TorchMD-Net 2.0: Fast Neural Network Potentials for Molecular Simulations

Raul P. Pelaez[a],[†] Guillem Simeon[a],[†] Raimondas Galvelis,[†,‡] Antonio Mirarchi,[†] Peter Eastman,[¶] Stefan Doerr,[‡] Philipp Thölke,[‡] Thomas E. Markland,[¶] and Gianni De Fabritiis[*,†,‡,§]

†Computational Science Laboratory, Universitat Pompeu Fabra, Barcelona Biomedical Research Park (PRBB), C Dr. Aiguader 88, 08003 Barcelona, Spain.
‡Acellera Labs, C Dr Trueta 183, 08005, Barcelona, Spain
¶Department of Chemistry, Stanford University, Stanford, CA 94305, USA
§Institució Catalana de Recerca i Estudis Avançats (ICREA), Passeig Lluis Companys 23, 08010 Barcelona, Spain

E-mail: g.defabritiis@gmail.com

[a]R.P.P. and G.S. contributed equally to this work.

Abstract

Achieving a balance between computational speed, prediction accuracy, and universal applicability in molecular simulations has been a persistent challenge. This paper presents substantial advancements in the TorchMD-Net software, a pivotal step forward in the shift from conventional force fields to neural network-based potentials. The evolution of TorchMD-Net into a more comprehensive and versatile framework is highlighted, incorporating cutting-edge architectures such as TensorNet. This transformation is achieved through a modular design approach, encouraging customized applications within the scientific community. The most notable enhancement is a significant improvement in computational efficiency, achieving a very remarkable acceleration in the computation of energy and forces for Tensor-Net models, with performance gains ranging from 2x to 10x over previous, non-optimized, iterations. Other enhancements include highly optimized neighbor search algorithms that support periodic boundary conditions and smooth integration with existing molecular dynamics frameworks. Additionally, the updated version introduces the capability to integrate physical priors, further enriching its application spectrum and utility in research. The software is available at https://github.com/torchmd/torchmd-net.

1 Introduction

Neural Network Potentials (NNPs)[1–7] are emerging as a key approach in molecular simulations, striving to optimize the balance between computational efficiency, predictive accuracy, and generality.

Some software frameworks to facilitate the use of neural network potentials have been developed, such as SchNetPack,[8] TorchANI,[9] DeePMD-Kit,[10] and others. Among the first to appear, we released TorchMD-Net, initially designed for the Equivariant Transformer architecture[11] and a simpler invariant graph neural network tailored for neural network potentials for protein coarse-graining.[12] Over time, TorchMD-Net has expanded its model architectures to include TensorNet,[13] an $O(3)$-

1

equivariant message-passing neural network utilizing rank-2 Cartesian tensor representations which achieved state-of-the-art accuracy on benchmark datasets. The evolution motivated by the progressive incorporation of different architectures and the changes in the framework needed to accommodate these, positions TorchMD-Net not just as a standalone tool, but as a versatile library for the development of NNPs.

Efficiency has been at the forefront of recent enhancements to TorchMD-Net. Among the optimizations, CUDA graphs have been integrated, providing a performance boost, especially for smaller workloads. TorchMD-Net has also incorporated the latest versions of its key dependencies (mainly PyTorch[14] and PyTorch Lightning[15]), with a notable addition being the search for compatibility with the `torch.compile` submodule from PyTorch 2.0, a feature that compiles Just-In-Time (JIT) modules into optimized kernels. While TorchMD-Net has introduced low precision modes (i.e. `bfloat16`) primarily as an exploratory tool for researchers, high precision (`float64`) is also available for ensuring detailed correctness checks during prototyping.

The new technical enhancements include the introduction of periodic boundary conditions, a CUDA-optimized neighbor list, and memory-efficient dataset loaders. The inclusion of TorchMD-Net in the conda-forge[16] package repository and the release of the documentation[17] are steps taken to enhance its accessibility to researchers. Another feature is TorchMD-Net's capacity to blend empirical physical knowledge into NNPs via priors. The integration of atom-wise and molecule-wise priors, such as the Ziegler-Biersack-Littmark[18] and Coulomb potentials, allows for a more nuanced approach in simulations.

TorchMD-Net emphasizes compatibility with leading molecular dynamics (MD) packages, especially with OpenMM.[19] OpenMM, widely recognized in the computational chemistry field, can now interface directly with TorchMD-Net through the OpenMM-Torch[20] plugin. This integration has been a collaborative effort, with OpenMM-Torch being co-developed by the core teams of both OpenMM and TorchMD-Net. This ensures streamlined and effective utilization of TorchMD-Net models within OpenMM's simulation framework.

In the following sections we provide an overview of the TorchMD-Net framework. The manuscript is ordered as follows. In the Methods section, we initially provide a schematic overview of the principal model components, to then go over the currently available NNP architectures. We continue in subsection Training with details about the different parts involved in the training and deployment of these architectures and how they are exposed in TorchMD-Net. Then, in the Optimization subsection, we lay out the optimization strategies employed in this release. Finally, we present a series of validation and performance results in the Results section.

TorchMD-Net is freely available with a permissive license (MIT) at https://github.com/torchmd/torchmd-net.

2 Background

We interpret a neural network potential as a machine learning model that takes as input a series of atomic positions, denoted by $\boldsymbol{R}$, embedding indices such as atomic numbers, Z, and optionally charges (which might be per-sample or per-atom),[21] q, and outputs a per-sample scalar value and optionally its negative gradient with respect to the positions, typically interpreted as the potential energy and atomic forces, respectively. Note, however, that TorchMD-Net is not limited to this interpretation of the outputs, which are generally labeled as `y` and `neg_dy` respectively.

Figure 1 provides a comprehensive overview of the TorchMD-Net architecture. The diagram's left section illustrates the various components of the primary module, designated as `TorchMD_Net`, which constitutes, conceptually and in the API itself, an NNP model. Each component within this object is modular and customizable, allowing for the creation of diverse models. At the heart of the NNP is the `representation_model`. This part of
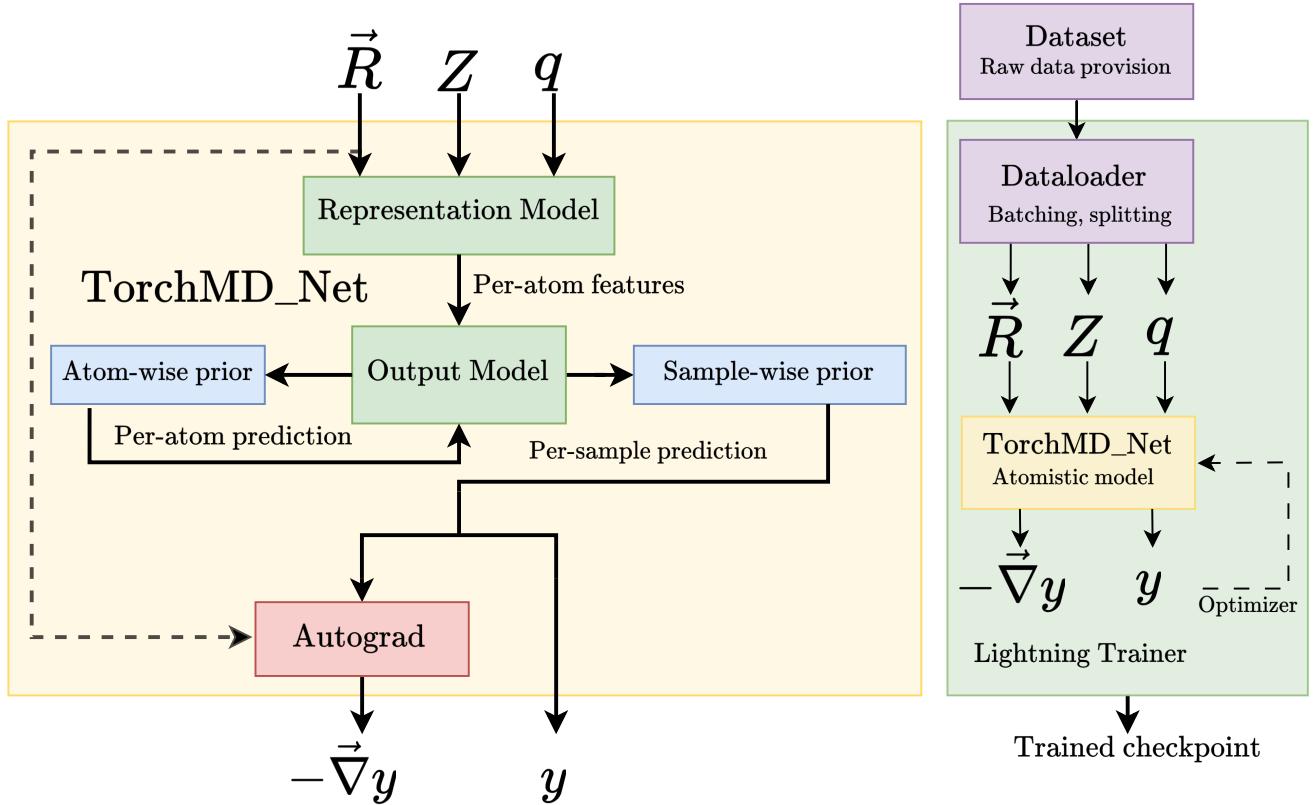
Figure 1: The main module in TorchMD-Net is called `TorchMD_Net` from the `torchmdnet.models.model` module. This class combines a given representation model (such as the Equivariant Transformer), an output model (such as the scalar output module) and a prior model (such as the Atomref prior), producing a module that takes as input a series of atoms features and outputs a scalar value (i.e. energy per molecule) and when `derivative = True`, its negative gradient with respect to the input positions (i.e. atomic forces).

