LISCENCE

Abstract

This code is designed for the training and running of atomistic neural network potentials. It pulls training data from sql files which should include energies, forces, and atomic coordinates. It is modeled after the ANAKIN-ME methodology with modifications to accelerate the training process. It is designed to be run in conjunction with aevmod (https://pypi.org/project/aevmod/). The models use ANI style atomic environment vectors (AEVs). For details on AEV descriptors see DOI:https://doi.org/10.1039/C6SC05720A. PyKinML offers training to a single fidelity level, multiple fidelity levels, forces, and delta learning. It also allows the user to take advantage of Pytorch's Distributed Data Parallel functionality to train across multiple GPUs or nodes.

1. Overview of arguments

Before training a model use pykinml.prepper.parse_arguments_list() to create an args object which will provide instructions to the code on how to prepare the data and train the model. The example files contain instructions for setting specific arguments for different training modes.

fidlevel:
    fidelity level of data to be pulled from SQL file

fidlevel_lf:
    If args.delta=True or performing multi fidelity training, the lower fidelity level.

input_data_type (string):
    file type for input_data_fname.

input_data_fname (string):
    path to files holding data. Readable via glob.

trtsid-name (string):
    name of file holding the tvtmask (train/validation/test). Format:
    database/xid int
    Where database is the name of the file holding the structure, xid is the id of that structure in the database, and int is 2 for training/validation or -1 for testing

tvt (list):
    portion of the dataset to be used for training, validation, testing. Format can be:
    [int, int, int]: if no trtsid-name is given. Will choose proper amount of structures from input_data_fnam
    [float, float, float] (sum <= 1.0): if no trtsid-name is given. Will partition all structures pulled according to fractions provided
    [int, int, 0]. if trtsid-name is provided. Will pull only structures listed in trtsid-name from each database. All ids labeled -1 in trtsid-name will be used for testing and int+int will be used for validation and training (chosen randomly).

sae-fit (bool):
    Use single atom energies (SAEs) to shift energies to be centered around 0 and reduces the range of energies when training to variable molecule sizes. Details for how this works are provided in https://doi.org/10.1021/acs.jpca.3c07872

nameset (list of strings):
    List of names for structures in SQL file. If there is any meta information you wish to use to identify or categorize structures in your database you can include a name for those structures, (e.g. red molecules, blue molecules). if you then wish to pull only those structures from the database, pass nameset containing the names [red molecules, blue molecules].

pre-saved (bool):
    training, testing and validation data has already been preprocessed and is saved in data_path During the preprocessing of the data, data.py saves dataset split into (training+validation) and testing. This ensures that the preprocessing of the data only needs to occur once even if a model is trained in multiple phases or if different models are trained to the same dataset.

data_path (str):
    if pre-saved is True: path to the pre-saved data.
    If pre-saved is false: this is ignored.

optimizer:
    Specify the optimizer type [SGD or Adam (default) or AdamW] as implemented in pytorch

lrscheduler (str):
    learning rate schedular:
    exp: ExponentialLR
    step: StepLR
    rop (default): ReduceLROnPlateau

learning-rate (float):
    initial learning rate. default=1.e-3
    Note: When using DDP we find that best results are achieved when you multiply the learning rate by the number of GPUs being used.

LR_patience (int):

if lrscheduler==rop: this is the patience (in epochs) used to determine when to reduce the learning rate
else: this is ignored

LR_threshold (float):
if lrscheduler==rop: this is the threshold (in kcal/mol) used to determine when to reduce the learning rate
else: this is ignored

epochs (int):
How many epochs to train

save_every (int):
how often (in epochs) to save the model

load-model (bool):
Load an already trained model

load_model_name (str):
if load-model: This is the path to the model to load
else: this is ignored

tr_batch_size (int):
training batch size

vl_batch_size (int):
validation batch size
Note that vl_batch_size does not affect the model accuracy but can affect the training time and memory usage

ts_batch_size (int):
testing batch size
Note that ts_batch_size does not affect the model accuracy but can affect the training time and memory usage

my-neurons (list of ints):
Number of nodes per layer. Final layer is output layer. If model is only trained to predict energies, the final layer should always be 1. Length must match that of my-actfn

my-actfn (list of strings):
    Activation functions for each layer. Length must match that of my-neurons

no-biases (bool):
    Train without biases. default=False

savenm (str):
    location to save the model

randomseed ([int, int]):
    random seeds to help with reproducibility.

floss (bool):
    Include forces in the loss function. This can potentially greatly increase the accuracy of your model but will also slow down training and increase memory usage.

optimize-force-weight (bool):
    if floss: Rather than give a static value for the force to energy ratio in the loss function, the ratio can be set as a learnable parameter within the model. See DOI 10.1109/CVPR.2018.00781 for details.
    else: This is ignored.

