Accelerating the training of Neural Network Interatomic Potentials on a GPU using Julia

Dariyan Khan
dariyank@mit.edu

Abstract

This project will involve accelerating the training of interatomic potentials in a Julia package titled PotentialLearning.jl. This will be done using a GPU and the package Flux.jl, and we will in particular make use of Flux abstractions. Furthermore, we will explore adding parallelisation using threading, alternative neural network architectures, and heuristic weight initialization methods among other techniques to aid accelerate convergence.

1 Introduction

The CESMIX (Center for Exascale Simulation of Materials in Extreme Environments) is a group at MIT that is working to advance the state-of-the-art in predictive simulations by connecting quantum and molecular simulations of materials with cutting-edge computational and high-performance techniques. One particular interest of the group is accelerating force calculations in atomistic simulations; simulations can take several months and force calculations can take up to 90% of that time. In CESMIX, a simulation can require between 3K and 100K atoms, and between 100K and 50M time steps hence why a simulation can take a large amount of time.

The Julia programming language is being used to aid accelerating these simulations, and there are two Julia packages being developed in particular which are important for our project.

The first of these packages is InteratomicBasisPotentials.jl [1]. This provides access to implementations of the Atomic Cluster Expansion (ACE) and the Spectral Neighbor Analysis Potential (SNAP) - two powerful machine learning potentials, popular in computational modeling of materials. Potentials here refer to mathematical models that describe the interaction between atoms or molecules in a material system, which can then be used to calculate the total energy or forces acting on each atom or molecule. ACE and SNAP are designed to learn the energy and force fields from a large dataset of atomic configurations, which allows them to then accurately predict behaviour between the atoms/molecules.

The second package, which is what the majority of the code for this project will be written for, is called PotentialLearning.jl[2]. This package is being developed to simplify the training of these potentials (as well as a few other tasks). It does this by incorporating elements of bayesian inference, machine learning, differentiable programming, software composability, and high-performance computing. For this package we use ACE rather than SNAP.

If we wanted to get an exact description of how the particles interact, we would need to use the Schrödinger equation for example. However, the Schrödinger equation for more than a few particles becomes very difficult to solve, and the system is chaotic even for 3 particles! (see the three body problem). And we want to use thousands or hundreds of thousands of particles in our simulations.

Hence, in many cases, calculations are carried out using Density Functional Theory (DFT) in order to get very accurate predictions as shown in [4]. These calculations are very

expensive however, and so surrogate models (such as machine learning potentials) are used to mimic the behaviour of these tasks trained on DFT data (which contains information such as energies, forces, and stresses). Despite the success of this approach, there can still be performance limitations during training such as if the training dataset is sufficiently large. In the last few years, Sivaraman et al. (10.1038/s41524-020-00367-7, Methods/Active Learning section) [5] proposed an active learning (AL) scheme, based on a clustering algorithm, to automatically obtain a minimal training dataset. In that work, AL is defined as a ML strategy in which a learning algorithm iteratively queries a very large set of unlabeled data to extract a minimum number of training data leading to a supervised ML model with superior accuracy compared to a training model with educated manual selection.

Hence, the core idea of the PotentialLearning.jl package is to show how key features of Julia, such as composability through multiple dispatch or the latest developments in HPC abstractions, impact these methods and implementations and in particular, studying how to accelerate ACE model training using a Julia-based AL scheme.

The current code does not use any GPU accelerated methods. However, considering the large size of data needed for these simulations and the nature of how neural networks operate, training involves many matrix multiplications, hence why I believe we should be able to gain significant speed-up from GPU parallelization. In fact, a member of the group has already shown that GPU parallelization helps on a simpler case that uses the Lennard-Jones potential instead of ACE. The ACE potential is more complicated than the Lennard-Jones potential as Lennard-Jones only assumes pairwise interactions. The code can be found in this notebook: shorturl.at/oxFJV.