the architecture takes the set of inputs stated above and outputs a series of per-atom features. These features are subsequently fed into an `output_model`. The purpose of this model is to further process these features into single atomic values, which typically will be aggregated and will represent the total potential energy, though it can represent other per-sample or per-atom quantities as well, depending on the specifics of its design and (optional) aggregation scheme. Output models normally include learnable parameters (e.g. a multilayer perceptron). Prior physical models can be employed to augment either the atom-level or aggregated per-molecule predictions with further physical insights. Furthermore, the framework integrates PyTorch's Autograd for automatic differentiation, enabling the computation of the negative gradient of the per-molecule scalar prediction with respect to atomic positions. This is particularly relevant when interpreting the per-molecule value as the potential energy, as it yields the atomic forces in a way that ensures, by construction, that the resulting force field is energy-conserving.

This modular logic allows for flexibility in the combination of representation models and output models. Therefore, by building a custom output module, researchers can make use of the representation models for other prediction tasks beyond potential energy and forces.

2.1 Available representation models

Although the framework is not restricted to it, current models in TorchMD-Net are message-passing neuralnetworks[22,23] (MPNNs) which learn approximations to the many-body potential energy function. Atoms are identified with graph nodes embedded in 3D space, building edges between them after the definition of some cutoff radius. The neural network uses atomic and geometric information to learn expressive representations by propagating, aggregating, and transforming features from neighboring nodes found within the cutoff radius.[24,25] In most current NNPs, after several message-passing steps, node features are used to predict

per-atom scalar quantities which are identified with atomic contributions to the energy of the molecule.

2.1.1 TensorNet

TensorNet[13] is an $O(3)$-equivariant model based on rank-2 Cartesian tensor representations. Euclidean neural network potentials[26–29] have been shown to achieve state-of-the-art performance and better data efficiency than previous models, relying on higher-rank equivariant features which are irreducible representations of the rotation group, in the form of spherical tensors. However, the computation of tensor products in these models can be computationally demanding. In contrast, TensorNet exploits the use of Cartesian rank-2 tensors (3x3 matrices) which can be very efficiently decomposed into scalar, vector and rank-2 tensor features. Furthermore, Clebsch-Gordan tensor products are substituted by straightforward and node-level 3x3 matrix products.

TensorNet achieved state-of-the-art accuracy on common benchmark datasets with a small number of message-passing layers, learnable parameters, and computational cost. The prediction of up to rank-2 molecular properties that behave appropriately under geometric transformations such as reflections and rotations is also possible.

2.1.2 Equivariant Transformer

The Equivariant Transformer[11] (ET) is an equivariant neural network that uses both scalar and Cartesian vector representations. The distinctive feature of the ET in comparison to other Cartesian vector models such as PaiNN[30] or EGNN[31] is the use of a distance-dependent dot product attention mechanism, which achieved comparable performance to state-of-the-art models on benchmark datasets at the time of publication.Furthermore, the analysis of attention weights allowed us to extract insights into the interaction of different atomic species for the prediction of molecular energies and forces. The model also exhibits a low computational cost for inference and train-

ing in comparison to some of the most used NNPs in the literature.[32]

As part of the current release, we removed a discontinuity at the cutoff radius. In the original description, vector features' residual updates, as opposed to scalar features' updates, received contributions from the value pathway of the attention mechanism which were not properly weighted by the cosine cutoff function envelope, which is reflected in Eq. 9 in the original paper.[11] We fixed it by applying $\phi(d_{ij})$, i.e., $\text{split}(V_j \odot D_{ij}^V) \to \text{split}(\phi(d_{ij})V_j \odot D_{ij}^V)$. To ensure backward compatibility, this modification is only applied when setting the new ET argument `vector_cutoff = True`. The impact of this modification is evaluated in the results section.

2.1.3 Graph Network

The graph network is an invariant model inspired by both the SchNet[33] and PhysNet[34] architectures. Described in Ref 12, the network was optimized to have satisfactory performance on coarse-grained proteins, allowing the building of NNPs that correctly reproduce fast-folder protein free energy landscapes. In contrast to the ET and TensorNet, the graph network only uses relative distances between atoms as geometrical information, which are invariant to translations, rotations, and reflections. The distances are used by the model to learn a set of continuous filters that are applied to feature graph convolutions as in SchNet,[33] progressively updating the initial atomic embeddings by means of residual connections.

2.2 Physical priors for models

Priors are additional physical terms that can be introduced for the prediction of potential energies. Some of these terms have been used in NNPs in the literature,[12,35,36] sometimes even including learnable parameters. In TorchMD-Net, we provide some predefined priors, which can be optionally added to the neural network energy prediction as additional physics-based contributions:

- Atomref: These are per-element atomic reference energies, which are usually provided directly in the dataset. In this case, the neural network has to predict the remaining contribution to the potential energy, which can be regarded as the formation energy of the molecule. There is also the option of making this prior learnable, in which case it is initialized with atomic reference energies, but these contributions are modified during training.

- Coulomb: This prior corresponds to the usual Coulomb electrostatic interaction, scaled by a cosine switching function to reduce its effect at short distances. Using this prior requires providing per-atom partial charges.

- D2 dispersion: In this case, the prior corresponds to the D2 dispersive correction used in DFT-D2.[37] C_6 coefficients and Van der Waals radii for elements are already incorporated in the method.

- ZBL potential: This prior implements the Ziegler-Biersack-Littmark (ZBL) potential for screened nuclear repulsion as described in Ref 18. It is an empirical potential effectively describing the repulsion between atoms at very short distances, and only atomic numbers need to be provided.

Note that forces are computed directly by `autograd` when adding the energy contributions coming from the priors before the backward automatic differentiation step. Even though the previous terms are the currently predefined options in TorchMD-Net, all these priors are derived from a general `BasePrior` class, which easily allows researchers to implement their own priors, following the modular logic behind the framework.

2.3 Training

The right diagram in Figure 1 depicts the main training loop in TorchMD-Net. A Dataset provides sample/output pairs for the NNP and is divided into training, validation and testing sets

and batched by a Dataloader (as provided by the Pytorch Geometric library[38]). We make use of the PyTorch Lightning library's[15] trainer, which also allows multi-GPU training. Checkpoints are generated during training, containing the current weights of the model, which can then be subsequently loaded for inference or further training.

2.3.1 Datasets

Within TorchMD-Net, datasets can be accessed through the YAML configuration file for use with the `torchmd-train` utility or programmatically via the Python API. Predefined datasets include SPICE,[39] QM9,[40] WaterBox,[41] (r)MD17[42],[43] MD22,[44] ANI1,[45] ANI1x,[46] ANI1ccx,[46] ANI2x[47] and the COMP6[48] evaluation dataset with all its subsets (ANIMD, DrugBank, GDB07to09, GDB10to13, Tripeptides and S66X8), offering diverse training environments for molecular dynamics and quantum chemistry applications. These datasets serve as common benchmarks in the field of neural network potentials. However, on top of these, the framework allows the flexible incorporation of user-generated datasets for customized applications. The `Custom` dataset functionality allows users to train models with molecular data encapsulated in simple NumPy file formats without writing a single line of code. By specifying paths to coordinate and embedding index (e.g. atomic numbers) and reference energy and force files, researchers can easily integrate their datasets into the training process. This capability ensures TorchMD-Net's adaptability to a wider array of applications beyond its pre-packaged offerings. In addition, TorchMD-Net offers support for other popular dataset formats, such as HDF5. Special care is taken to ensure data is cached as much as possible, using techniques such as in memory datasets and memory mapped files.

2.3.2 Losses

During training, a weighted sum of mean squared error (MSE) losses of energy and forces is used, weighting each of them according to

user input. In validation, we provide both L1 and MSE losses separately for energies and forces, while for testing L1 losses alone are used. The framework allows to use of an exponential moving average (EMA) to update the losses during the training and validation stages to smooth out the progression of loss values.

2.4 Usage examples

In the following sections we showcase code for some typical use cases of TorchMD-Net. While these snippets are generally self-contained the reader is pointed to the online documentation[17] for further information.

