PyGOP Documentation

Release 0.2.2

Dat Thanh Tran

Apr 25, 2019

CONTENTS:

PyGOP provides a reference implementation of existing algorithms using Generalized Operational Perceptron (GOP) based on Keras and Tensorflow library. The implementation adopts a user-friendly interface while allowing a high level of customization including user-defined operators, custom loss function, custom metric functions that requires full batch evaluation such as precision, recall or f1. In addition, PyGOP supports different computation environments (CPU/GPU) for both single machine and cluster using SLURM job scheduler. What's more? Since training GOP-based algorithms might take days, PyGOP allows resuming to what has been learned in case the script got interfered in the middle during the progression!

CHAPTER

ONE

WHAT IS GENERALIZED OPERATIONAL PERCEPTRON?

Generalized Operational Perceptron (GOP) is an artificial neuron model that was proposed to replace the traditional McCulloch-Pitts neuron model. While standard perceptron model only performs a linear transformation followed by non-linear thresholding, GOP model encapsulates a diversity of both linear and non-linear operations (with traditional perceptron as a special case). Each GOP is characterized by learnable synaptic weights and an operator set comprising of 3 types of operations: nodal operation, pooling operation and activation operation. The 3 types of operations performed by a GOP loosely resemble the neuronal activities in a biological learning system of mammals in which each neuron conducts electrical signals over three distinct operations:

- Modification of input signal from the synapse connection in the Dendrites.
- Pooling operation of the modified input signals in the Soma.
- Sending pulses when the pooled potential exceeds a limit in the Axon hillock.

By defining a set of nodal operators, pooling operators and activation operators, each GOP can select the suitable operators based on the problem at hand. Thus learning a GOP-based network involves finding the suitable operators as well as updating the synaptic weights. The author of GOP proposed Progressive Operational Perceptron (POP) algorithm to progressively learn GOP-based networks. Later, Heterogeneous Multilayer Generalized Operational Perceptron (HeMLGOP) algorithm and its variants (HoMLGOP, HeMLRN, HoMLRN) were proposed to learn heterogeneous architecture of GOPs with efficient operator set search procedure. In addition, fast version of POP (POPfast) was proposed together with memory extensions (POPmemO, POPmemH) that augment POPfast by incorporating memory path.

CHAPTER

TWO

DOCUMENTATION OVERVIEW

- *Installation* gives instruction on how to install PyGOP through pip or source.
- *Quick Start* gives a brief introduction on how to use PyGOP interface.
- *Illustrative Examples* gives illustrative examples on CelebA and Mnist dataset with complete code.
- *Data Feeding Mechanism* gives instruction on the data feeding mechanism of PyGOP.
- *Common Interface* is dedicated to common parameters and interface shared by all algorithms.
- *Computation Environments* discusses how to setup parameters related to computation devices and computation environment such as single machine or cluster.
- *Algorithms* gives brief description of each algorithm and algorithm-specific parameters.
- *Customization* details how to define custom loss, metrics or operators.
- *Change Log* documents major changes between versions.
- *Contributions* gives instructions on how to contribute to PyGOP.
- *License* details license statement.

2.1 Installation

2.1.1 PyPi installation

Tensorflow is required before installing PyGOP. To install tensorflow CPU version through *pip*:

```
pip install tensorflow
```

Or the GPU version:

```
pip install tensorflow-gpu
```

To install PyGOP with required dependencies:

```
pip install pygop
```

At the moment, PyGOP only supports Linux with both python 2 and python 3 (tested on Python 2.7 and Python 3.4, 3.5, 3.6, 3.7 with tensorflow for cpu)

2.1.2 Installation from source

To install latest version from github, clone the source from the project repository and install with setup.py:

```
git clone https://github.com/viebboy/PyGOP
cd PyGOP
python setup.py install --user
```

2.2 Quick Start

This library includes the implementation of the following models:

- *Progressive Operational Perceptron (POP)*
- *Heterogeneous Multilayer Operational Perceptron (HeMLGOP)*
- *Homogeneous Multilayer Operational Perceptron (HoMLGOP)*
- *Heterogeneous Multilayer Randomized Network (HeMLRN)*
- *Homogeneous Multilayer Randomized Network (HoMLRN)*
- *Fast Progressive Operational Perceptron (POPfast)*
- *Progressive Operational Perceptron with Memory (POPmem)* (POPmemO and POPmemH)

The library exposes a common interface for all models. Start by importing all the models:

```
from GOP import models
```

To create an instance of a model, e.g., POP:

```
model = models.POP()
```

Default parameters can be retrieved by calling *get_default_paramters*:

```
params = model.get_default_parameters()
```

For full description of parameters of each model, please refer to *Algorithms*. This library adopts a particular data feeding mechanism that uses python generator. This design allows low memory usage even with large datasets, flexible and efficient user-defined preprocessing steps.