2 Baseline results

We tested running the current learning function in PotentialLearning.jl, that does not utilise any GPU accelerations, on the a-Hfo2-300K-NVT-6000 dataset from the paper "Machine-learned interatomic potentials by active learning: amorphous and liquid hafnium dioxide"[5], which contains approximately 588,000 lines of data. The pipeline consists of four parts.

The first two parts are to calculate the force and energy descriptors of the dataset. These are necessary to put the energy and forces in a form that can then be learned from. The initial code does not have any parallelisation or (any other methods of acceleration) implemented. When we time how long it takes for the energy and force descriptors to be calculated, we get the results seen in figure 1 and figure 2.

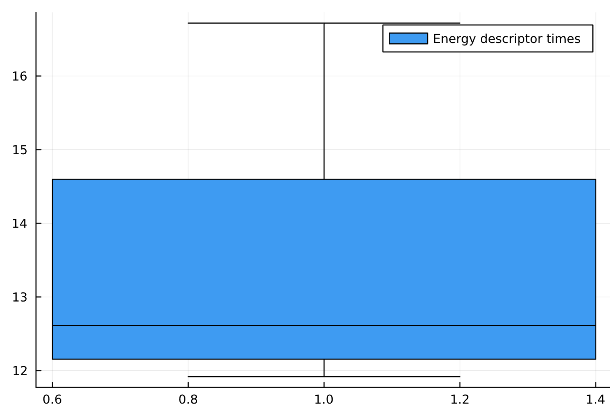

Figure 1: Time taken to calculate energy descriptors with no parallelisation

As you can see, without any acceleration computing the descriptors does not take too much time: the energy descriptors were calculated in a median time of approximately 14 seconds whereas the force descriptors took a median time of about 2.45 seconds. Nevertheless, we will still see if any parallelisation we add can speed up the computations.

Learning the forces and energies is what takes the vast majority of the time. Before timing the training process we first determine a good

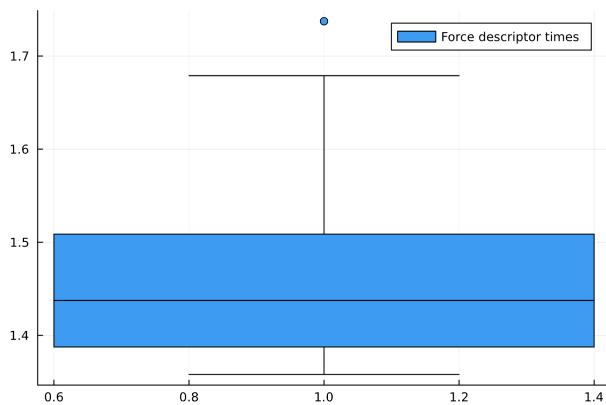

Figure 2: Time taken to calculate force descriptors with no parallelisation

learning rate of Adam to use in our model. Nevertheless, as training in this setting can take some time, we use a relatively small model as suggested by the group: a two layer neural network. The first layer has an input size of 26 and output size of 8 with a ReLU activation function, whereas the second layer has an input of size 8 and a single output. I tested neural networks with 4, 8, 16, and 32 neurons in the first layer, and found that 8 was a good choice considering time of computation and accuracy. I trained for 10 epochs with a learning rate of 0.01 and 0.001, and achieved the following results in figure 3:

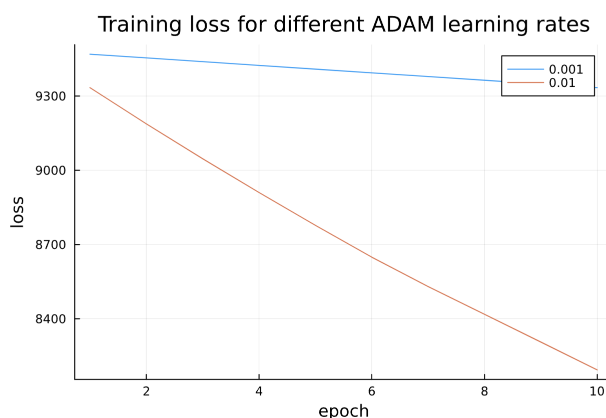

Figure 3: Times to train and minimum loss achieved under baseline

Clearly, a learning rate of 0.01 is superior and leads to a rapid change in loss.