2.4.1 Training code example

The project introduces a command line tool, `torchmd-train`, designed as a code-free method for model training. This tool is set up through a YAML file, with several examples available in the TorchMD-Net GitHub repository for reference. However, we also offer an illustrative script here that outlines the process of training an existing model using the Python API. The `LNNP` class, found within the `torchmdnet.module` module, encapsulates the procedures for both the creation and training of a model. This class is inherited from Pytorch Lightning `LightningModule`, offering all the extensive customization available in it. The following is a succinct yet comprehensive example of how to utilize `LNNP` for training purposes:

```python
from torchmdnet.data import DataModule
from torchmdnet.module import LNNP
from pytorch_lightning import Trainer
args = {
    'dataset': 'ANI1X',
    'model': 'tensornet',
    'num_epochs': 200,
    'embedding_dimension': 128,
    'num_layers': 2,
    'num_rbf': 32,
    'rbf_type': 'expnorm',
    'trainable_rbf': False,
    'activation': 'silu',
    'cutoff_lower': 0.0,
    'cutoff_upper': 5.0,
    'max_z': 100,
    'max_num_neighbors': 64,
    'derivative': True # So the model returns forces.
}
data = DataModule(args)
data.prepare_data()
data.setup("fit")
lnnp = LNNP(args,
prior_model=None,
mean=data.mean,
std=data.std)
trainer = Trainer(max_epochs=args['num_epochs'])
trainer.fit(lnnp, data)
model = LNNP.load_from_checkpoint(trainer.checkpoint_callback.
    best_model_path)
trainer = pl.Trainer(inference_mode=False)
```

```
31    trainer.test(model, data)
```

This example shows the minimal steps required to prepare data, initialize the `LNNP` class, train and test a model using PyTorch Lightning's `Trainer`. The `Trainer` here is simplified for brevity; in practice, additional callbacks and logger configurations could be added.

2.4.2 Loading a Trained Model for Inference

After training a model, the next logical step is to use it for inference. TorchMD-Net offers a dedicated function, `load_model`, to facilitate this. Below is a concise example:

```
1   from torchmdnet.models.model import load_model
2   # Define the path to the saved model checkpoint
3   checkpoint_path = "path/to/saved_model_checkpoint.ckpt"
4   # Load the model
5   loaded_model = load_model(checkpoint_path)
6   # Prepare the input data (atomic numbers, positions, batch
        index, etc.)
7   # For demonstration, these are placeholders and should be
        replaced with actual data
8   input_data = {
9       'z': torch.Tensor([...]),
10      'pos': torch.Tensor([...]),
11      'batch': torch.Tensor([...])
12      # ... other optional fields
13  }
14  # Perform inference
15  energy, forces = loaded_model(**input_data)
16  # Energy and forces are now available for further analysis or
        visualization
```

In this example, `checkpoint_path` should point to the location where the trained model checkpoint is saved. The `input_data` dictionary should be populated with the actual atomic numbers, positions, and other required or optional fields. Finally, `energy` and `forces` are obtained from the loaded model and can be used as needed.

2.4.3 Integration with OpenMM

It is possible to run TorchMD-Net neural network potentials as force fields in OpenMM[19] to run molecular dynamics. The OpenMM-Torch[20] package is leveraged for this. Integration consists of writing a wrapper class to accommodate the unit requirements of OpenMM and to provide the model with any information not proper to OpenMM (like the embedding indices). The following code showcases an example of how to add a TorchMD-Net NNP as an OpenMM Force.

```
1   from torchmdnet.models.model import load_model
2   from openmmtorch import TorchForce
3   from openmm import System
4
5   class Wrapper(torch.nn.Module):
6
7       def __init__(self, embeddings, model):
8           super(Wrapper, self).__init__()
9           self.embeddings = embeddings
10          # OpenMM will compute the forces
11          #  by backpropagating the energy,
12          # so we can load the model with derivative=False
13          self.model = load_model(model, derivative=False)
14
15      def forward(self, positions):
16          # OpenMM works with nanometer positions
17          #  and kilojoule per mole energies
18          # Depending on the model, you might need
19          #  to convert the units
20          positions = positions * 10.0 # nm -> A
21          energy = self.model(z=self.embeddings, pos=positions)
                [0]
22          return energy * 96.4916 # eV -> kJ/mol
23
24  # The embeddings used during training (e.g. atomic numbers)
25  #  for each atom in the simulation.
26  z = torch.tensor([1,1], torch.long)
27  model = torch.jit.script(Wrapper(z, "model.ckpt"))
28  # Create a TorchForce object from the model
29  torch_force = openmmtorch.TorchForce(model)
30  system = System()
31  # The TorchForce object can be used as a regular OpenMM Force
32  system.addForce(torch_force)
33  # Set up the rest of the OpenMM simulation
34  # ...
```

2.5 Optimization techniques

Typical neural network potential (NNP) algorithms implemented in PyTorch[14] comprise a series of sequential operations such as multi-layer perceptrons and message-passing operations.

As PyTorch operations translate into highly optimized CUDA kernel calls, and owning to the eager-first nature of PyTorch, the efficiency of modern GPUs often turns kernel launching overhead into a performance bottleneck. CUDA graphs address this by consolidating multiple kernel calls into a single graph, drastically reducing kernel launch overhead. However, CUDA graphs impose stringent limitations. These include the need for static shapes in graphed code sections, which can lead to costly recompilations or memory inefficiencies, and the exclusion of operations requiring CPU-GPU synchronization.

Conversely, developments in the compiler community[49], including technologies like OpenAI's Triton[50] and subsequently PyTorch enhancements, are gradually diminishing the reliance on CUDA graphs by automatically changing the structure of the code in ever more profound ways (i.e kernel fusion[51–53]). These advancements, such as TorchDynamo introduced in PyTorch 2.0 through `torch.compile`, optimize code structure through Just-In-Time

(JIT) compilation.

Even with JIT, and in general transpilation-based techniques, CUDA graphs often provide the best out-of-the-box performance improvements and at the bare minimum, facilitate the optimizations introduced by the former. Encapsulating a piece of code within a CUDA graph, a process known as 'stream capture', necessitates adherence to several specific requirements. This often demands substantial modifications to the code. Crucially, for code to be eligible for capture, it must avoid any CPU-GPU synchronization activities, including synchronization barriers and memory copies. Additionally, all arrays involved in the operations must possess static shapes and fixed memory addresses, precluding any dynamic memory allocations during the process.

The CUDA graph interface in PyTorch alleviates many challenges associated with adapting code for stream capture. It particularly excels in managing memory allocations within captured environments automatically and transparently. However, challenges arise in specific implementations, as exemplified by TensorNet. The main issue in TensorNet is its neighbor list, which inherently varies in shape at each inference step due to the fluctuating number of neighbors. This variation affects the early stages of the architecture, resulting in TensorNet primarily operating on dynamically shaped tensors. To address this, we implemented a static shape mode that creates placeholder neighbor pairs up to a predetermined maximum. We then ensure the architecture disregards these placeholders' contributions. Although this method increases the initial workload, our empirical data indicates that the performance gains from capturing the entire network substantially outweigh this added overhead. Work on static shaped graph networks has been done before in JAX libraries such as jgraph.[54]

Regardless of the choice of framework, the aforementioned general optimization techniques (and in particular complying with CUDA graph requirements) constitute a good roadmap in the quest for efficiency. Many of these frameworks are down the line aiming to somehow transform user code into a series of CUDA (or some other massively parallel language with similar properties such as Triton or OpenCL) kernels, with a preferably small number of them. General strategies like eliminating CPU-GPU synchronization barriers or communication help in this regard. The generality of these directives makes the resulting implementations more aligned with `torch.compile` in our case, but would for instance also aid `jax.jit` if a developer were to port our neighbor list to JAX.

By making our operators strive for CUDA-graph compatibility, making them `torch.compile`-aware and easy to import, we intend to boost the development of future architectures and provide efficient tools for the community even outside the TorchMD-Net ecosystem.

In the following sections, we explore the impact of these optimizations on both inference and training performance.

2.5.1 Neighbor search and periodic boundary conditions

Message-passing neural networks, such as the architectures currently supported in the framework, require a list of connections among nodes referred to as edges. This list is constructed by proximity after the definition of a cutoff radius (a neighbor list). TorchMD-Net offers a neighbor list construction engine specifically tailored for NNPs, exposing a naive brute-force $O(N^2)$ algorithm that works best for small workloads and a cell list (a standard $O(N)$ hash-and-sort strategy widely used in MD[55,56]) that performs better for large systems (see Figures 3 and 2). Effectively, this engine makes neighbor search a negligible part of the overall computation. This operation is exposed as a PyTorch Autograd extension with a reference pure PyTorch CPU implementation. The forward pass has a registration for inputs in a CUDA device written in C++/CUDA. Finally, a custom implementation of the backward pass is also included and written in PyTorch to easily accommodate for higher order derivatives[1]. From a user perspec-

---

[1]The second backward pass of this operation is needed when training using forces in TorchMD-Net, as

tive this translates into the operation being usable in an Autograd environment from any device compatible with PyTorch.