fw (float):
    if floss and not optimize-force-weight: Set weight of force in loss function. Total_loss=(1-fw)*Energy_loss + fw*Force_loss
    else: This is ignored.

aev_params (list of 3 ints):
    hyper parameters for AEV construction

cuttoff_radius (list of 2 floats):
    radial and angular cutoff used for AEV construction

beta (float):
    scaling factor used for AEV angular component to avoid NAN at 0 and pi angles. Recommended slightly less than 1.0. Default=0.95

temp (scaler):
    temperature associated with structure in SQL database

delta (bool):
    Prepare data for delta learning. If True, for each structure, two sets of energies and forces will be pulled from the database specified in args.input_data_fname for each structure (one energy at fidlevel and one at fidlevel_lf) and the difference between the two will be used for training. If any structure does not have energies and forces at both fidelity levels, an error will occur.

multi_fid (bool):
    Train to two fidelity levels simultaneously. The first set of layers predicts the lower fidelity level and the second predicts the difference between the lower and higher fidelity levels.

ddp (bool):
    Train using Pytorch's Distributed Data Parallel

gpus (list of ints):
    If DDP: Indices of the GPUs to use for training.
    else: This is ignored.

Data Preparation

Data preparation is primarily handled by data.py and prepper.py. First, prepper.prep_data() is called. If a file is specified in args.trtsid_name, prep_data will read the the tvt mask in this file to determine which structures to pull from the database specified in args.input_data_fname and which of those will be used for training/validation/testing. If no file is given in args.trtsid, all structures with the specified fidelity level will be pulled from the database and the splitting between training/validation/testing will be random. In either case, how many structures will be used for training/validation/testing can be specified in args.tvt.

Prep.prep_data() calls data.get_data() which pulls the structures from the database. These structures are then sent aevmod which will compute their Atomic Environment Vectors (AEVs) which will serve as the input for the model. If args.floss=True (meaning you are performing force training), aevmod will also calculate the Jacobian of the AEV.

The next step is to separate the data into training/validation and testing. In pyKinML, the training and validation sets are pulled from the same pool of molecules and the split between the two is random. The separation of the data is handled by prepper.prep_training_data() and prepper.prep_testing_data(). In addition to separating the data these functions also save the data which will be loaded later by prepper.load_data(). This is done so that the data preparation and training can be done independently of one another.

Finally, if args.sae_fit=True, the single atom energies (SAEs) to use will be fit from the training data. The SAEs are used to reduce the range of energy values and set the mean of the training energies to 0. For details on SAE calculation, see https://doi.org/10.1021/acs.jpca.3c07872.

Training a model

The first step when training a model is to load the data. The previous section discussed how the data is pulled from the sql file, separated into training/validation and test set, and saved. The prepper.load_data() function loads the data from the path specified in args.data_path and separates the training and validation set. It also prepares a set of ‘keys’ which are the keys of the dictionaries containing the training, validation, and test data.
The load_data function is called inside prepper.load_train_objs(), which also calls the prep_model function which sets up the model, optimizer, and learning rate scheduler.
The training loop for single fidelity models for pyKinML is held inside the Trainer class in trainer.py. An example for how to run this class can be seen in examples/train_single_fid.py.

This function handles training on a single GPU or cpu. When training on multiple gpus, torch.multiprocessing.spawn calls this function for each GPU. When called, This function

1. initializes DDP via ddp_setup function in trianer.py (only if args.ddp=true)

2. Calls prepper.load_train_objs() to setup model, optimizer, learning rate scheduler, and dataset.

3. Calls set_up_task_weights to balance weighting of energy and force in the loss function. Note that if args.floss is False this is still called, but the force weight will be set to zero and forces will not be calculated during training.

4. Wraps the model in DistributedDataParallel (if args.ddp is True) and send the model to the device where training will take place

5. Initializes Trainer class:

6. Train for args.epochs

7. destroy_process_group() (if args.ddp)

By default the Trainer class tries to minimize the L2 loss. It uses the function my_loss found in prepper.py.

Training a multi-fidelity model

Training a multi-fidelity model is very similar to training a single fidelity model. Before the training loop starts prepper.prep_data() is called twice, once for the high fidelity data and once for the low fidelity. It is important that for each structure in the dataset there exists an energy value (and force if perfuming force training) for each fidelity level you are training to.

When preparing the network PyKinML makes one network for each fidelity level using the same network architecture for each. The two networks are connected by sending the output of the low fidelity network as part of the input to the high-fidelity network. The low fidelity network is identical to single fidelity network. Once it outputs an atomic energy, that energy value is appended to the end of its atom's AEV and used as the input for the high fidelity network. The model then returns both the low fidelity output and the sum of the high and low outputs. The loss is computed independently for each output then summed. The backpropagation step tries to minimize the sum of the L2 error of each fidelity level.