I continued to train the ADAM(0.01) model for 60 epochs, and as the plot below demonstrates, even after 60 epochs the loss is still decreasing at a rapid rate:

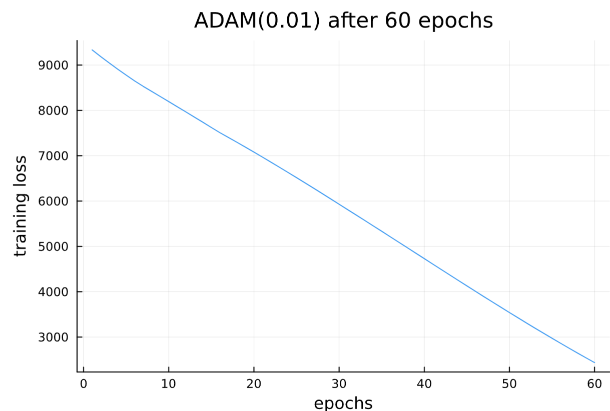

Figure 4: Training loss of ADAM(0.01) after 60 epochs

In order to time the training, I ran the training function 5 times for 2 epochs each time in order to determine how long the function takes to run.

	Run time
run 1	150.6
run 2	104.7
run 3	117.6
run 4	114.4
run 5	112.5
mean	120

Figure 5: Times to train and minimum loss achieved under baseline

The mean time for 2 epochs is 2 minutes (i.e. an epoch a minute). Since in reality this model would be run for 500 or even 1000 epochs, it is important to try and accelerate the learning as much as possible.

3 Energy and Force Descriptors

We first look at accelerating the energy and force descriptors by using threading. There are two different ways that we can parallelise: we can either parallelise the energy and force computations separately and then run one after the other, or we could calculate both descriptors at the same time. I implemented both methods and tested the results on 2, 4,

and 8 threads:

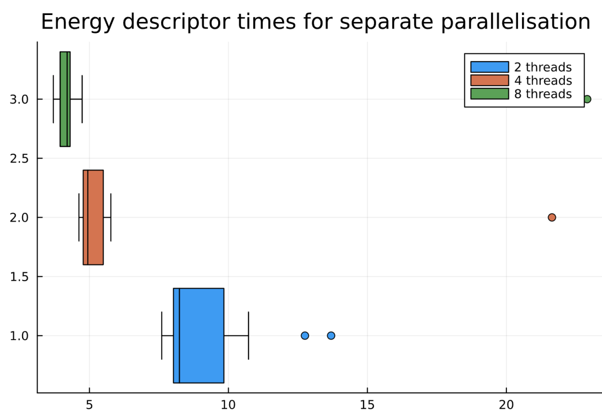

Figure 6: Time to calculate energy descriptors after parallelisation

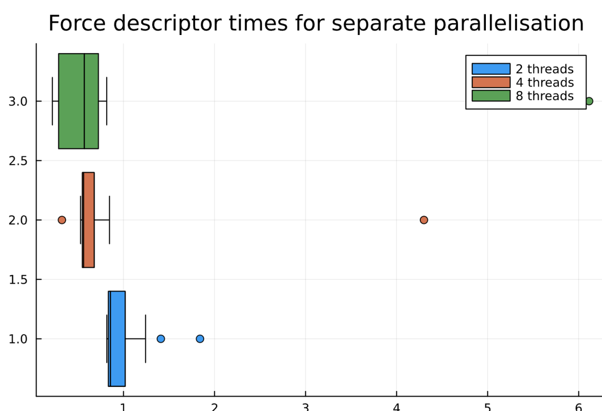

Figure 7: Time to calculate force descriptors after parallelisation

As can be seen in the figures 6, 7, and 8, we achieve the best results when we run the parallelised descriptor functions one after the other. If we run the functions using 8 threads, the energy descriptors achieves a median time of approximately 4 seconds, whereas the force descriptors calculation achieves a median of 0.5 seconds, leading to a total average time of 4.5 seconds. This is a big improvement over the baseline results, which had a total median time of approximately 13.95 seconds.