Special measures are taken into account to ensure that the neighbor search is compatible with CUDA-graphs. For this matter, it is required that the neighbor search works on a statically-shaped set of input/outputs, which poses a problem given that the number of neighbor pairs in the system is not known in advance and is bound to change from input to input. We solve this by requiring an upper bound for the number of pairs in the system and padding the outputs with a special value (-1) for unused pairs. Furthermore, TorchMD-Net architectures support rectangular and triclinic periodic boundary conditions.

Contrary to usual MD workloads, it is common to have batches of input samples in NNPs. This owns to the very nature of neural network training but also can benefit inference (for instance, allowing the possibility of running many simulations in parallel, like TorchMD[57] does). Our neighbor list is able to handle arbitrary batch sizes while maintaining compatibility with CUDA graphs. There are many implementations of various neighbor list algorithms in external packages but, regardless of their performance, none of them fully satisfy our set of constraints. These being that it should be available in pytorch (discarding, for instance, any JAX-centered library such as JAX-MD[58]), support CUDA-graphs (discarding the `radius_graph` function in Pytorch-Geometric[38]), be able to handle batches (discarding ASE[59] and PyMat-Gen[60]) and finally support rectangular and triclinic periodic boundary conditions (which some of the aforementioned implementations lack).

The current cell list implementation constructs a single cell list including atoms for all batches, excluding pairs of particles that pertain to different batches when finding neighbors in it. This makes it so that each particle has to perform a check against every other particle in the vicinity for all batches, which degrades per-

the forces are computed via backpropagation of the energy.

formance with increasing batch size. We find this to be an acceptable compromise given that doing it this way facilitates compatibility with CUDA graphs and we assume that with an increasing number of particles (where the cell list excels) the typical batch size will decrease. Still, the particularities of the cell list implementation make its performance especially susceptible to the batch size, as evidenced by the variability observed in the cell list curves in figures 3 and 2.

Our approach to handling batches consists of assuming there is only one batch, delaying this check-up until just before fetching the atom positions to check their distances. While the scalability of this approach with the number of batches is questionable (many unnecessary pairs might be checked) it allows us to overcome a series of limitations besides the aforementioned for the cell list. The operation takes as principal inputs a contiguous tensor of atomic positions and another with batch indices (indicating to which batch an atom pertains). Splitting these tensors as a precomputation would in general require knowing the number of distinct batches (involving a costly reduction operation and synchronization barriers if the value is required CPU-side) as well as computing where each one starts and ends in the positions tensor (assuming the input is sorted by batch). Even if the batch locations were known in advance, calling the neighbor operation for each batch independently could result in the launching of several instances of small CUDA kernels, which would not fill the GPU and hurt performance overall. On the other hand, an approach with a control flow depending on the contents of the input tensors would hinder the static requirements of CUDA graphs.

We offer two different brute-force neighbor search strategies, which are selected depending on the number of atoms received by the operation. In both of them, the neighbor list is generated in a single CUDA kernel in which every possible pair in the system is checked, first to ensure both are in the same batch and then for their distance. The range of applicability of these algorithms (intended for workloads of less than 10 thousand atoms) allows for most, if not

all, of the inputs to be read fit in the deepest cache levels of the GPU[2], leaving these kernels typically bottlenecked by the actual writing of the neighbor pairs, which is carried out using an atomic counter to "reserve" a space in the neighbor list and then writing the pair to this list. The first brute-force strategy launches a thread per possible pair, this limits its function to less than $2^{16} - 1$ atoms given that the number of possible pairs in this case, $\frac{N(N-1)}{2}$, coincides with the maximum number of threads that can be launched in a single CUDA kernel, $2^{31} - 1$. The other strategy is the Fast N-Body algorithm described in chapter 31 of GPU-Gems 3,[55] which launches a thread per atom and leverages CUDA's shared memory. Remarkably, we find the thread-per-pair strategy to be the fastest of the two up until its theoretical limit of $2^{16} - 1$ atoms. Figure 2 shows how the brute algorithm (in this case the thread-per-pair implementation) scales well with the number of batches, as the loading of the positions can be mostly avoided and the batch tensor fits in the cache in its entirety.

All data presented in this section was gathered in an RTX4090 NVIDIA GPU using CUDA 12. Each point is obtained by averaging 50 identical executions. Warmup executions are also performed before measuring.

2.5.2 Training

Optimizing neural network training presents distinct challenges compared to inference optimization. Primarily, the variable length of each training sample, exacerbated by batching processes (where a varying number of samples constitutes a single network input), impedes optimizations dependent on static shapes (i.e. CUDA graphs). A potential solution involves integrating 'ghost molecules', akin to strategies used in static neighbor list shaping, to standardize the atom count inputted to the network. However, this method increases memory
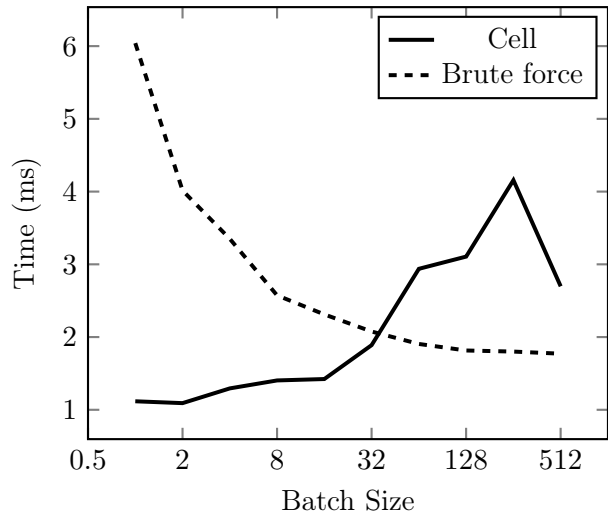

Figure 2: Performance comparison of cell (solid line) and brute-force (dashed line) neighbor search strategies across different batch sizes for a random cloud of 32k particles with 64 neighbors per particle on average. The particles are split into a certain number of batches.
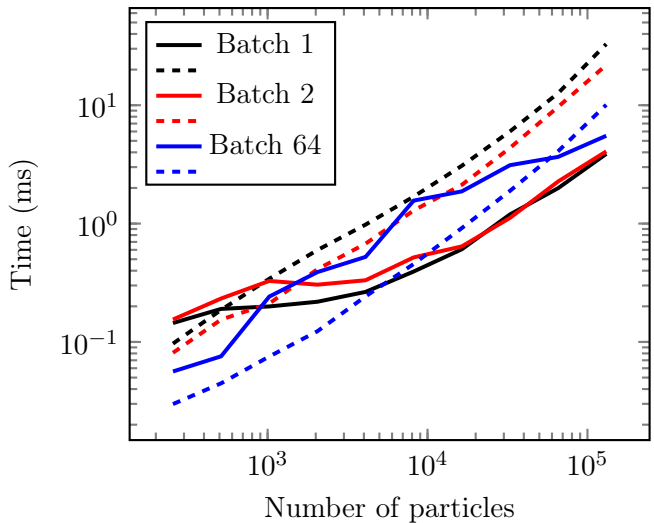

Figure 3: Performance comparison of cell (solid line) and brute-force (dashed line) neighbor search strategies across different batch sizes for a random cloud of particles with 64 neighbors per particle on average. Cell list performance tends to degrade with increasing batch size, while the opposite is true for brute force.

---

[2]For reference, the RTX4090 has 128KB of L1 cache (the fastest access global memory cache) per streaming multiprocessor, which is theoretically enough to hold the data for the positions of 10^4 atoms stored in single precision $(10^4(\text{atoms})3(\text{coordinates})4(\text{bytes/value}) \approx 117KB)$

consumption in an already memory-constrained environment and can potentially be wasteful when the dataset presents samples highly heterogeneous in size.

Moreover, training necessitates backpropagation through the network. In our context, this involves a double backpropagation process when the loss function includes force calculations. Currently, double backpropagation is inadequately supported by the PyTorch compiler. A workaround is to manually implement the network's initial backward pass (specifically, the force computation). This adjustment enables Autograd to perform only a single backward pass during training, leveraging the PyTorch compiler's capabilities. Nevertheless, challenges persist with the PyTorch compiler when managing dynamic input shapes. We hope future versions of PyTorch will improve support for both dynamic shapes and double backpropagation, allowing to make use of our inference optimizations in training seamlessly.

Given the current constraints, the current release does not include any training-specific optimizations besides the improved dataloader support as previously described.

3 Results

3.1 Validation

In this subsection, we evaluate the impact of the architectural modifications introduced in the models on predictive accuracy. In the case of TensorNet the modifications targeted its computational performance alone, while for the ET one needs to consider the changes induced by `vector_cutoff = True`.