Generally, to feed the data to a model, the model accepts a **data_func** (a python function) and **data_arguments** (any type that is can be pickled) with the assumption that a python generator and the number of mini-batches can be produced with syntax:

```
data_gen, steps = data_func(data_arguments)
```

For example, if *data.pickle* contains *x_train* and *y_train* and *data.pickle* resides in directory *data_dir*, we can simply have **data_func** and **data_arguments** as follows:

```
import numpy as np

def data_func(data_arguments):
"""Define data function that loads pickled data from filename
   and yield mini-batch of batch_size

"""
```

(continues on next page)

(continued from previous page)

```
    # 1st element from data_arguments is filename
    # 2nd element is batch size
    filename, batch_size = data_arguments

    # load pickled data from filename
    with open(filename, 'r') as fid:
        data = pickle.load(filename)

    x_train = data['x_train']
    y_train = data['y_train']

    # calculate number of mini-batches, suppose 1st dimension of x_train is #samples
    N = x_train_shape[0]
    steps = int(np.ceil( N / float(batch_size)))

    # give definition of generator

    def gen():
        while True:
            for step in range(steps):
                start_idx = step*batch_size
                stop_idx = min(N, (step+1)*batch_size)

                yield x_train[start_idx:stop_idx], y_train[start_idx:stop_idx]


    return gen(), steps # note that gen() but not gen

# Now define data_argument
data_arguments = ['data_dir/data.pickle', 128]
```

With **data_func** and **data_argument**, we can fit the model by simply calling *fit*

```
performance, progressive_history, finetune_history = model.fit(params, data_func,␣
↪data_argument)
```

performance is a dictionary of best performances (loss and metrics), *progressive_history* contains all performances during progressive learning step and *finetune_history* contains all performances (at each epoch) during the fine-tuning step.

The trained model can be serialised and saved to disk with the given filename, e.g. 'pop_model.pickle' using *save*:

```
model.save('pop_model.pickle')
```

The pickled model can be loaded again later using *load*:

```
model = models.POP()
model.load('pop_model.pickle')
```

Using this trained model to evaluate test data e.g., *test_func* and *test_arguments* with new metrics, e.g. *mean_absolute_error*:

```
metrics = ['mean_absolute_error',]
performance = model.evaluate(test_func, test_arguments, metrics)
# performance is a dictionary of a single key 'mean_absolute_error'
```

Or using this trained model to predict (*predict*) with unseen data e.g., *new_data_func*, *new_data_arugments*. Note that

the generator produced by *new_data_func(new_data_arguments)* should only yield x but not (x,y):

```
prediction = model.predict(new_data_func, new_data_arguments)
```

Or finetune this trained model using *finetune* on potentially new training data and select best model settings through validation data and also report performances on test data

```
history, performance = model.finetune(params, train_func, train_data, val_func, val_
↪data, test_func, test_data)
```

While the above example is for POP, all other algorithms have the same interface, thus can be used in the same way. Different model, however, requires some specific parameters which should be consulted from *Algorithms*.

To configure computation environment (using CPU/GPU or using cluster), please refer *Computation Environments*

For more discussion on data feeding mechanism, please refer *Data Feeding Mechanism*

To deal with customization such as using custom loss, custom metrics or custom operators for nodal, pooling and activation, please refer *Customization*

Finally, it's worth noting that in case the script got interfered before completing the progressive learning step, PyGOP allows resuming to what has been learned as long as the 'tmp_dir' and 'model_name' in params have not been modified

2.3 Illustrative Examples

In this section, we illustrate a complete usage of all algorithms through a hand-written digits recognition task and a face recognition task.