After we compute the descriptors, we can also apply PCA to reduce the number of descriptors and the time taken to train, while not drastically impacting accuracy. The plots below, produced by another person

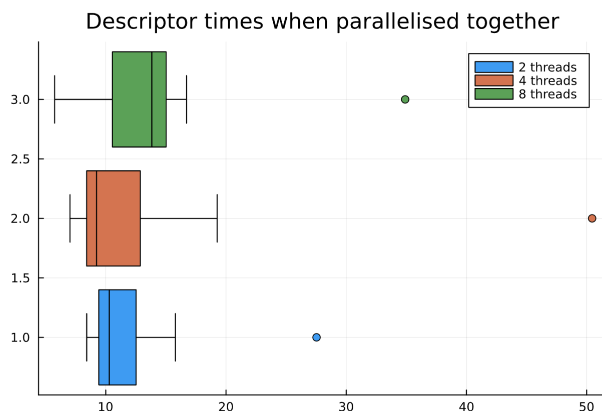

Figure 8: Time to calculate descriptors when parallelised together

in the CESMIX group, show the impact of accuracy when we apply to PCA to maintain 50 descriptors and then 20 descriptors - the original data set contains 100 descriptors:

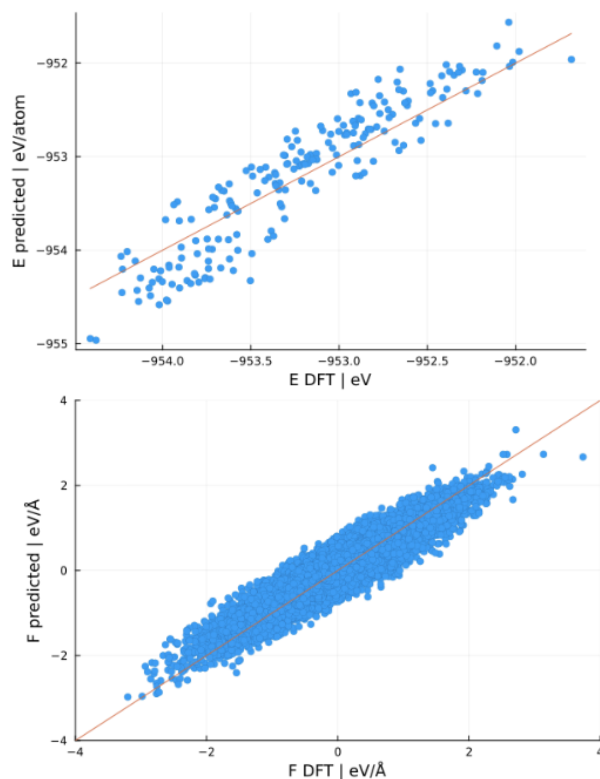

Figure 9: Accuracy obtained when training with 50 descriptors

As can be seen in figure 9 and 10, in both cases our predicted energies and forces lie relatively close to the y=x line, and decreasing the number of descriptors from 50 to 20 does not drastically affect the accuracy. But, going

