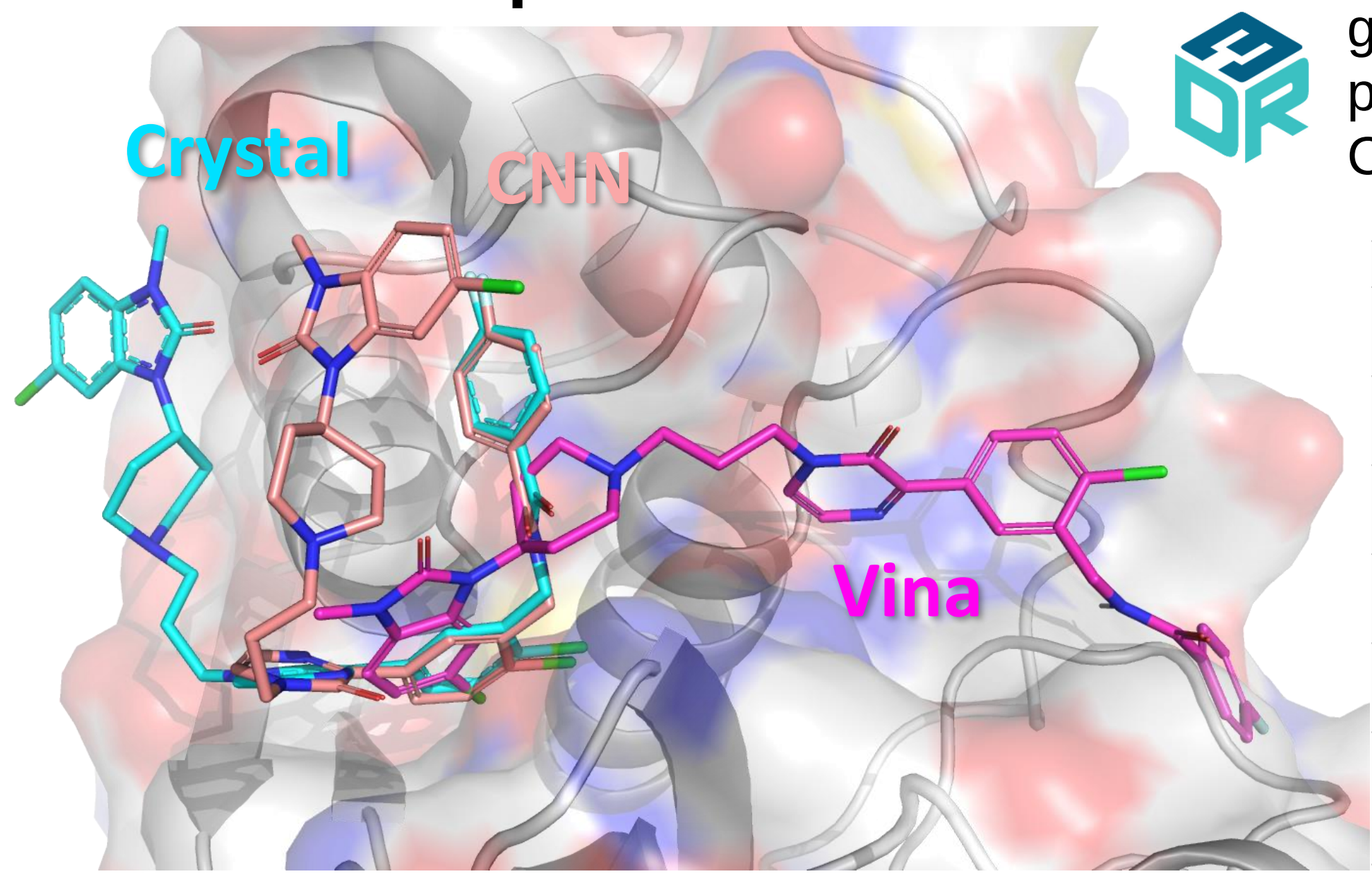
GPU Speedup vs CPU



In the Pipeline

- Native layers for all supported platforms for common operations
  - Internal gridding layers
  - Pre-implemented recurrent layers
- Memory mapped molecular caches for memory efficient and/or out-of-core training of large datasets
- Full support for vector atom typing
- Expand gridding options, especially to include spherical grids
- Enhanced documentation and tutorials

Blinded Independent Evaluation



As an objective measure of the application of grid-based CNNs to affinity ranking and pose prediction, we participated in the 2017 D3R Challenge, a blinded community benchmark.

Target	Rank	MCC	Method	Vina
JAK2 (SC2)	3/27	**0.44**	CNN affinity refine	0.07
VEGFR2	1/33	**0.53**	CNN scoring rescore	0.34
p38α	9/29	**0.21**	CNN affinity refine	0.15
JAK2 (SC3)	2/18	**0.23**	CNN affinity refine	-0.55
TIE2	1/17 (tie)	**0.78**	CNN affinity rescore	0.55
ABL1	N/A	0.56	CNN affinity rescore/refine (tie)	**1.00**

https://github.com/gnina/libmolgrid

References
Ragoza, M., Hochuli, J., Idrobo, E., Sunseri, J., & Koes, D. R. (2017). Protein–Ligand scoring with Convolutional neural networks. *Journal of chemical information and modeling*, 57(4), 942-957.
Sunseri, J., King, J. E., Francoeur, P. G., & Koes, D. R. (2018). Convolutional neural network scoring and minimization in the D3R 2017 community challenge. *Journal of computer-aided molecular design*, 1-16.
Ragoza M, Turner L, Koes DR. Ligand pose optimization with atomic grid-based convolutional neural networks. arXiv preprint arXiv:1710.07400. 2017 Oct 20

This work is supported R01GM108340 from the National Institute of General Medical Sciences and a grant from Relay Therapeutics.