2.3.1 Hand-written Digits Recognition with Mnist dataset

The complete example is available from https://github.com/viebboy/PyGOP/tree/master/examples/train_mnist.py . 'train_mnist.py' is the only source code we need, beside having PyGOP installed

Since we will use Mnist dataset available from Keras, we will simply create a data function by loading Mnist from keras and create a generator to generate mini-batches of data according to the batch size and decide whether to shuffle the data depending on the train or test set:

```
def data_func(data_argument):
    """
    Data function of mnist for PyGOP models which should produce a generator and the␣
↪number
    of steps per epoch

    Args:
        data_argument: a tuple of batch_size and split ('train' or 'test')

    Return:
        generator, steps_per_epoch

    """
    batch_size, split = data_argument

    # load dataset from keras datasets
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

(continues on next page)

(continued from previous page)

```
    if split == 'train':
        X = x_train
        Y = y_train
    else:
        X = x_test
        Y = y_test

    # reshape image to vector
    X = np.reshape(X, (-1, 28 * 28))
    # convert to one-hot vector of classes
    Y = to_categorical(Y, 10)
    N = X.shape[0]

    steps_per_epoch = int(np.ceil(N/ float(batch_size)))

    def gen():
        while True:
            indices = np.arange(N)
            # if train set, shuffle data in each epoch
            if split == 'train':
                np.random.shuffle(indices)

            for step in range(steps_per_epoch):
                start_idx = step * batch_size
                stop_idx = min(N, (step + 1) * batch_size)
                idx = indices[start_idx:stop_idx]
                yield X[idx], Y[idx]

    # it's important to return generator object, which is gen() with the bracket
    return gen(), steps_per_epoch
```

It's worth noting that PyGOP will pickle *data_argument* passed to *data_func*, we should not pass the actual dataset through *data_argument* because it is computationally expensive. *data_argument* should ideally contain hyper-parameters of the generator only, such as *batch_size* and *split* above, or as you will see from the next example, a path to the actual data. In this way, we can perform the data reading step inside *data_func* to avoid many write and read operations of the actual data.

The main function then continues with parsing the two arguments: the name of the model and the type of computation (cpu/gpu):

```
try:
    opts, args = getopt.getopt(argv, "m:c:")
except getopt.GetoptError:
    print('train_mnist.py -m <model> -c <computation option cpu/gpu>')
    sys.exit(2)

for opt, arg in opts:
    if opt == '-m':
        model_name = arg
    if opt == '-c':
        computation = arg
```

The type of model/algorithm is in *model_name* variable, and the type of computation is in *computation* variable. The main function continues with creating the corresponding model instance and setting the model's parameters:

```
# input 728 raw pixel values
# output 10 class probability
input_dim = 28 * 28
output_dim = 10

if computation == 'cpu':
    search_computation = ('cpu', 8)
    finetune_computation = ('cpu', )
else:
    search_computation = ('gpu', [0, 1, 2, 3])
    finetune_computation = ('gpu', [0, 1, 2, 3])

if model_name == 'hemlgop':
    Model = models.HeMLGOP
elif model_name == 'homlgop':
    Model = models.HoMLGOP
elif model_name == 'hemlrn':
    Model = models.HeMLRN
elif model_name == 'homlrn':
    Model = models.HoMLRN
elif model_name == 'pop':
    Model = models.POP
elif model_name == 'popfast':
    Model = models.POPfast
elif model_name == 'popmemo':
    Model = models.POPmemO
elif model_name == 'popmemh':
    Model = models.POPmemH
else:
    raise Exception('Unsupported model %s' % model_name)

# create model
model = Model()
model_name += '_mnist'

# get default parameters and assign some specific values
params = model.get_default_parameters()

tmp_dir = os.path.join(os.getcwd(), 'tmp')
if not os.path.exists(tmp_dir):
    os.mkdir(tmp_dir)

params['tmp_dir'] = tmp_dir
params['model_name'] = model_name
params['input_dim'] = input_dim
params['output_dim'] = output_dim
params['metrics'] = ['acc', ]
params['loss'] = 'categorical_crossentropy'
params['output_activation'] = 'softmax'
params['convergence_measure'] = 'acc'
params['direction'] = 'higher'
params['search_computation'] = search_computation
params['finetune_computation'] = finetune_computation
params['output_activation'] = 'softmax'
params['input_dropout'] = 0.2
params['weight_constraint'] = 3.0
params['weight_constraint_finetune'] = 3.0
```

(continues on next page)

(continued from previous page)