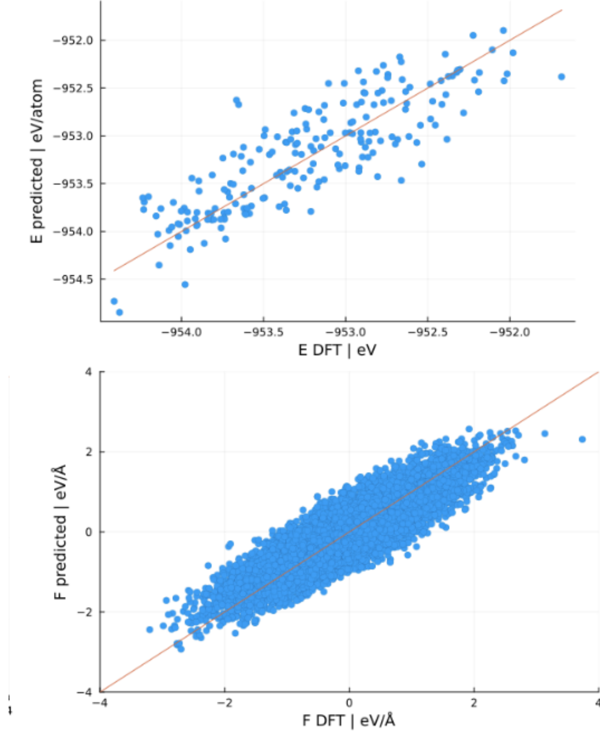

Figure 10: Accuracy obtained when training with 20 descriptors

from 50 descriptors to 20 descriptors reduces the time taken to train with 5 epochs from approximately 153 seconds to 74 seconds.

4 Training energies and forces

When training the energies and forces, we make use of two different types of descriptors: global descriptors and local descriptors. Global descriptors summarize information on the whole system (or at least a large part of it).

We define a global descriptor as Global descriptors, on the other hand, summarize larger-scale or whole-system information. They might include statistical properties of the entire system, such as the overall density of atoms, or distribution of different types of atoms among other metrics. Local descriptors, on the other hand, describe properties specific to individual atoms or a small group of atoms, for example, the types and arrangement of an atom's nearest neighbors.

When we are using ACE to compute the energies in our system, ACE computes the predicted energies as a linear combination of the global descriptors:

$$E^{ACE,s} = \mathbf{B}(\mathbf{R}^s) \cdot \beta \tag{1}$$

Figure 11: Equation for calculating energies using ACE in a non-neural network setting

In this equation, $\mathbf{R}^s$ represents a data set of atomic configurations and $\mathbf{B}(\mathbf{R}^s)$ represents the global descriptors. The vector β is arbitrary in this case and is just used to denote a linear function.

We can use a neural version of ACE to calculate the energies either as a non-linear function of the global descriptors, or as a non-linear function of local descriptors.

$$E^{NACE,s} = NN(\mathbf{B}(\mathbf{R}^s), \omega) \tag{2}$$

Figure 12: Non linear neural equation for calculating energies using global descriptors

$$E^{NACE,s} = NN(\mathbf{B}(\mathbf{R}^s), \omega) \tag{3}$$

Figure 13: Non linear neural equation for calculating energies using local descriptors

In equation (3), $\mathbf{B}_i(\mathbf{R}^s)$ indexes the local descriptors.

We time both of these two methods to see which one is faster using the BenchmarkTools package. We benchmark how long each method takes to calculate the predicted neural potential energy 5 times. We expect the global descriptor method to be faster, as we are processing the data all in one batch and hence achieving some sort of parallelisation. Just as in the baseline, we use a two layer fully connected neural network.

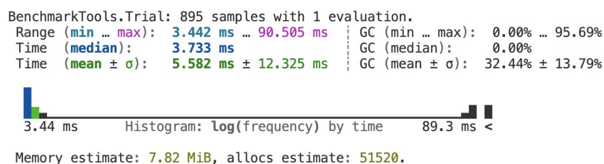

Figure 14: Time taken to calculate predicted energies using the global descriptor method

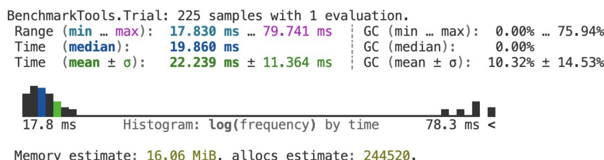

Figure 15: Time taken to calculate predicted energies using the local descriptor method

As predicted, the global descriptor method achieved a mean time of 5.582ms, whereas the local descriptor method achieved a mean time of 22.239ms. Although both times are small, the global descriptor time was 4 times quicker.