3.1.1 Accuracy with TensorNet

Original test MAE presented in Ref. 13 for the QM9 U_0 target quantity is 3.9(1) meV, while the latest optimized versions of the model (see Figure 4) yield 3.8(2) meV, confirming that the architectural optimizations do not affect TensorNet's prediction performance. The training loss was computed in this case as the MSE between

predicted and true energies. This state-of-the-art performance is achieved with the largest model with 4.0 million trainable parameters, with specific architectural and training hyperparameters being found in Table 5. We also provide in Table 1 the accuracy of smaller and shallower models on the same QM9 quantity (that is, using the same hyperparameters as in Table 5, except for `embedding_dimension = 128` and `num_layers = 0, 1, 2`), while comparing them to other NNPs. Overall, TensorNet demonstrates very satisfactory performances, achieving close to state-of-the-art accuracy (< 5 meV MAE) with a very reduced number of parameters.

Table 1: Mean absolute error in meV for different models trained on QM9 target property U_0. TensorNet 3L* uses an embedding dimension of 256, while in other cases 128. For the ET, subscripts new and old correspond to the new and the original implementation, that is, with `vector_cutoff = True` and `False`, respectively.

Model	U_0 MAE (meV)
Cormorant[61]	22
SEGNN[62]	15
SchNet[63]	14
EGNN[31]	11
Equiformer[64]	6.6
DimeNet++[65]	6.3
SphereNet[66]	6.3
ET_{old}[11]	6.2
PaiNN[30]	5.9
Allegro[28]	4.7
MACE[29]	4.1
ET_{new}	5.7
TensorNet 0L	7.2
TensorNet 1L	4.7
TensorNet 2L	4.4
TensorNet 3L*	3.9

3.1.2 Accuracy with the Equivariant Transformer

As previously mentioned, we provide an implementation of the ET where it is modified by applying the cutoff function to the values'
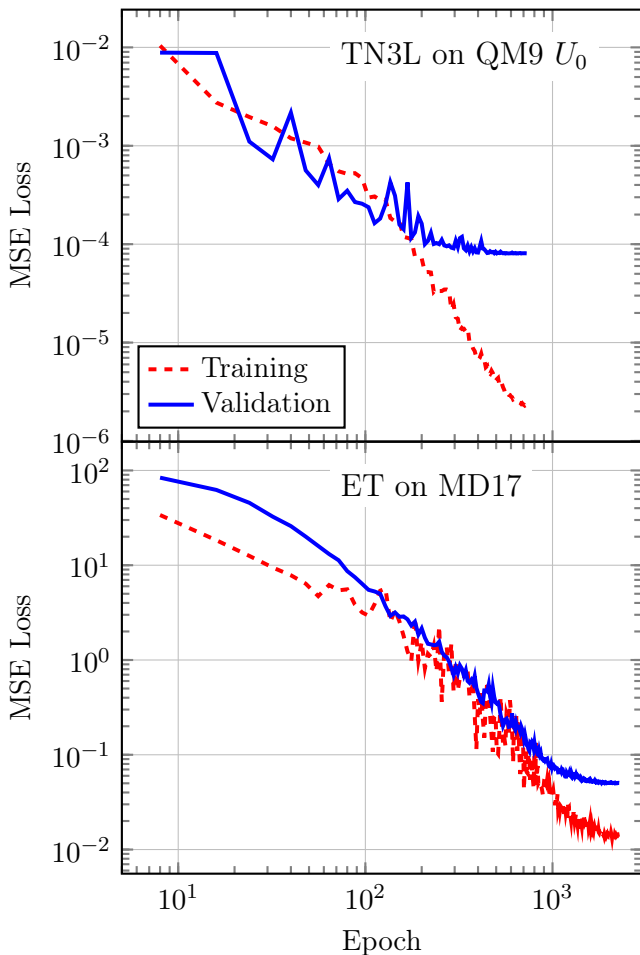
Figure 4: Training and validation curves for two different training benchmark datasets. Up: TensorNet with 3 interaction layers on the QM9 U_0, parameters in table 5. Down: Equivariant Transformer on MD17, parameters in table 3

.

pathway of the attention mechanism to enforce a continuous energy landscape at the cutoff distance. Therefore, we checked to which extent these changes, together with TorchMD-Net's ones, affect the accuracy of the Equivariant Transformer. We trained the model on the MD17 aspirin dataset (Figure 4) using the hyperparameters defined for the original version of the ET (Table 3, with the addition of `vector_cutoff = True`), giving final test MAEs of 0.139 kcal/mol and 0.232 kcal/mol/Åin energies and forces, respectively, compared to the original implementation which gave 0.123 kcal/mol and 0.253 kcal/mol/Å.[11] Regarding QM9 U_0, we reused the original hyperparameters for the dataset found in Table 4 (again, adding `vector_cutoff = True`), and comparative results can be found in Table 1.

3.2 Molecular Simulations

We performed NVT molecular dynamics simulations, in vacuum, employing TensorNet models trained on the ANI-2x dataset.[47] A table detailing the hyperparameters is provided for reference in Table 6. Note that we did not include any physical priors in these trainings nor in the subsequent simulations, i.e. all forces in the system come from the model itself. Starting from the SPICE dataset,[39] we selected the PubChem subset and utilized it to create a test set comprising four randomly chosen conformers. Following the order as depicted in Figure 5, the number of atoms for each molecule is 44, 47, 41, and 46. This test set aimed to evaluate the ability of the NNP to perform stable molecular dynamics (MD) simulations on molecules not encountered during the training stage.[67,68] The training dataset, as well as the PubChem subset, represent a broad diversification of molecules containing the elements H, C, N, O, S, F, Cl. To generate the input data, the SMILES and the coordinates of interest were used to build a molecule object using openff-toolkit,[69] and the atomic numbers were used as embeddings. Using the more accurate TensorNet 2L model, a 200 ns trajectory,

i.e. $2 \cdot 10^8$ steps [3], with a time-step of 1 fs was generated for each molecule using OpenMM's [19] `LangevinMiddleIntegrator` at 298.5K and a friction coefficient of 1 ps^{-1}. We also used for one of the molecules a TensorNet 0L model with the same simulation settings to test its stability. A root mean square displacement (RMSD) analysis was performed for each trajectory taking the starting conformation as a reference, see Figure 5. The results highlight the model's ability to run stable MD simulations, even for the 0L case where the model's receptive field and parameter count are substantially reduced. In terms of computational efficiency, for the specific cases in this section, TensorNet 0L achieves a simulation speed of about 70 ns/day, while the 2-layer version runs at approximately 20 ns/day. For comparison, considering the same number of particles, i.e. around 40 atoms, GFN2-xTB [70] runs at 0.05 ns/day, while PySCF, using B3LYP/6-31g level of theory,[71] runs at $3 \cdot 10^{-4}$ ns/day.

3.3 Speed performance

All results presented in this work were carried out using an NVIDIA RTX4090 GPU (driver version 525.147) with a dual 8 core Intel(R) Xeon(R) Silver 4110 CPU in Ubuntu 20.04. We used CUDA 12.0 with Pytorch version 2.1.0 from conda-forge. We provide all timings in million steps per day, which can be easily converted to nanoseconds per day. These units are more commonly used in molecular dynamics settings, and the conversion can be done by taking the quantity in million steps per day times the timestep in femtoseconds. Therefore, for example, 1 million steps per day is equivalent to 1 ns/day for a timestep of 1 fs.

To study the optimization strategies laid out in section 2.5 we show energy and forces inference performance for several equivalent implementations of TensorNet in Figure 6. Note that in TorchMD-Net, running inference requires one backpropagation step to compute forces as the negative gradient of the energies

with respect to the input positions, which are computed via Autograd. This step is also included in these benchmarks.