```
params['optimizer'] = 'adam'
params['lr_train'] = (1e-3, 1e-4, 1e-5)
params['epoch_train'] = (60, 60, 60)
params['lr_finetune'] = (1e-3, 1e-4, 1e-5)
params['epoch_finetune'] = (60, 60, 60)
params['direct_computation'] = False
params['max_block'] = 5
params['block_size'] = 40
params['max_layer'] = 4
params['max_topology'] = [200, 200, 200, 200]
```

To train the model instance, we simply call the *fit()* method from the model instance, using *train_func* as specified above:

```
batch_size = 64
start_time = time.time()

performance, _, _ = model.fit(params,
                              train_func=data_func,
                              train_data=[batch_size, 'train'],
                              val_func=None,
                              val_data=None,
                              test_func=data_func,
                              test_data=[batch_size, 'test'],
                              verbose=True)

stop_time = time.time()
```

In order to run the script using *Heterogeneous Multilayer Operational Perceptron (HeMLGOP)* algorithm, for example, with cpu, simply run the following command:

```
python train_mnist.py -m hemlgop -c cpu
```

This will train the model with 8 parallel threads on cpu. The number of cpu threads or the gpu devices can be set within *train_mnist.py*

2.3.2 Face Recognition with CelebA dataset

The dataset is a small subset of CelebA dataset including facial images of 20 identities, each having 100/30/30 train/validation/test images. We have extracted the deep features (using pretrained VGGface) to be used as input to all networks.

Preparation

To run this example, please fetch the examples directory from https://github.com/viebboy/PyGOP/tree/master/examples

The examples directory includes the following files:

- prepare_miniCelebA.py: this script loads raw images and generate deep features. However, we have extracted aand also provide the features, which can be downloaded via this
- data_utility.py: this script includes the data loading functionalities.
- train_miniCelebA.py: the training script used for all algorithms.

To run this example, it suffices to just download miniCelebA_deepfeatures.tar.gz and extract it to the same folder as train_miniCelebA.py and data_utility.py.

However, readers who want to do the data extraction process by their own can download the raw data miniCelebA.tar.gz and extract the data in the same example folder. After that, running prepare_miniCelebA.py will generate the deep features in data directory in the same directory. Since prepare_miniCelebA.py requires a package called keras_vggface, which uses an older version of keras. It is advised to create a new environment when running this data preparation script to prevent breaking your current setup of keras.

Usage

After preparing the necessary files and data, the example folder should hold at least the following content

- examples/data_utility.py
- examples/train_miniCelebA.py
- examples/data/miniCelebA_x_train.npy
- examples/data/miniCelebA_y_train.npy
- examples/data/miniCelebA_x_val.npy
- examples/data/miniCelebA_y_val.npy
- examples/data/miniCelebA_x_test.npy
- examples/data/miniCelebA_y_test.npy

The signature of train_miniCelebA.py is as follows:

```
python train_miniCelebA.py -m <model name> -c <computation device (cpu/gpu)>
```

For example, to train HeMLGOP on CPU, we simply run:

```
python train_miniCelebA.py -m hemlgop -c cpu
```

or training POP on GPU, we simply run:

```
python train_miniCelebA.py -m pop -c gpu
```

For CPU, we have configured the script to run 8 parallel processes, and for GPU we have configured the script to run on 4 GPUs. Please change the configuration inside train_miniCelebA.py to suit your setup.

After completing the training process, the performance and time taken will be written in result.txt in the same folder.

2.4 Data Feeding Mechanism

As briefly mentioned in *Quick Start*, this library adopts a particular data feeding mechanism that requires the user to give a function that returns a *data generator* and *the number of steps*, in other words, *the number of mini-batches* in an epoch.

The reference about python generator can be found here. Generally, a python generator is defined in a very similar way as a function, but with the *yield* statement. An example of a generator that takes 2 numpy arrays *X*, *Y* and produces mini-batch of size *batch_size* is given below:

```
import numpy as np

def example_generator(X, Y, batch_size, no_mb):
"""An example generator that takes 2 numpy arrays X, Y and batch size and number of␣
↪mini-batches

"""

    # assume 1st dimension of X and Y is the number of samples
    N = X.shape[0]

    # this while statement allows this generator to generate data infinitely
    while True:
        for step in range(no_mb):
            start_idx = step*batch_size
            stop_idx = min(N, (step+1)*batch_size) # dont allow index out of range

            x = X[start_idx:stop_idx]
            y = Y[start_idx:stop_idx]