The force calculations are carried out using automatic differentiation. We define the predicted spacial component c of the force of atom i in configuration s as:

$$F_{i,c}^{Pred,s} = -\frac{\partial NN(\mathbf{B},\omega)}{\partial \mathbf{B}}\bigg|_{\mathbf{B}=\mathbf{B}(\mathbf{R}^s} \cdot \frac{\partial \mathbf{B}(\mathbf{R}}{\partial \mathbf{r}_{i,c}}\bigg|_{\mathbf{R}=\mathbf{R}^s} \quad (4)$$

Figure 16: Non linear neural equation for calculating energies using local descriptors

The first term in the product would be calculated using Zygote.jl, wheras the second term would be calculated using InteratomicBasisPotentials.jl.

Although there is only one method to calculate the forces, we will still benchmark the function over 5 epochs in order to compare how long it takes to execute compared to the energy functions:

As you can see, despite taking an average time of 1.096 seconds, the force calculations take significantly more time, and in addition require much more memory.

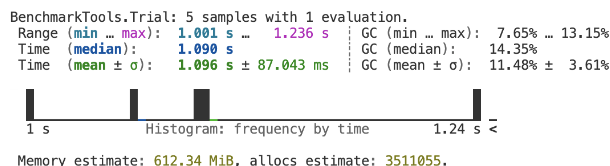

Figure 17: Time taken to calculate predicted forces

However, in order to understand which part of the training function requires the most processing time and leads to the approximate an epoch a minute baseline result, we will time how long it takes the neural network to update its weights when the loss function only depends on the mean squared error (MSE) of the predicted energies, and then only the MSE in the predicted forces. Again, we only train for 5 epochs.

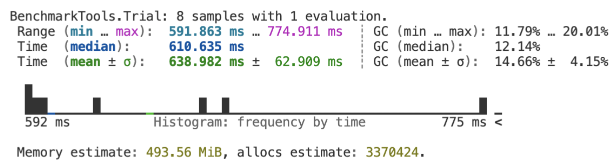

Figure 18: Time taken to update neural network weights when loss function depends only on predicted energies

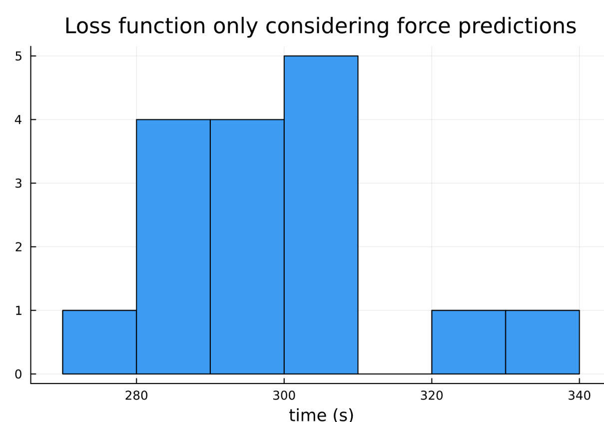

Figure 19: Time taken to update neural network weights when loss function depends only on predicted forces

Hence, the training function when run just on energy predictions takes on average 638 ms. On the other, the training function when just run on force descriptors takes on average just under 300s. Hence, the force term in the loss function is the true bottleneck in the code (but

since in reality we will train for many epochs, accelerating the learning of the energies will help tremendously too).

At first, we attempted to apply Flux's GPU abstractions directly to relevant data in the training function in order to calculate the gradients on the gpu. We wanted to do it such that we wouldn't have to deviate too much from the way the energies and forces had intended to be calculated.