Force Training

In PyKinML the AEV that is used as model input and it's Jacobian are both calculated using AEVMOD (https://github.com/sandialabs/aevmod)
When training, the AEVs (and their Jacobians) are calculated only once before the training loop starts and then used throughout the loop, rather than being recalculated every epoch.

The forces are calculated by taking the derivative of the energy with respect to the AEV (dE/dAEV) and using Pytorch's matmul to perform matrix between dE/dAEV the AEV jacobian (dAEV/dxyz) to get the derivative of the energy with respect to the atomic coordinated (dE/dxyz), aka, the forces.

When performing force training, the balance between the forces and energies in the loss function can greatly affect the accuracy of the trained model. PyKinML offers two ways to decide this balance. The first is by setting the value of args.fw. If this method is used, the loss will be calculated as

$$L_{Total} = (1 - fw) * L_{energy} + fw * L_{force}$$

The second option is to set args.optimize-force-weight=True. If this second option is chosen the balance between the forces and energies will be set as a learnable parameter and updated during the back propogation step of training. DOI 10.1109/CVPR.2018.00781 outlies how this is done. Within pyKinML, this is done by the the task_weights class within pykinml/prepper.py. This task weight optimization can be easily expanded to new task by simply setting num_tasks=N where N is a positive integer.

When args.optimize-force-weight=True the total loss is calculated as:

$$L_{Total} = {}^{1}/_{e^{\sigma_{energy}}} * L_{energy} + \sigma_{energy} + {}^{1}/_{e^{\sigma_{force}}} * L_{force} + \sigma_{force}$$

Where $\sigma_{energy}$and $\sigma_{force}$are learnable parameters initially each set to 0.

Note that during training a value is always set for each σ, even when training only to energies. During training this value is output by the model alongside the model prediction. This was done in order to keep the optimize_force_weight feature available when using DDP as DDP requires that learnable parameters interact with

the model output during the model's forward call. If you are training only to energies, this value is unused.

During the preprocessing of the data, when performing force training, the force data and aev Jacobians are padded with extra arrays of value zero to account for differences in the number of atoms between molecules. The "true" length of each of the forces is saved in a list called fdims which is used to ensure only the true forces are used when calculating the loss.

Examples

The examples directory holds several useful examples that cover different training methods. After installing PyKinML it is recommended to run these examples to both familiarize yourself with the software and ensure that it is working properly. The exact output of each example will vary on different machines but each should have a summary of the data preparation/loading, the model architecture, and the error at each epoch.

train_single_fid:

train a model. You should see a number outputs related to the data preparation, such as “parsing SQLite xyz data base data_holder/C5H5.db” and the time taken to generate the AEVs. It should also print the model architecture that looks like

```
net: CompositeNetworks(
 (NNs): ParameterList(
    (0): Object of type: ModuleList
    (1): Object of type: ModuleList
  (0): ModuleList(
    (0): LinearBlock(
     (linear): Linear(in_features=224, out_features=48, bias=False)
    )
    (1): LinearBlock(
     (linear): Linear(in_features=48, out_features=24, bias=False)
    )
    (2): LinearBlock(
     (linear): Linear(in_features=24, out_features=12, bias=False)
    )
    (3): LinearBlock(
     (linear): Linear(in_features=12, out_features=1, bias=False)
    )
  )
  (1): ModuleList(
    (0): LinearBlock(
     (linear): Linear(in_features=224, out_features=48, bias=False)
    )
    (1): LinearBlock(
     (linear): Linear(in_features=48, out_features=24, bias=False)
    )
    (2): LinearBlock(
     (linear): Linear(in_features=24, out_features=12, bias=False)
    )
    (3): LinearBlock(
     (linear): Linear(in_features=12, out_features=1, bias=False)
    )
  )
 )
)
```

load_model:

Loads the model trained by train_single_fid and continues training. Although this example uses the same training set, you can use a different one by simply setting the arguments for data preparation. You can even choose to train to forces even if the first phase of training was only to energy, or vise versa. Note that in order to train to forces args.floss needs to be True during data preparation.

run_model:

uses the model trained by train_single_fid to compute the energy and atomic forces for a molecule (in eV and eV/Å respectively).

train_delta:

Train a model to predict the difference between two fidelity levels. Although we use the same dataset here as in train_multi_fid, the architecture is set up as a single fidelity model as only one energy prediction is output by the model. The difference between the two fidelity levels is taken during the data preparation and as such the saved dataset reflects this.

train_multi_fid:

Train a model two fidelity levels simultaneously. Two neural networks will be set up, one for each fidelity level. The output of the first network will be fed into the second network. The model will then output it's energy prediction for the lower fidelity level and its prediction for the difference between the low fidelity level and the high fidelity level.

train_multi_node:

identical to train_single_fid but set up to run on multiple nodes in parallel when submitted through a slurm script.