            """
            Potentially some processing steps here
            """

            yield x, y # produce pair (x,y)
```

Note that after generating data for N steps, the for loop finishes and the while loop continues to run new iteration. The sequence of N mini-batches is exactly the same for each iteration of the while loop (or each epoch). This behavior, however, should be avoided when using stochastic gradient descend methods. There should be randomness at each iteration in the way data is generated. Below is a slight modification of the *example_generator* that introduces randomness:

```
import numpy as np
import random

def example_generator(X, Y, batch_size, no_mb):
"""An example generator that takes 2 numpy arrays X, Y and batch size and number of␣
↪mini-batches

"""
    # assume 1st dimension of X and Y is the number of samples
    N = X.shape[0]

    # this while statement allows this generator to generate data infinitely
    while True:

        # generate the list of indices and shuffle
        indices = range(N)
        random.shuffle(indices)

        for step in range(no_mb):
            start_idx = step*batch_size
            stop_idx = min(N, (step+1)*batch_size) # dont allow index out of range

            x = X[indices[start_idx:stop_idx]]
            y = Y[indices[start_idx:stop_idx]]
```

(continues on next page)

(continued from previous page)

```
            """
            Potentially some processing steps here
            """

            yield x, y # produce pair (x,y)
```

Using this definition of *data generator*, the user needs also to define a function that returns *data generators* and *the number of mini-batches*. Let assume that the data is stored on disk in a pickled format. We can write a simple *data_func* as follows:

```
import pickle
import numpy as np

def data_func(filename):
"""An example of data_func that returns example_generator and number of batch
"""

    with open(filename, 'r') as fid:
        data = pickle.load(fid)

    # assume that X, Y is stored as elements in dictionary data
    X, Y = data['X'], data['Y']

    N = X.shape[0] # number of samples
    batch_size = 128 # size of mini-batch
    no_mb = int(np.ceil(N/float(batch_size))) # calculate number of mini-batches

    # get an instance of example_generator
    gen = example_generator(X, Y, batch_size, no_mb)

    # return generator and number of mini-batches
    return gen, no_mb
```

The above example of *data_func* takes the path to the data file, performs data loading, calculates the number of mini-batches and returns an instance of *example_generator* and *number of mini-batches*.

Since data_func and data_argument will be serialized and written to disk during computation, it is recommended to pass small parameters through data_argument such as filename. Although it is possible to pass the actual data as data_argument, doing so would incur overhead computation

2.5 Common Interface

2.5.1 Common parameters

Parameters are given to a model through a dictionary **params**. While different model has different model-specific parameters, there are some common parameters that possessed by all models. Parameters are given as (key,value) pairs in the dictionary **params**. Description of common parameters are below

- **tmp_dir**: String of temporary directory used during computation, compulsory parameter
- **model_name**: String of the current model instance. This allows several model instances sharing the same **tmp_dir**, compulsory parameter
- **input_dim**: Int that specifies the input dimension of the data, compulsory parameter
- **output_dim**: Int that specifies the output dimension of the data, compulsory parameter

- **nodal_set**: List of (string/callable) that specify nodal operators, default ['multiplication','exponential','harmonic','quadratic','gaussian','dog']
- **pool_set**: List of (string/callable) that specify pool operators, default ['sum','correlation1','correlation2','maximum']
- **activation_set**: List of (string/callable) that specify activation operators, default ['sigmoid','relu','tanh','soft_linear', 'inverse_absolute', 'exp_linear']
- **metrics**: List of metrics, each metric is either a string indicating the metric function supported by Keras or a callable defined by using Keras/Tensorflow operations, default ['mse',]. See *Custom metrics*
- **special_metrics**: List of special metrics, each metric is a callable that takes 2 numpy arrays (y_true, y_pred). Special metrics are those that requires full batch evaluation such as Precision, Recall or F1, default None. See *Special Metrics*
- **loss**: Loss function, either a string indicating loss function supported by Keras or a callable defined by using Keras/Tensorflow operations, default 'mse'. See *Custom loss*
- **convergence_measure**: String indicates which metric value to monitor the stopping criterion and to gauge the performance when choosing operator sets and weights. **convergence_measure** should also belong to either **metrics** or **special_metrics**, default 'mse'.
- **direction**: String indicates how to compare two measures. 'higher' means the higher the better, 'lower' means the lower the better. This should be set in accordance with **convergence_measure**, e.g., if **convergence_measure** is *mean_square_error* then **direction** should be 'lower' to indicate that lower values of mean_square_error are better, default 'lower'.
- **direct_computation**: True if perform computation on full batch when possible, otherwise computation is done in a mini-batch manner. For large dataset, it is recommended to set False, default False
- **search_computation**: Tuple with 1st element indicating computation devices during the search procedure. 1st element should be either 'cpu' or 'gpu'. If using 'gpu' then 2nd element should be a list(int) of gpu devices, default ('cpu',). See *Computation Environments* for detail description.
- **finetune_computation**: Tuple with 1st element indicating computation devices during the finetune procedure. 1st element should be either 'cpu' or 'gpu'. If using 'gpu' then 2nd element should be a list(int) of gpu devices, default ('cpu',). See *Computation Environments* for detail description.
- **use_bias**: Bool indicates whether to use bias in the weights, default True
- **output_activation**: String indicates optional activation function (supported by Keras) for output layer, default None.
- **input_dropout**: Float indicates dropout percentage applied to input layer during Back Propagation, default None.
- **dropout**: Float indicates dropout percentage applied to hidden layers during Back Propagation in progressive learning step, default None.
- **dropout_finetune**: Float indicates dropout percentage applied to hidden layers during Back Propagation in finetuning step, default None.
- **weight_regularizer**: Float weight decay coefficient used during Back Propagation in progressive learning step, default None
- **weight_regularizer_finetune**: Float weight decay coefficient used during Back Propagation in finetuning step, default None
- **weight_constraint**: Float max-norm constraint value used during Back Propagation in progressive learning step, default None

- **weight_constraint_finetune**: Float max-norm constraint value used during Back Propagation in finetuning step, default None
- **optimizer**: String indicates the name of the optimizers implemented by Keras, default 'adam'
- **optimizer_parameters**: A dictionary to supply non-default parameters for the optimizer, default to None which means using default parameters of Keras optimizer
- **lr_train**: List of learning rates values in a schedule, default [0.01, 0.001, 0.0001]
- **epoch_train**: List of number of epochs for each learning rate value in **lr_train**, default [2,2,2]
- **lr_finetune**: List of learning rates values in a schedule, default [0.0005,]
- **epoch_finetune**: List of number of epochs for each learning rate value in **lr_train**, default [2,]
- **cluster**: Bool indicates if using SLURM cluster to compute. See *Cluster Computation* for details using computation on a cluster.
- **class_weight**: Dict containing the weights given to each class in the loss function, default None. This allows weighing loss values from different classes

Refer *Customization* when custom loss, custom metrics or operators

Below describes common interface implemented by all models.

2.5.2 fit