---

[3]Note a simulation step contains one forward pass of our model to compute total energy in addition to a backward pass to compute atomic forces.
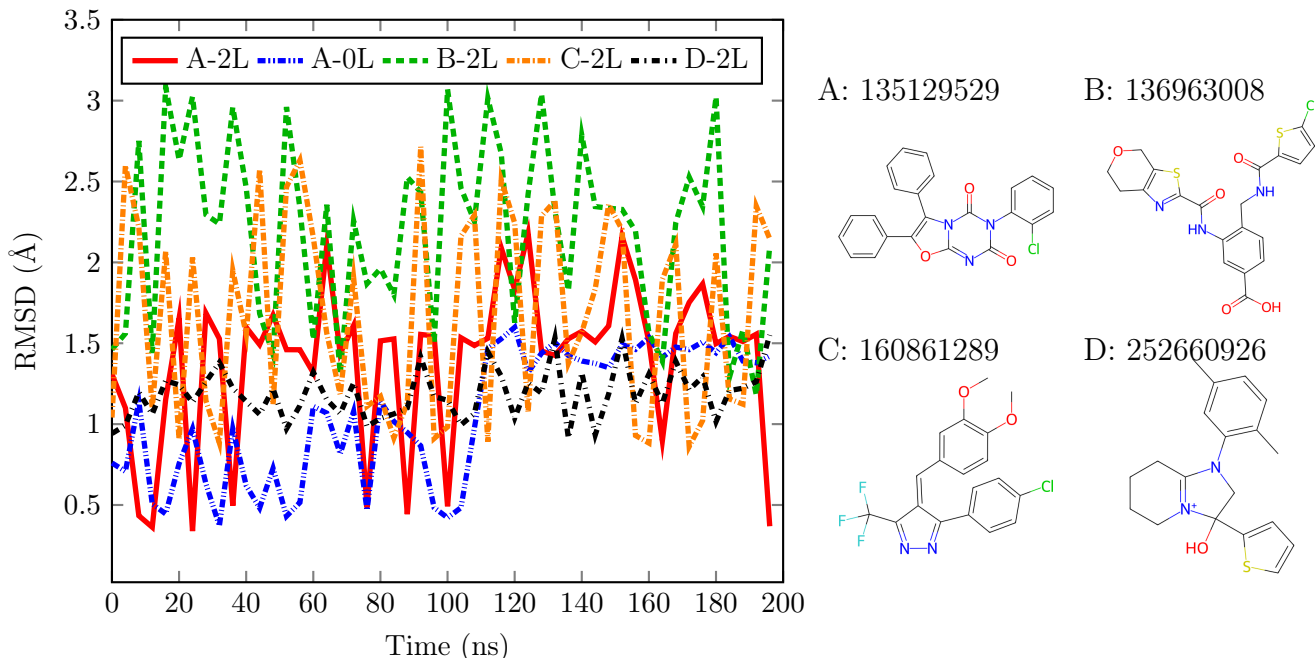
Figure 5: (Left) RMSD analysis for the trajectories of 4 molecules outside of the training set. Simulations are carried out with TensorNet 2L, using the parameters in table 6, with the exception of A-0L, in which a 0L TensorNet model is showcased. Presented data is plotted only every 4 ns for visualization clarity. (Right) Representation of the simulated molecules. Labels show the PubChem ID for each molecule.

Table 2: TensorNet inference times in million steps per day for the "Plain" (**P**), "Compile" (**C**) and "Graph" (**G**) implementations and varying number of message passing layers L

Molecule (atoms)	P 0L	P 1L	P 2L	C 0L	C 1L	C 2L	G 0L	G 1L	G 2L
Alanine dipeptide (22)	19.86	10.29	8.50	40.19	28.70	21.23	172.80	84.71	56.47
Testosterone (49)	15.05	11.93	8.56	38.57	27.00	21.49	154.29	63.53	39.82
Chignolin (166)	19.77	11.88	7.90	36.77	24.90	21.39	77.14	26.02	15.57
DHFR (2489)	5.56	1.67	0.98	14.47	3.27	1.83	5.65	1.69	1.00
Factor IX (5807)	2.32	0.69	0.41	5.42	1.35	0.77	2.33	0.70	0.42
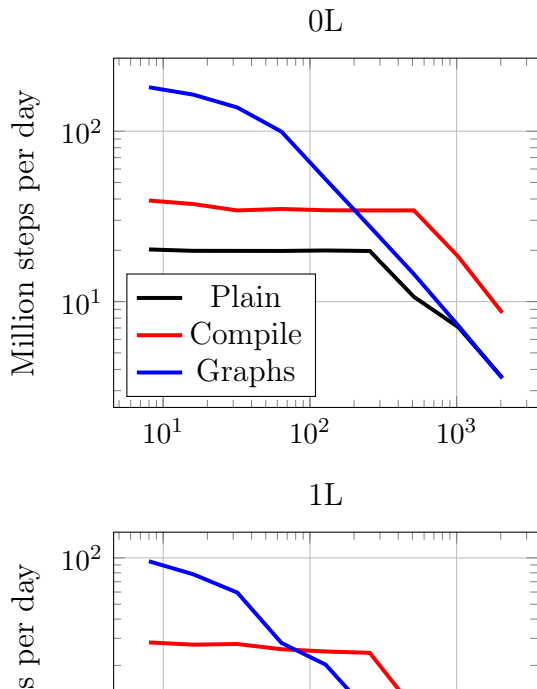
We make sure not to include any warmup times in these benchmarks by running the models for 100 iterations before timing. We refer as "Graph" to an implementation that has been modified to ensure every CUDA graph requirement is met. For "Compile" the implementation is carefully tailored to look for the best performance in `torch.compile` in addition to the changes introduced for "Graph". Finally, "Plain" represents the baseline implementation in PyTorch.

Although in principle the code received by the compiler is entirely capturable by a graph, it often decides to capture only some sections of

14

it, introducing other kinds of optimizations instead. This is also made evident by the appearance of the same kind of "plateau" performance for smaller workloads in both Plain and Compile, which can be attributed to a bottleneck produced by kernel launch overhead. Still, the torch compiler is able to provide a speedup of a factor of 2 to 3 for all workloads with respect to the original implementation.

CUDA kernel overhead (and thus the performance gain of CUDA graphs) is expected to dominate for small workloads, where it is usual for the kernel launching time to be larger than the actual execution. Figure 6 indeed corroborates this by showing speedups between 10 and 2 times for molecules with up to a few hundred atoms and for all numbers of interaction layers (0, 1, and 2). Starting from workloads consisting of several hundreds of atoms, the performance of the Plain version is recovered.

We also explore inference times for some molecules with varying numbers of atoms in Table 2. For these molecules, which can be found in the repository for speed benchmarking purposes, we measure time to compute the potential energy and atomic forces of a single example using TensorNet with 0, 1 and 2 interaction layers. Again, we express this time in million steps per day. In all cases, we use a cutoff of 4.5Å, an embedding dimension of 128, 32 radial basis functions and a maximum of 32 neighbors per particle.

Another factor to take into account is the additional memory consumption introduced by enforcing static shapes. There are two sources of memory overhead in this case; the padding of the edges until a maximum number of pairs and the extra node (ghost atom) that these edges are assigned to. The extra node bears a cost scaling with $1/N$ and is thus negligible. On the other hand, the extra edges have a cost in memory that scales with the difference between the maximum number of pairs and the actual pairs found. In practice these values are chosen to be as close as possible while leaving a safety buffer, making the expected memory overhead a small percentage of the total memory except in some

pathological cases[4].

4 Conclusions

TorchMD-Net has significantly evolved in its recent iterations, becoming a comprehensive platform for neural network potentials (NNPs). It provides researchers with robust tools for both rapid prototyping of new models and executing production-level tasks. However, despite these advancements, NNPs still face substantial challenges before they can fully replace traditional force fields in molecular dynamics simulations. Currently, while the necessary software infrastructure is largely in place, as evidenced by the first-class support for NNPs in popular packages,[19] issues such as memory requirements and computational performance remain significant concerns.

The impact of memory limitations is anticipated to diminish with ongoing hardware advancements. Yet, enhancing computational performance to a level that is competitive with traditional methods necessitates more intricate strategies. This involves developing architectures and their implementations in a manner that leverages the full capabilities of GPU hardware.

From a software development perspective, the compilation functionality within PyTorch is an evolving feature, still in its early stages. Its current development trajectory, which aims to minimize the necessary code modifications for effective utilization, suggests that future PyTorch releases will likely bring performance enhancements. Continuous improvements in the relevant toolset, encompassing PyTorch, CUDA, Triton, and others, are gradually narrowing the performance gap between highly optimized code and more straightforward implementations.

---

[4]For instance, for simplicity we fix the maximum number of pairs as 32 times the number of atoms for the benchmarks in table 2 even for Alanine dipeptide, which only has 22 atoms.

Acknowledgement

References

(1) Behler, J.; Parrinello, M. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett.* **2007**, *98*, 146401.

(2) Kocer, E.; Ko, T. W.; Behler, J. Neural Network Potentials: A Concise Overview of Methods. arXiv, 2021; 10.48550/arXiv.2107.03727.

(3) Behler, J. Perspective: Machine learning potentials for atomistic simulations. *The Journal of Chemical Physics* **2016**, *145*, 170901.

(4) Schütt, K. T.; Sauceda, H. E.; Kindermans, P.-J.; Tkatchenko, A.; Müller, K.-R. SchNet – A deep learning architecture for molecules and materials. *The Journal of Chemical Physics* **2018**, *148*, 241722.

(5) Deringer, V. L.; Caro, M. A.; Csányi, G. Machine Learning Interatomic Potentials as Emerging Tools for Materials Science. *Advanced Materials* **2019**, *31*, 1902765.

(6) Botu, V.; Batra, R.; Chapman, J.; Ramprasad, R. Machine Learning Force Fields: Construction, Validation, and Outlook. *The Journal of Physical Chemistry C* **2017**, *121*, 511–522.

(7) Ko, T. W.; Finkler, J. A.; Goedecker, S.; Behler, J. A fourth-generation high-dimensional neural network potential with accurate electrostatics including non-local charge transfer. *Nature Communications* **2021**, *12*, 398.

(8) Schütt, K. T.; Hessmann, S. S. P.; Gebauer, N. W. A.; Lederer, J.; Gastegger, M. SchNetPack 2.0: A neural network toolbox for atomistic machine learning. *The Journal of Chemical Physics* **2023**, *158*.