Initially, we came across some roadblocks, in particular scalar indexing errors on the gpu. After tracing back through the PotenialLearning.jl package, we realised that the error had been caused by a function in the methods that compute the energy and force descriptors outputting data in Float64, whereas the neural network and the rest of the piepline was expecting Float32 values. Of course if we wwwant our models to run quickly, our data should be in Float32. On a CPU, these values can easily be converted. However, on a GPU converting the types causes the computation to serialize, negating the benefits of parallelisation obtained from a GPU and leading to slow execution and a scalar indexing error.

Once the function was amended to output Float32 values, we continued to try to apply the abstractions without an overhaul of the code, but to no avail. We realised that the force calculations would need a custom GPU kernel written for them. Hence, we decided to first work on accelerating the energies and then on accelerating the forces.

Firstly, we noted that previously the loss function for a specific configuration was calculated one at a time. Instead, we transfer batches of data to the GPU in order to make use of parallelisation and speed up performance. Doing this required the energy prediction function to be adjusted, so that the calculations are split depending on how many atoms are in each configuration. Furthermore, we added the ability to choose how many batches of configurations a user wanted to be on each epoch (initially all configurations would be processed each epoch).

I then tested the code by running the training function on the GPU for 5 epochs and achieved the following results:

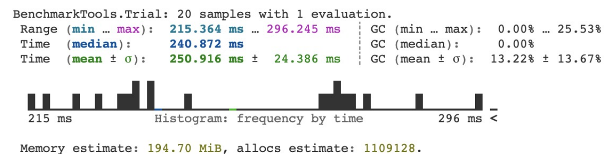

Figure 20: Time taken to train energies with the new algorithm on a gpu

The mean time is 250ms, which is 2.5 times as fast as the original algorithm (which can be seen in figure 13). Hence, we are seeing improvements in speed. Although the code for the predicted energies is mostly complete, there are still some minor changes/improvements that need to be made so that my pull request can be merged into the CESMIX project. Once this pull request is completed, I will then turn my attention back to writing a custom GPU kernel for the forces.

5 Training with a CNN

Instead of using a dense neural network, we investigate if a CNN may yield better results. According to a paper written by Junqi Yi et al. [3], CNN layers can provide benefits of reducing computational cost compared to models of fully-connected layers of a similar size, and allow the model to be independent of lattice size. Again, in order to not over-inflate the training time, we restrict ourselves to relatively simple CNNs. Although traditional CNNs consist of max pooling layers, flattening layers etc., I found that these layers were incompatible with the training function, in particular with some of the gradient calculations in Zygote.jl. Unfortunately, I haven't had time to delve into why these problems arise, so for this paper we will just utilise standard CNN layers. The

purpose is to just explore a new architecture that the group could potentially look further into at some other point. After some trial and error, a promising architecture was

```
nn_conv = Chain(
Conv((2,2), 1=>2, leakyrelu, pad=1),
Conv((2,2), 2=>1, leakyrelu, pad=1),
Conv((5,5), 1=>1, leakyrelu, pad=1),
Dense(26, 1)
)
```

I first trained the model for 60 epochs with a learning rate of 0.01, before then training the model for an extra 30 epochs with a learning rate of 0.001. This produced the following results:

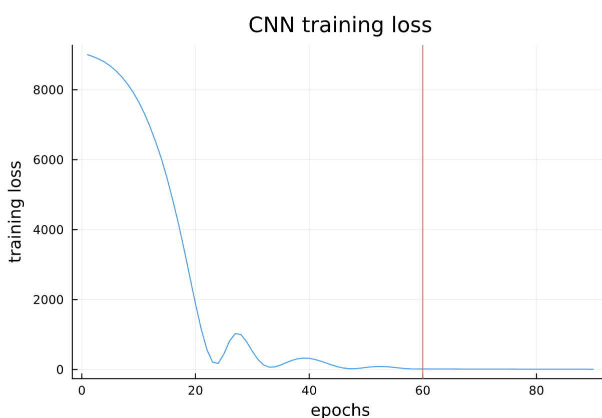

Figure 21: CNN training loss

As we can see, we get convergence in much fewer epochs with the CNN model. After 60 epochs, the model had a loss of 18.5. Then, after reducing the learning rate the loss monotonically decreased to a loss of 7.7. As in reality we would likely train for many more epochs (e.g. 500 or 1000), an architecture similar to this does seem promising.