```
fit(params, train_func, train_data, val_func=None, val_data=None, test_func=None,␣
↪test_data=None, verbose=False)
```

Fits the model with the given parameters and data, this function perform *progressive_learn* to learn the network architecture and *finetune* to finetune the whole architecture. *Note that when validation data is available, the model weights selection and convergence criterion is measured on validation data, otherwise on train data*

Arguments:

- **params**: Dictionary of model parameters. Consult above section *Common parameters* and *Algorithms* for details of each model
- **train_func**: Callable that produces train data generator and the number of mini-batches. See *Data Feeding Mechanism*
- **train_data**: Input to **train_func** See *Data Feeding Mechanism*
- **val_func**: Callable that produces validation data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*
- **val_data**: Input to **val_func**, default None. See *Data Feeding Mechanism*
- **test_func**: Callable that produces test data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*
- **test_data**: Input to **test_func**, default None. See *Data Feeding Mechanism*
- **verbose**: Bool to indicate verbosity or not, default False.

Returns:

- **performance**: Dictionary that holds best performances with keys are loss, metrics and special metrics defined in **params**
- **p_history**: List of full history during progressive learning, with **p_history** [layer_idx][block_idx] is a dictionary similar to **performance**

- **f_history**: Dictionary of full history during finetuning

2.5.3 progressive_learn

```
progressive_learn(params, train_func, train_data, val_func=None, val_data=None, test_
↪func=None, test_data=None, verbose=False)
```

Progressively learn the network architecture according to specific algorithm specified by each model. *Note that when validation data is available, the model weights selection and convergence criterion is measured on validation data, otherwise on train data*

Arguments:

- **params**: Dictionary of model parameters. Consult above section *Common parameters* and *Algorithms* for details of each model
- **train_func**: Callable that produces train data generator and the number of mini-batches. See *Data Feeding Mechanism*
- **train_data**: Input to **train_func** See *Data Feeding Mechanism*
- **val_func**: Callable that produces validation data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*
- **val_data**: Input to **val_func**, default None. See *Data Feeding Mechanism*
- **test_func**: Callable that produces test data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*
- **test_data**: Input to **test_func**, default None. See *Data Feeding Mechanism*
- **verbose**: Bool to indicate verbosity or not, default False.

Returns:

- **history**: List of full history during progressive learning, with **history** [layer_idx][block_idx] is a dictionary with keys are loss, metrics and special metrics defined in **params**

2.5.4 finetune

