ænet-PyTorch

ænet

The Atomic Energy NETwork (**ænet**) package (http://ann.atomistic.net) is a collection of tools for the construction and application of atomic interaction potentials based on artificial neural networks (ANN). ANN potentials generated with **ænet** can then be used in larger scale atomistic simulations.

ænet-PyTorch

ænet-PyTorch is an extension of that code to allow GPU-support for the training process of **ænet**, substituting the `train.x` training step. It is enterily written in PyTorch and includes new features that the previous code did not: the ability to fit reference forces in addition to energies with GPU support. **ænet-PyTorch** is fully compatible with all the **ænet** tools: interfaces with LAMMPS and TINKER, and ASE.

This is the documentation for the extension of **ænet** to allow training on GPUs using PyTorch. This assumes familiarity with the previous implementation in Fortran 90, and only the changes with respect to it are mentioned here. For a more detailed explanation the user is referred to the **ænet** documentation.

Installation

Installation of ænet

The modified version of ænet can be installed the same way as ænet. See its documentation or any of the tutorials available in the ænet for a comprehensive guide on how to install it. In short, these are the steps to follow:

1. Compile the L-BFGS-B library

   - Enter the directory "aenet_modified/lib"

     ```
     $ cd aenet_modified/lib
     ```

   - Adjust the compiler settings in the "Makefile" and compile the library with

     ```
     $ make
     ```

   The library file `liblbfgsb.a`, required for compiling **ænet**, will be created.

2. Compile the **ænet** package

- Enter the directory “aenet_modified/src”

  ```
  $ cd aenet_modified/src
  ```

- Compile the ænet source code with an approproiate `Makefile.XXX`

  ```
  $ make -f makefiles/Makefile.XXX
  ```

The following executables will be generated in “./bin”:

- `generate.x`: generate training sets from atomic structure files
- `train.x`: train new neural network potentials
- `predict.x`: use existing ANN potentials for energy/force prediction

Installation of ænet-PyTorch

ænet-PyTorch is enterily written in Python, using the PyTorch framework. In the following the required packages will be listed. It is highly recommended to install all the packages in an isolated Python environment, using tools such as virtual-env (https://virtualenv.pypa.io/en/latest/) or a conda environment (https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html) .

- `Python`: 3.7 or higher
- `Numpy`: 1.19 or higher
- `PyTorch`: 1.10 or higher
- `CUDA`: 10.2 or higher (optional for GPU support)

We will assume that `CUDA` and Python 3.7 are already installed.

1. Install PyTorch 1.10

   - Installation using pip with CUDA support

     ```
     $ pip install torch==1.10.1+cu102 -f https://download.pytorch.org/whl/cu102/torch_s
     ```

     or for only CPU usage

     ```
     $ pip install torch==1.10.1+cpu -f https://download.pytorch.org/whl/cpu/torch_stabl
     ```

2. Install Numpy

   - Numpy should be automatically installed when installing PyTorch.

Installation of the tools for ænet-PyTorch

Compile the tools needed to make **ænet-PyTorch** compatible with **ænet**

- Enter the directory “pytorch-aenet/tools”

  ```
  cd tools
  ```

- Compile the tools

  ```
  make
  ```

The following exacutables will be generated in the same "tools/" directory:

- `trainbin2ASCII.x`: convert the output from generate.x (`XXX.train`) to a format readable by **ænet-PyTorch** (`XXX.train.ascii`).
- `nnASCII2bin.x`: convert the **ænet-PyTorch** output files (`XXX.nn.ascii`) to the usual binary files (`XXX.nn`).

Usage

A detailed explation of the data format of the reference structures and the selection of the atomic descriptors can be found in the **ænet** documentation. In the following, the main changes to the generation of descriptors and all the novelties in the training will be explained.

Training set generation with `generate.x`

If only energy training is needed, the generation of the descriptors is the same as in ænet. If force training in PyTorch is wanted, there are two new keywords in the generate.in input file file: `FORCES` and `FORCESPERCENT`.

Execution

The generation of the descriptors is carried out using the `generate.x` tool

```
$ generate.x generate.in > generate.out
```

In order to read the binary file `.train` in **ænet-PyTorch** it needs to be transformed using the `trainbin2ASCII.x` tool

```
$ pytorch-aenet/tools/trainbin2ASCII.x [NAME].train [NAME].train.ASCII
```

List of keywords

- `output` (**default** : refdata.train)
  Defines the path to the training set file that is going to be generated. If forces are required, another file with the same name but with '.forces' extension will be created.
- `types`
  Defines the number of atomic species, their names, and atomic energies. The first line after the keyword contains the number of different species

<NT>; the following <NT> lines each contain the chemical symbol <T_i> and atomic energy <E_atom-i> of one species.