However, the CNN does take much longer to train than the previously used fully-connected neural networks. In fact, I measured a time of approximately an epoch every 5 minutes. Hence, although we do achieve more rapid convergence, it needs to be weighed up whether the longer training time negates this.

6 Initialisation

Initially, when Flux.jl initialises a model it uses the Xavier method to initialize the weights. Although this is a good heuristic in many problems, we decided to investigate whether a better initialization exists that aids in accelerating convergence. In particular, we look for a parallel heuristic to precondition the ML model weights and biases, to accelerate convergence, using threads. One option that we investigated was first training a model just on the energies, then training another the model just on the forces, before then setting the initial parameters to be the average of the parameters of the aforementioned models. After training the initial energy and force models each for 3 epochs and then training the 'full' model for 60 epochs, we achieved the subsequent results:

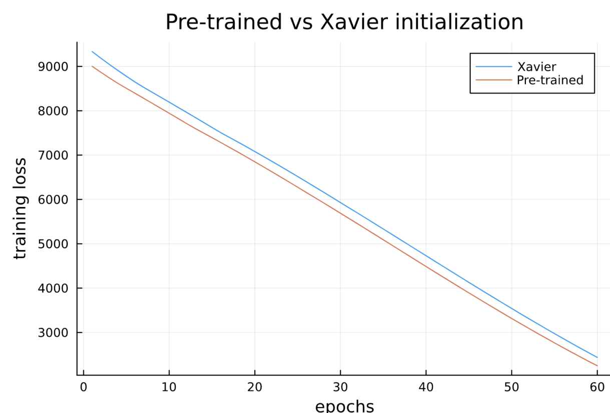

Figure 22: Pre-trained weights vs Xavier initialization

As can be seen although the pre-trained weight curve is below the Xavier initialization curve, they both have almost identical gradients throughout, and remember that the pre-trained weights require training for 3 epochs beforehand, which effectively negates any benefits it provides.

In the future, we would like to explore other methods or variations we can make to aid convergence.

7 Conclusion

In conclusion, although the accelerations are still very much a working progress, speed-ups in learning the energies with a gpu have been demonstrated, and a variety of other methods such as using different architectures or threading have provided promising results. In the future, once my energy accelerations code has been merged with the repository, I would like to work on creating the custom GPU kernel for force training, as well as investigate other CNN architecture and heuristic weight initializations.

Most of the code that I have written can be found within:
https://shorturl.at/vAHV9

and

https://shorturl.at/BFRW2

The second link contains code pertaining to sections 5 and 6 of the paper. I have uploaded code for these files in silo with the rest of the repository.

References

[1] *InteratomicBasisPotentials.jl: Julia package for interatomic basis potentials*. Accessed: 03/24/2023.

[2] *PotentialLearning.jl: a Julia library for active learning of interatomic potentials in atomistic simulations of materials*. Accessed: 03/24/2023.

[3] Junqi Yi Jiaxin Zhang Kipton Barros Massimiliano Lupo Pasini, Ying Wai Li and Markus Eisenbach. Fast and stable deep-learning predictions of material properties for solid solution alloys. Accessed: 05/11/2023.

[4] Y. Mishin. Machine-learning interatomic potentials for materials science. Jun 2021. Accessed: 05/13/2023.

[5] Ganesh Sivaraman, Aravind N. Krishnamoorthy, Matthias Baur, et al. Machine-learned interatomic potentials by active learning: amorphous and liquid hafnium dioxide. *npj Computational Materials*, 6(1):104, Jul 2020. Accessed: 03/24/2023.