Netrain

Netrain aims to simplify the process of training a huge number of neural networks on the PNI clusters. It was originally developed as a bookkeeping system for use when training thousands of networks for hyperparameter optimization. Though it is now clear that training thousands of networks is computationally impractical given, Netrain remains a valuable tool to have for any project which involves experimenting with a large number of ANN models.

Netrain tool enables you to run highly configurable ANN training schedules and keep track of them in a very organized fashion. It generates a self contained project for each batch of networks you train and helps you keep track of all your batches. This keeps your networks organized and makes it very easy to annotate certain training runs, make changes to the code for particular networks, or retrain certain networks. with Netrain everything stays organized and modular.

Workflow

Netrain answers the call of duty as soon as you have a bunch of network models you want to train. Netrain allows you to organize your models by type and specify which models of which type to train. Say you are working on 3 autoencoder models and 4 classifier models, the former group named `"auto"` and the latter group named `"class"`, Netrain allows you to specify exactly which models from these two groups to train. In netrain language, a model's group is called its `"class"` and its specific structure is called its `"architecture"`.

The general workflow is as follows:

1. **Configuration**

- Configure training and evaluation data generator functions for each network model
- Configure training specifications for each model
- Configure a model generator function; this allows you to alter different hyperparameters for the same general model structure.

2. **Build**

- Specify (a) a group of networks to train and (b) parameters for training (number of epochs, batches per epoch, etc)
- Use the netrain build tool to generate a self contained project for this training run.

- Submit an array job to the cluster.

What you'll eventually want to do is to generate a self contained project for that batch and store it in some first step is to generate a self contained project for this batch of networks. To set this up you will need to follow these steps:

Batch Submit Builder

1 – Specify Training Build Directory

Netrain builds a new mini project each time you train a set of networks. First, you will need to specify the directory to which Netrain will write each of these self contained training codebases. Create such a directory and one you have done that, find the `Constants.py` file in `src`. Search for the constant `TRAININGS_DATA_DIR`. It should say `os.path.join(PATH_PREFIX, 'trainings')` .Replace this with the directory you created to which netrain will write training code.

2 – Tell netrain about your networks

The only way netrain can train networks is if it knows which networks to train.

1. First, you'll need to get the Netrain API into your python path so you can use it in your code.
2. In your project directory, make a file called `netrain_main.py`. Every time a job starts running on the cluster, this file `netrain_main.py` is responsible for implementing the training of the models. `netrain_main.py` is passed command line arguments which specify the parameters for the model training delegated to it.
3. Navigate to the netrain `src` folder in Netrain and open the `Constants.py` file. Find the variable named `MAIN_SCRIPT` and set it to `path/to/netrain_main.py`.
4. Then, go back and insert the following into `netrain_main.py`

```
import os
os.path.append("path/to/netrain/src/")
from Netrain import Netrain
```

5. Now that you have imported `Netrain` into this script, you'll need to instantiate a `Netrain` application object and pass it a **model configuration function**. Read on to see what this function does.

3 – Model Configuration Function

The model configuration function is integral. It is passed to the `Netrain` application instance and returns a dictionary containing all the information necessary for Netrain to automatically get models running.

As an apriori, append the following to your `netrain_main.py`:

```
app = Netrain(model_configuration_function)
app.run()
```

Now, you'll need to actually define `model_configurator`. Remember that `netrain_main.py` is passed command line arguments to specify which models it should train? Well, `model_configuration_function` is called by the `Netrain` application instance and passed these input parameters, specified below, and expected to return another dictionary specified subsequently.

Configuration Function Input

Field	Value Specification
`"class"`	A `string` specifying group of the model to be trained by this job. In our case, this would be either `"auto"` or `"class"`
`"architecture"`	The name of the model within the group `Class`.
`"nb_epoch"`	The number of epochs all models trained by this script should run for.
`"epoch_samples"`	How many samples per epoch for all models trained by this script
`"nnets"`	The number of models trained by this script. NOTE: they will all have the same architecture. The option to train multiple models in one script is useful if (1) the cluster is busy so you want to take advantage of allocated jobs by training multiple models with them, (2) you want to do hyperparameter optimization for this model of these models.
`"intelligent"`	Whether the model training should auto-terminate once a certain error threshold is met
`"wpath"`	Path to the directory to which mdoel weights are stored
`"archpath"`	Path to the directory to which architectures are stored
`"metricspath"`	Path to the directory to which metrics are stored

Once passed a dictionary with this data, the `model_configuration_function` must return a dictionary with the following data all present. If all fields are not present, then something will break. So you must be sure to be positive all fields are present.

Configuration Function Output

"name"	
"nb_epoch"	The number of epochs to train this model with
"archpath"	The directory where architectures are stored (so you can actually use them after training). You can just use the `archpath` from the configuration function's input dictionary.
"wpath"	The directory where model weights are stored (obviously the most integral directory of them all). You can just use the `wpath` from the configuration function's input dictionary.
"metricspath"	The directory where model metrics are stored. You can just use the `model` from the configuration function's input dictionary.
"model_generator"	
"training_generator"	A generator function that returns training data for the model. Output ought to be a two-tuple of the form `(x, y)` where `x` is the measured input and `y` is the measured output which the model tries to predict.
"evaluation_generator"	A generator function that returns evaluation data for the model. Output ought to be a two-tuple of the form `(x, y)` where `x` is the measured input and `y` is the measured output which the model tries to predict.
"optimizer"	Which optimization scheme to use (see keras optimizers https://keras.io/optimizers/ for the valid options)
"loss"	Which loss function to use for the model (see keras objectives for the valid options https://keras.io/objectives/)

WARNING – Make sure `wpath` and `archpath` have enough space to store all your networks and their architectures before you begin training them.