- `setups`
  Specifies paths to structural fingerprint basis function setup files. Each of the <NT> lines following the keyword contains the chemical symbol <T_i> and the path to the setup file for one species.
- `forces` (**default** : False)
  If present, output all the necessary information to compute atomic forces. Output is stored in `.forces` binary file.
- `forcespercent` (**default** : 100)
  Percentage of reference structures to be used in the training of forces set.

Input file template (generate.in)

```
OUTPUT  <path/to/output/file>

TYPES
<NT>
<T_1>   <E_atom-1>
<T_2>   <E_atom-2>
...
<T_NT>  <E_atom-NT>

SETUPS
<T_1>   <path/to/setup-1>
<T_2>   <path/to/setup-2>
...
<T_NT>  <path/to/setup-NT>

FORCES
FORCESPERCENT    <NF>

FILES
<NF>
<path/to/file-1.xsf>
<path/to/file-2.xsf>
...
<path/to/file-NF.xsf>
```

Training with ænet-PyTorch using `main.py`

ANN potential training with **ænet-PyTorch** requires the output generated by `generate.x`. The train.in input file is similar to the one of ænet, but with some additional parameters.

Assuming that the ænet-PyTorch source files are stored in the directory called `~/pytorch-aenet/src/`. If not specified, the input file is assumed to be called `train.in`. The training is done as follows:

```
$ ~/pytorch-aenet/src/main.py [train.in]
```

Once the training is done, we need to transform the neural network output from ASCII to Fortran90 binary. Assuming that the neural network file-name for its element is stored as `$element.pytorch.nn.ascii`:

```
for element in [list of elements]; do
    ./tools/nnASCII2bin.x $element.pytorch.nn.ascii $ element.pytorch.nn
done
```

List of keywords

- `trainingset`
  Path to the `.train.ascii` training set file produced by `generate.x` and transformed by `trainbin2ASCII.x`.
- `testpercent` (**default** : 10)
  Percentage of reference structures to be used as independent testing set.
- `iterations`
  Number of training iterations/epochs. One iteration is defined as going through the whole dataset.
- `iterwrite` (**default** : 1)
  Print train and test errors every `iterwrite` iterations.
- `batchsize` (**default** : 256)
  Number of structures used to compute the error and its gradients in each batch-iteration.
- `maxenergy` (**default** : False)
  Highest formation energy to include in the training. If present, all the structures with higher energy are excluded from the training process.
- `forces` (**default** : False)
  If present, training will be performed on forces as well. It requires `.train.ascii.forces` to be present. The next line contains options to tune the force training:
  - `alpha`
    Weight of the force error when summing it with the energy error.
- `maxforces` (**default** : False)
  Highest force item to include in the training. Structures with at least one force component on one of its atoms with higher force value are excluded from the training.

- `memory_mode` (**default** : cpu)
  Controls the storage of the information to compute the energy and forces of each batch.
  - `cpu` Store everything on cpu RAM, and send to GPU only the information of the current batch.
  - `gpu` Store everything on the GPU VRAM.
- `save_energy` (**default** : False)
  Write final energies of all the structures of the training and test sets.
- `save_forces` (**default** : False)
  Write final forces of the atoms of all the structures of the training and test sets.
- `verbose` (**default** : False)
  If present, long output format will be displayed.
- `regularization` (**default** : 0)
  Regularization or weight decay. Value of the L2 regularization.
- `method`
  Training algorithm to be used for the weight optimization. The line following the keyword contains as first item the name of the method and then the learning rate is specified with the 'lr' keyword. Currently available: adam, adadelta, adagrad, adamw, adamax. Any other algorithm included in PyTorch can be used by including it in `/pytorch-aenet/src/traininit.py` (in the `init\_optimizer` function).
- `networks`
  Defines the architectures and specifies files for all ANNs. Each of the `<NT>` (= number of types) lines following the keyword contains the chemical symbol `<T_i>` of the *i*-th atomic species in the training set, the path to the ANN output file, and the architecture of the hidden network layers. The latter is defined by the number of hidden layers followed by the number of nodes and the activation function separated by a colon.

  Accepted activation functions are linear, tanh or sigmoid.

Input file template (train.in)

```
TRAININGSET <path/to/data/ascii/file>
TESTPERCENT <percentage>
ITERATIONS  <NI>
ITERWRITE   <NW>
BATCHSIZE   <NB>

MAXENERGY   <emax>
MAXFORCES   <fmax>
```

```
VERBOSE

MEMORY_MODE <cpu or gpu or disk>

FORCES
alpha=<alpha>

METHOD
method=<method name>   lr=<learning rate>

REGULARIZATION <l2>

NETWORKS
! atom   network         hidden
! types  file-name       layers  nodes:activation
  <T_1>  <path/to/net-1>    2    10:tanh    10:tanh
  <T_2>  <path/to/net-2>    2    10:tanh    10:tanh
```

Restarting training

When the training is finished, a file called `model.restart` is created, with the necessary information to restart the training. If that file is present when starting a new training, it will automatically restart from that checkpoint. Note that if parameters such as the network architecture are changed, the training will fail.