(9) Gao, X.; Ramezanghorbani, F.; Isayev, O.; Smith, J. S.; Roitberg, A. E. TorchANI: A Free and Open Source PyTorch-Based Deep Learning Implementation of the ANI Neural Network Potentials. *Journal of Chemical Information and Modeling* **2020**, *60*, 3408–3415.

(10) Zeng, J.; Zhang, D.; Lu, D.; Mo, P.; Li, Z.; Chen, Y.; Rynik, M.; Huang, L.; Li, Z.; Shi, S. et al. DeePMD-kit v2: A software package for deep potential models. *The Journal of Chemical Physics* **2023**, *159*.

(11) Thölke, P.; De Fabritiis, G. TorchMD-NET: Equivariant Transformers for Neural Network based Molecular Potentials. arXiv, 2022; 10.48550/ARXIV.2202.02541.

(12) Majewski, M.; Pérez, A.; Thölke, P.; Doerr, S.; Charron, N. E.; Giorgino, T.; Husic, B. E.; Clementi, C.; Noé, F.; Fabritiis, G. D. Machine Learning Coarse-Grained Potentials of Protein Thermodynamics. arXiv, 2022; 10.48550/arXiv.2212.07492.

(13) Simeon, G.; De Fabritiis, G. TensorNet: Cartesian Tensor Representations for Efficient Learning of Molecular Potentials.

Advances in Neural Information Processing Systems. 2023; pp 37334–37353.

(14) Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L. et al. *Advances in Neural Information Processing Systems 32*; Curran Associates, Inc., 2019; pp 8024–8035.

(15) Pytorch Lightning. `lightning.ai/pytorch-lightning`, (Accessed February 15, 2024).

(16) conda-forge community The conda-forge Project: Community-based Software Distribution Built on the conda Package Format and Ecosystem. 2015; `https://doi.org/10.5281/zenodo.4774216`.

(17) TorchMD-NET Documentation. torchmd-net.readthedocs.io, (Accessed February 13, 2024).

(18) Biersack, J. P.; Ziegler, J. F. *Ion Implantation Techniques*; Springer Berlin Heidelberg, 1982; p 122–156.

(19) Eastman, P.; Galvelis, R.; Peláez, R. P.; Abreu, C. R. A.; Farr, S. E.; Gallicchio, E.; Gorenko, A.; Henry, M. M.; Hu, F.; Huang, J. et al. OpenMM 8: Molecular Dynamics Simulation with Machine Learning Potentials. *The Journal of Physical Chemistry B* **2024**, *128*, 109–116, PMID: 38154096.

(20) OpenMM-Torch. `https://github.com/openmm/openmm-torch`, (Accessed: February 15 2024).

(21) Simeon, G.; Mirarchi, A.; Pelaez, R. P.; Galvelis, R.; De Fabritiis, G. On the Inclusion of Charge and Spin States in Cartesian Tensor Neural Network Potentials. arXiv, 2024; 10.48550/ARXIV.2403.15073.

(22) Bronstein, M. M.; Bruna, J.; Cohen, T.; Veličković, P. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. arXiv, 2021; 10.48550/ARXIV.2104.13478.

(23) Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; Dahl, G. E. Neural Message Passing for Quantum Chemistry. arXiv, 2017; 10.48550/ARXIV.1704.01212.

(24) Joshi, C. K.; Bodnar, C.; Mathis, S. V.; Cohen, T.; Liò, P. On the Expressive Power of Geometric Graph Neural Networks. arXiv, 2023; `https://arxiv.org/abs/2301.09308`, 10.48550/ARXIV.2301.09308.

(25) Duval, A.; Mathis, S. V.; Joshi, C. K.; Schmidt, V.; Miret, S.; Malliaros, F. D.; Cohen, T.; Lio, P.; Bengio, Y.; Bronstein, M. A Hitchhiker's Guide to Geometric GNNs for 3D Atomic Systems. arXiv, 2023; 10.48550/ARXIV.2312.07511.

(26) Geiger, M.; Smidt, T. e3nn: Euclidean Neural Networks. arXiv, 2022; 10.48550/ARXIV.2207.09453.

(27) Batzner, S.; Musaelian, A.; Sun, L.; Geiger, M.; Mailoa, J. P.; Kornbluth, M.; Molinari, N.; Smidt, T. E.; Kozinsky, B. E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials. *Nature Communications* **2022**, *13*.

(28) Musaelian, A.; Batzner, S.; Johansson, A.; Sun, L.; Owen, C. J.; Kornbluth, M.; Kozinsky, B. Learning local equivariant representations for large-scale atomistic dynamics. *Nature Communications* **2023**, *14*.

(29) Batatia, I.; Kovacs, D. P.; Simm, G. N. C.; Ortner, C.; Csanyi, G. MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields. Advances in Neural Information Processing Systems. 2022.

(30) Schütt, K. T.; Unke, O. T.; Gastegger, M. Equivariant message passing for the prediction of tensorial properties and molecular spectra. arXiv, 2021; `https://arxiv.org/abs/2102.03150`, 10.48550/ARXIV.2102.03150.

(31) Satorras, V. G.; Hoogeboom, E.; Welling, M. E(n) Equivariant Graph Neural Networks. arXiv, 2021; 10.48550/ARXIV.2102.09844.

(32) Bihani, V.; Pratiush, U.; Mannan, S.; Du, T.; Chen, Z.; Miret, S.; Micoulaut, M.; Smedskjaer, M. M.; Ranu, S.; Krishnan, N. M. A. EGraFF-Bench: Evaluation of Equivariant Graph Neural Network Force Fields for Atomistic Simulations. arXiv, 2023; 10.48550/ARXIV.2310.02428.

(33) Schütt, K. T.; Arbabzadah, F.; Chmiela, S.; Müller, K. R.; Tkatchenko, A. Quantum-chemical insights from deep tensor neural networks. *Nature Communications* **2017**, *8*, 13890.

(34) Unke, O. T.; Meuwly, M. PhysNet: A Neural Network for Predicting Energies, Forces, Dipole Moments, and Partial Charges. *J Chem Theory Comput* **2019**, *15*, 3678–3693.

(35) Unke, O. T.; Meuwly, M. PhysNet: A Neural Network for Predicting Energies, Forces, Dipole Moments, and Partial Charges. *Journal of Chemical Theory and Computation* **2019**, *15*, 3678–3693.

(36) Unke, O. T.; Chmiela, S.; Gastegger, M.; Schütt, K. T.; Sauceda, H. E.; Müller, K.-R. SpookyNet: Learning force fields with electronic degrees of freedom and nonlocal effects. *Nature Communications* **2021**, *12*.

(37) Grimme, S. Semiempirical GGA-type density functional constructed with a long-range dispersion correction. *Journal of Computational Chemistry* **2006**, *27*, 1787–1799.

(38) Fey, M.; Lenssen, J. E. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* **2019**, *abs/1903.02428*.

(39) Eastman, P.; Behara, P. K.; Dotson, D. L.; Galvelis, R.; Herr, J. E.; Horton, J. T.; Mao, Y.; Chodera, J. D.; Pritchard, B. P.; Wang, Y. et al. SPICE, A Dataset of Drug-like Molecules and Peptides for Training Machine Learning Potentials. *Scientific Data* **2023**, *10*.

(40) Ramakrishnan, R.; Dral, P. O.; Rupp, M.; von Lilienfeld, O. A. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific Data* **2014**, *1*.

(41) Cheng, B.; Engel, E. A.; Behler, J.; Dellago, C.; Ceriotti, M. Ab initio thermodynamics of liquid and solid water. *Proceedings of the National Academy of Sciences* **2019**, *116*, 1110–1115.

(42) Chmiela, S.; Tkatchenko, A.; Sauceda, H. E.; Poltavsky, I.; Schütt, K. T.; Müller, K.-R. Machine learning of accurate energy-conserving molecular force fields. *Science Advances* **2017**, *3*.

(43) Christensen, A. S.; lilienfeld, A. V. Revised MD17 dataset (rMD17). **2020**,

(44) Chmiela, S.; Vassilev-Galindo, V.; Unke, O. T.; Kabylda, A.; Sauceda, H. E.; Tkatchenko, A.; Müller, K.-R. Accurate global machine learning force fields for molecules with hundreds of atoms. *Science Advances* **2023**, *9*.

(45) Smith, J. S.; Isayev, O.; Roitberg, A. E. ANI-1: an extensible neural network potential with DFT accuracy at force field computational cost. *Chemical Science* **2017**, *8*, 3192–3203.

(46) Smith, J. S.; Zubatyuk, R.; Nebgen, B.; Lubbers, N.; Barros, K.; Roitberg, A. E.; Isayev, O.; Tretiak, S. The ANI-1ccx and ANI-1x data sets, coupled-cluster and density functional theory properties for molecules. *Scientific Data* **2020**, *7*.

