CS 6804 Optimization in Machine Learning Final Project:

Improve performance of Atomistic Machine Learning Package

Jiamin Wang, Chemical Engineering Department

Background and Motivation

In computational chemistry field, the feed-forward class of artificial neural networks has seen the greatest application, especially in case of large training datasets. In this Atomistic Machine Learning package, it also employ the neural network model. Training is basically adjusting the model parameters to fit experimental data, like the energy, interatomic forces calculated from conventional quantum mechanics DFT. The process of fitting is carried out by minimizing a loss function. Here the loss function defined in this package is shown below:

$$\Gamma = \frac{1}{2}\sum_{j=1}^{M}\{\left(\frac{\widehat{E}_j}{N_j} - \frac{E_j}{N_j}\right)^2 + \frac{\alpha}{3N_j}\sum_{k=1}^{3}\sum_{i=1}^{N_j}\left(\widehat{F}_{ik} - F_{ik}\right)^2\}$$

Where the atom-normalized residuals of the machine-learned energies $\widehat{E}_j$ are compared to the training(electronic structure) energies E_j. The parameter M is the number of energy data points or atomic configurations(one energy data point per atomic configuration) and N_j is the number of atoms in the atomic configuration j. The squared error loss function can be minimized by adjusting model parameters. The interatomic forces are gradients of the potential energy surface, which provide more information than the energy points. For example the system I used to benchmark include 14 atoms which are two oxygen atoms on 12 Ru surface, this system will have 14*3 = 42 unique force values, whereas it will only have a single potential energy. In the loss function, F_{ik} and $\widehat{F_{ik}}$ are the quantum mechanics and machine-learned forces for atom i in the direction k, and α is the learning rate, which determine the path of convergence, ideally should be chosen such that both energy convergence and force convergence happen together.

For the neural network model, the backpropagation method can be used to efficiently calculate the gradient of the loss function with respect to the parameters. Compared with a variety of gradient-based optimization algorithm, this can be used to find optimal model parameters by minimizing the loss function. But the loss function for a neural network mode is not convex and has multiple minima, so the gradient-descent procedure is dependent upon the initial guess of parameters, when starting with a random set of initial model parameters.

Strategies to Improve the Amp training and preliminary results

1*) The gradient-based optimizer might get trapped in a local minimum which is not deep enough to yield a satisfactory small value of loss function.*

Solution: Global search scheme simulated annealing is implemented to skip from the current local basin to a nearby deeper basin. It has been also suggested that preprocessing of the input data to the neural network can significantly improve the training.

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function. It is inspired by the metallurgic process of annealing whereby metals must be cooled at a regular schedule in order to settle into their lower energy state. As the loss function hold a non-convex form, it is desirable to perform a global search in parameter space before the gradient descent optimization.

The procedure is firstly random searching in parameter space, which contains multiple local minimum. Then the parameters corresponding to the minimum loss function were found and the gradient-descent optimization is performed to find the minimum basin found in the first step**.** The simulated-annealing function is used from open-source code of Wagner and Perry[1], the process involves:

1. Randomly move or alter the state
2. Assess the energy of new state using an objective function
3. Compare the energy to the previous state and decide whether to accept the new solution or reject it based on the current temperature.
4. Repeat until you have converged on an acceptable answer.

Simulated annealing parameters are shown below. How to tune the parameters heavily depend on your real problem, the objective function and parameter space.

$$T_{max} \# \, max\ starting\ temperature:$$

$$T_{min} \# \, min\ ending\ temperature$$

$$steps \# \, number\ of\ iterations$$

$$updates \# \, number\ of\ updates$$

Starting temperature T_{max} should be able to accept 98% of the moves, If T_{max} takes a small value (greater than zero), then the algorithm reduces to the simple random-walk search. The ending temperature should be low enough such that the final solution will not be improved. The iteration steps might affect the results, which should be large enough to adequately explore the parameter space to guarantee to find the global minima. The number of updates are useful to exam the annealing process. The default setup can print out all the information includes the current temperature, state energy, the percentage of moves accepted and improved and remaining time. All these four simulated annealing parameters implemented in Amp are listed as below:

$$T_{max} = 2000.0$$

$$T_{min} = 2.5$$

$$steps = 10000$$

$$updates = steps/200 = 50$$

The training datasets are *ab initio* molecular dynamics ~200 trajectories, the neural network architecture is two hidden layers, and each layer contain 20 nodes. The simulated annealing model was implemented in the *utilities.py.* The way to import annealer is shown below:

From amp.utilities import Annealer
Annealer(calc=calc, images=train_images, Tmax=2000, Tmin=1, steps=4000)

I only use 300 trajectories training energy to test the performance, smaller training datasets might cause the training results not stable. If simulated annealing was used before training, the time used for training to reach the criteria is 2.6min, while without the simulated annealing, it will take 2.7 min to accomplish.

2) Optimizer L-BFGS is implemented to deal with larger size datasets

Amp implemented different gradient descent optimizers, such as steepest descent, conjugate gradient, and the Broyden- Fletcher-Goldfarb-Shanno(BFGS) algorithm.

However, Urban A *et al.*[3] proved that Levenberg-Marquardt(LM) method is more efficient for small ANN architectures than other algorithms. But this method is computationally demanding for the larger ANN architectures and training sets, the reason is that the dimension of the Hessian matrix that has to be inverted at each LM step is determined by the number of weight parameters and reference structures. While the L-BFGS can be parallelized when in reference structure space, so L-BFGS is the method of choice for larger reference sets and architectures.

The Amp call the optimizer from the open-source software library scipy for the loss-function minimization, so the implementation of L-BFGS optimizer in Amp is very straightforward, just switch from BFGS to L-BFGS-B.

```
if optimizer == 'BFGS':
        from scipy.optimize import fmin_bfgs as optimizer
        optimizer_kwargs = {'gtol': 1e-15, }
elif optimizer == 'L-BFGS':
        from scipy.optimize import fmin_l_bfgs_b as optimizer
        optimizer_kwargs = {'pgtol': 1e-08,}
```

Here, I used 3000 trajectories, approximated 126,000 data points. The training results are in the following two figures, left figure is the results from optimizer BFGS and right is from L-BFGS-B. L-BFGS has the lower rmse for both training and testing datasets compared to BFGS. L-BFGS takes more iterations and each iteration takes less time compared to BFGS. Besides, L-BFGS take less space to store the training parameters.

Fig. L-BFGS optimizer used for training

Fig. BFGS optimizer used for training