Contents

A Src-level Tour of EMADE (specifically the nlp/nn branch(es))

Drafted by Anish Thite. Send me a message on Slack for questions/corrections/memes

QuickStart/Running:

- Make sure to set up the mysql database properly (nothing in there, name/password matches on input file, etc.)
- python src/GPFramework/seeding_from_file.py templates/input_movie_reviews.xml seeding_test_nn
- python src/GPFramework/launchEMADE.py templates/input_movie_reviews.xml

Theory of EMADE:

What's The Point?

Given a dataset, EMADE can create an algorithm that will optimize for a variety of different objectives. The algorithm is a tree-like structure where nodes are primitives or terminals (functions and/or literals).

Terminology Review

- **Primitive**: Nodes in our tree representation of an algorithm (typically a function). EMADE optimizes an algorithm created from a tree of these primitives.
- **Terminals**: The leaf nodes in our tree structure (typically constants)

EMADE vs DEAP:

EMADE differs from DEAP in a few ways: * EMADE uses a tree structure to represent algorithms (Lists r 4 noobs) * EMADE seeks to optimize based on multiple objectives * EMADE has 5 letters and DEAP has 4

The Worker/Master Relationship:

The master computer generates the individuals, and hands the indivudals to the worker(s) to evaluate. The worker(s) than return the evaluated individuals to the master. In `EMADE.py` you will find a function called `workeralgorithm` or something along those lines. This is where the worker algorithm is specified.

`EMADE.py`

This is where the master algorithm is run. Very few people actually know what it does. The exteremly simplified version is: 1. Evaluation 2. Fitness Computation 3. Selection 4. Mating 5. Mutation

The templates folder

This is where all of the input xml files are located. Database information, worker/master parameters, and evolution parameters (which mutation functions to use, which mating functions to use, fitness functions, etc.) are defined here.

Primitives

Most primitives are stored in files ending in _methods.py. However, this is not a biconditional, so not all _methods.py contain primitives. The good news is that documentation for this is well-defined so check the top of the file for what kind of functions it contains.

A typical primitive would be definied as such:

```
def get_max(data):
    return np.max(data)
```

We typically can make calls to a library or another implementation of the function elsewhere. For example we use gensim's implementation of Word2Vec. Once you have primitives defined, you can seed those primitives so that EMADE will start optimizing from them.

You also have to add the primitive to the toolbox. This is done in `gp_framework_helper.py`. An example of such code would be :

```
pset.addPrimitive(tp.count_vectorizer,[EmadeDataPair, bool, int, int,
   int], EmadeDataPair, name='CountVectorizer')
```

In this case we add a text processing primitive found in the text_processing_methods.py file we had previously imported as 'tp'. We also specify input types, output type, and a name.

DataPairs and Learning

If you're a fellow bandwagoner, you want to run EMADE with some Machine Learning primitives. There are a few primitives that you especially want to make sure you understand:

`EmadeDataPair`

This is the data that we pass up through our individual. It contains both the train and test data.

`Learner`

The Learner primitive particularly perplexed me at first. A Learner Primitive takes in data and a model and fits the model on the data. It then uses the trained model to make predictions on the test data which is bundled up and sent to the eval functions.

Seeding

EMADE starts off with a seeding file, and optimizes from the seed. For example, lets look at the `nlp_seeding_file`. The first four lines are:

```
Learner(CountVectorizer(ARG0, trueBool, 0, 0, 235), learnerType('LOGR',
   {'penalty':0, 'C': 1.0}, 'SINGLE', None))
Learner(TfidfVectorizer(ARG0, falseBool, 0, 1, 1121),
   learnerType('RAND_FOREST', {'n_estimators': 100, 'criterion':0,
   'max_depth': 3, 'class_weight':0}, 'SINGLE', None))
Learner(TfidfVectorizer(ARG0, falseBool, 0, 0, 45), learnerType('LOGR',
   {'penalty':0, 'C': 1.0}, 'SINGLE', None))
Learner(TfidfVectorizer(ARG0, trueBool, 0, 1, 10), learnerType('LINSVC',
   {'C':1.0}, 'SINGLE', None))
```

Each one of these lines specifies an individual in our starting population. If you create a primitive, you can put in an algorithm using these primitives here. Run the instructions in seeding_from_file.py so that the individuals in the seed file are loaded into the mysql database BEFORE you run `launchEMADE.py`

Neural Network Branch Stuff

Goal: Architecture search and hyperparameter tuning via EMADE

Implementation:

We are implementing neural networks using the Keras API (tensorflow backend).

We introduce a variety of primitives:

- NNLearner(DataPair, LayerList)
  - This is analgous to a Learner. It takes in an EMADE DataPair and a LayerList.
- LayerList
  - Literally a list of Layers, nothing more to it.
- Layers

- There are a variety of layers we use, such as Embedding, GRU, LSTM, and Dense. These layer functions create a dictionary representing the layer type as well as important parameters for that layer
- eg. A Dense layer with an out_dim of 10: `{"DENSE": [10]}` Here is where the magic starts:

The NNLearner iterated over the LayerList, and creates Keras layers using the parameters specified in the dictionaries. It adds these layers to a Keras model, and fits it on the training data. We use this model to make predictions for the test data which is then bundled up to send to the eval functions. Since this model does not need to be used later and the lays of physics forbid infinite memory, we make sure to remove it from VRAM (still to be tested).

So why does this (theoretically) work? If we treat the layer types and the parameters as primitives EMADE can mutate, mate, and select based off of them.

How to add Primitives to an NNLearner:

1) Make a Layer function for your layer (or activation function). It should take in a `layerlist` as well as any relevant parameters you need (for example out dimension). Your function should add a dictionary to the layerlist with the key being some sort of unique identifier string (name of layer/activation is fine) and the value being a list of necessary parameters.

2) Add the layer function to the pset in `gp_framework_helper.py`. Also add it to `modify_list`. The code is something like:

```
pset.addPrimitive(nnm.DenseLayer, [int, str, nnm.LayerList],
   nnm.LayerList, name='DenseLayer')

modify_list.append('DenseLayer')
```

   Explanation: The first parameter is the function to call (your Layer function). After that are the types of parameters to you function specified as a list. Following is the return type of your function, and finally is a name for the primitive. The primitive name is stored in `modify_list`

3) Here you tell `NNLearner` how to use your dictionary to actually add the layer to the Keras model (add the import statement for the Keras function at the top of the file)! For example:

```
if  name == 'DENSE':
    model.add(Dense(params[0],kernel_initializer=params[1]))
```

   Make sure the name in the condition matches the unique identifier set in step 1. You can access params via the `params` list.

Let us know if you have any questions!