(47) Devereux, C.; Smith, J. S.; Huddleston, K. K.; Barros, K.; Zubatyuk, R.; Isayev, O.; Roitberg, A. E. Extending the Applicability of the ANI Deep Learning Molecular Potential to Sulfur and

Halogens. *Journal of Chemical Theory and Computation* **2020**, *16*, 4192–4202, PMID: 32543858.

(48) Smith, J. S.; Nebgen, B.; Lubbers, N.; Isayev, O.; Roitberg, A. E. Less is more: Sampling chemical space with active learning. *The Journal of Chemical Physics* **2018**, *148*, 241733.

(49) Jia, Z.; Padon, O.; Thomas, J.; Warszawski, T.; Zaharia, M.; Aiken, A. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York, NY, USA, 2019; p 47–62.

(50) Tillet, P.; Kung, H. T.; Cox, D. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2019). Phoenix, Arizona, United States, 2019.

(51) Appleyard, J.; Kociský, T.; Blunsom, P. Optimizing Performance of Recurrent Neural Networks on GPUs. *CoRR* **2016**, *abs/1604.01946*.

(52) Wang, G.; Lin, Y.; Yi, W. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing. 2010; pp 344–350.

(53) Filipovič, J.; Madzin, M.; Fousek, J.; Matyska, L. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* **2015**, *71*, 3934–3957.

(54) Godwin*, J.; Keck*, T.; Battaglia, P.; Bapst, V.; Kipf, T.; Li, Y.; Stachenfeld, K.; Veličković, P.; Sanchez-Gonzalez, A. Jraph: A library for graph neural networks in jax. 2020; http://github.com/deepmind/jraph.

(55) Nguyen, H.; Corporation, N. *GPU Gems 3*; Lab Companion Series v. 3; Addison-Wesley, 2008.

(56) Tang, Y.-H.; Karniadakis, G. E. Accelerating dissipative particle dynamics simulations on GPUs: Algorithms, numerics and applications. *Computer Physics Communications* **2014**, *185*, 2809–2822.

(57) Doerr, S.; Majewski, M.; Pérez, A.; Krämer, A.; Clementi, C.; Noe, F.; Giorgino, T.; Fabritiis, G. D. TorchMD: A Deep Learning Framework for Molecular Simulations. *Journal of Chemical Theory and Computation* **2021**, *17*, 2355–2363.

(58) Schoenholz, S. S.; Cubuk, E. D. JAX M.D. A Framework for Differentiable Physics. Advances in Neural Information Processing Systems. 2020.

(59) Larsen, A. H.; Mortensen, J. J.; Blomqvist, J.; Castelli, I. E.; Christensen, R.; Dułak, M.; Friis, J.; Groves, M. N.; Hammer, B.; Hargus, C. et al. The atomic simulation environment—a Python library for working with atoms. *Journal of Physics: Condensed Matter* **2017**, *29*, 273002.

(60) Pymatgen (Python Materials Genomics). https://pymatgen.org, (Accessed April 17 2024).

(61) Anderson, B.; Hy, T.-S.; Kondor, R. Cormorant: Covariant Molecular Neural Networks. Proceedings of the 33rd International Conference on Neural Information Processing Systems. Red Hook, NY, USA, 2019.

(62) Brandstetter, J.; Hesselink, R.; van der Pol, E.; Bekkers, E. J.; Welling, M. Geometric and Physical Quantities Improve E(3) Equivariant Message Passing. arXiv, 2021; 10.48550/ARXIV.2110.02905.

(63) Schütt, K. T.; Sauceda, H. E.; Kindermans, P.-J.; Tkatchenko, A.; Müller, K.-R. SchNet – A deep learning architecture

for molecules and materials. *The Journal of Chemical Physics* **2018**, *148*, 241722.

(64) Liao, Y.-L.; Smidt, T. Equiformer: Equivariant Graph Attention Transformer for 3D Atomistic Graphs. The Eleventh International Conference on Learning Representations. 2023.

(65) Gasteiger, J.; Giri, S.; Margraf, J. T.; Günnemann, S. Fast and Uncertainty-Aware Directional Message Passing for Non-Equilibrium Molecules. arXiv, 2020; 10.48550/ARXIV.2011.14115.

(66) Liu, Y.; Wang, L.; Liu, M.; Zhang, X.; Oztekin, B.; Ji, S. Spherical Message Passing for 3D Graph Networks. arXiv, 2021; 10.48550/ARXIV.2102.05013.

(67) Fu, X.; Wu, Z.; Wang, W.; Xie, T.; Keten, S.; Gomez-Bombarelli, R.; Jaakkola, T. S. Forces are not Enough: Benchmark and Critical Evaluation for Machine Learning Force Fields with Molecular Simulations. *Transactions on Machine Learning Research* **2023**, Survey Certification.

(68) Vita, J. A.; Schwalbe-Koda, D. Data efficiency and extrapolation trends in neural network interatomic potentials. *Machine Learning: Science and Technology* **2023**, *4*, 035031.

(69) Wagner, J.; Thompson, M.; Mobley, D. L.; Chodera, J.; Bannan, C.; Rizzi, A.; trevorgokey; Dotson, D. L.; Mitchell, J. A.; jaimergp et al. openforcefield/openff-toolkit: 0.14.5 Minor feature release. 2023; `https://doi.org/10.5281/zenodo.10103216`.

(70) Grimme, S.; Bannwarth, C.; Shushkov, P. A robust and accurate tight-binding quantum chemical method for structures, vibrational frequencies, and noncovalent interactions of large molecular systems parametrized for all spd-block elements (Z= 1–86). *Journal of chemical theory and computation* **2017**, *13*, 1989–2009.

(71) Sun, Q.; Berkelbach, T. C.; Blunt, N. S.; Booth, G. H.; Guo, S.; Li, Z.; Liu, J.; McClain, J. D.; Sayfutyarova, E. R.; Sharma, S. et al. PySCF: the Python-based simulations of chemistry framework. *Wiley Interdisciplinary Reviews: Computational Molecular Science* **2018**, *8*, e1340.

A Hyperparameters

In the pursuit of transparency and reproducibility, this appendix provides a detailed account of the hyperparameters employed in our computational experiments. The tables contained herein present the specific values and settings used to achieve the results discussed in the main body of this paper. Readers and fellow researchers are encouraged to refer to these tables when attempting to replicate our results or when utilizing the torchmd-train utility for their own training purposes.

Table 3: MD17 hyperparameters used for the ET training in Figure 4.

Parameter	Value
activation	silu
attn_activation	silu
batch_size	8
cutoff_lower	0.0
cutoff_upper	5.0
derivative	True
distance_influence	both
early_stopping_patience	300
ema_alpha_neg_dy	1.0
ema_alpha_y	0.05
embedding_dimension	128
lr	1e-3
lr_factor	0.8
lr_min	1e-7
lr_patience	30
lr_warmup_steps	1000
neg_dy_weight	0.8
num_heads	8
num_layers	6
num_rbf	32
seed	1
train_size	950
val_size	50
vector_cutoff	True
y_weight	0.2

Table 4: QM9 U_0 hyperparameters used to obtain the results for ET_{new} in Table 1.

Parameter	Value
activation	silu
attn_activation	silu
batch_size	128
cutoff_lower	0.0
cutoff_upper	5.0
derivative	False
distance_influence	both
early_stopping_patience	150
ema_alpha_neg_dy	1.0
ema_alpha_y	1.0
embedding_dimension	256
lr	4e-4
lr_factor	0.8
lr_min	1e-7
lr_patience	15
lr_warmup_steps	10000
neg_dy_weight	0.0
num_heads	8
num_layers	8
num_rbf	64
remove_ref_energy	true
seed	1
train_size	110000
val_size	10000
vector_cutoff	True
y_weight	1.0

Table 5: QM9 U_0 hyperparameters used for the trainings in Figure 4 to obtain the results for TensorNet 3L* in Table 1.

Parameter	Value
activation	silu
batch_size	16
cutoff_lower	0.0
cutoff_upper	5.0
derivative	False
early_stopping_patience	150
embedding_dimension	256
equivariance_invariance_group	O(3)
gradient_clipping	40
lr	1e-4
lr_factor	0.8
lr_min	1e-7
lr_patience	15
lr_warmup_steps	1000
neg_dy_weight	0.0
num_layers	3
num_rbf	64
remove_ref_energy	true
seed	2
train_size	110000
val_size	10000
y_weight	1.0

Table 6: ANI2x hyperparameters used for TensorNet training. The model was then used to run the stable molecular dynamics simulations in Figure 5.

Parameter	Value
activation	silu
batch_size	256
cutoff_lower	0.0
cutoff_upper	5.0
derivative	True
early_stopping_patience	50
embedding_dimension	128
equivariance_invariance_group	O(3)
gradient_clipping	100
lr	1e-3
lr_factor	0.5
lr_min	1e-7
lr_patience	4
lr_warmup_steps	1000
neg_dy_weight	100
num_layers	{0,2}
num_rbf	32
seed	1
train_size	0.9
val_size	0.1
y_weight	1.0