```
finetune(params, train_func, train_data, val_func=None, val_data=None, test_func=None,
↪ test_data=None, verbose=False)
```

Finetune the whole network architecture, this required a trained model data exists either by calling *load()* or *fit()* or *progressive_learn()*. *Note that when validation data is available, the model weights selection and convergence criterion is measured on validation data, otherwise on train data*

Arguments:

- **params**: Dictionary of model parameters. Consult above section *Common parameters* and *Algorithms* for details of each model
- **train_func**: Callable that produces train data generator and the number of mini-batches. See *Data Feeding Mechanism*
- **train_data**: Input to **train_func** See *Data Feeding Mechanism*
- **val_func**: Callable that produces validation data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*

- **val_data**: Input to **val_func**, default None. See *Data Feeding Mechanism*
- **test_func**: Callable that produces test data generator and the number of mibi-batches, default None. See *Data Feeding Mechanism*
- **test_data**: Input to **test_func**, default None. See *Data Feeding Mechanism*
- **verbose**: Bool to indicate verbosity or not, default False.

Returns:

- **history**: List of full history during progressive learning, with **history** [layer_idx][block_idx] is a dictionary with keys are loss, metrics and special metrics defined in **params**
- **performance**: Dictionary of best performances with keys are loss, metrics and special metrics defined in **params**

2.5.5 evaluate

```
evaluate(data_func, data_argument, metrics, special_metrics=None, computation=('cpu',
↪))
```

Evaluate the model with given data and metrics

Arguments:

- **data_func**: Callable that produces data generator and the number of mini-batches
- **data_argument**: Input to **data_func**
- **metrics**: List of metrics, with each metric can be computed through aggregation of evaluation on mini-batches, e.g., accuracy, mse
- **special_metrics**: List of special metrics, which can only be computed over full batch, e.g., f1, precision or recall
- **computation**: Tuple with 1st element is a string to indicate 'cpu' or 'gpu'. In case of 'gpu', 2nd element is a list of int which specifies gpu devices

Returns:

- **performance**: Dictionary of performances with keys are the metric names in **metrics** and **special_metrics**

2.5.6 predict

```
predict(data_func, data_argument, computation=('cpu',))
```

Using current model instance to generate prediction

Arguments:

- **data_func**: Callable that produces data generator and the number of mini-batches
- **data_argument**: Input to **data_func**
- **computation**: Tuple with 1st element is a string to indicate 'cpu' or 'gpu'. In case of 'gpu', 2nd element is a list of int which specifies gpu devices

Returns:

- **pred**: Numpy array of prediction

2.5.7 save

```
save(filename)
```

Save the current model instance to disk

Arguments:

- **filename**: String that specifies the name of pickled file

Returns:

2.5.8 load

```
load(filename)
```

Load a pretrained model instance from disk

Arguments:

- **filename**: String that specifies the name of pickled file

Returns:

2.5.9 get_default_paramters

```
get_default_parameters()
```

Get the default parameters of the model

Arguments:

Returns:

- **params**: Dictionary of default parameters

2.6 Computation Environments

computation is specified through a tuple. The first element is a string, either 'cpu' or 'gpu', which indicates computation devices.

- If 'gpu' is specified, the 2nd element of the tuple must be a list of int indicating the GPU device numbers.
- If 'cpu' is specified, the 2nd element should be an integer indicating the number of parallel processes to run during operator set search (ideally it should be the number of cores on the local machine). This number, however, has no effect when doing finetuning since a single process will use the available cores.

computation can be specified in the following cases:

- In **params** (dictionary) (see *Common parameters*) argument that is given to *fit*, *finetune*:
  - *search_computation*: this key specifies the computation device during the operator set search procedure.
  - *finetune_computation*: this key specifies the computation device during the finetuning step.
- *computation* is also an argument to *evaluate*, *predict*

Note 1: depending on the computation setup available, 'search_computation' must be carefully set. If only tensorflow-gpu is installed on the system but 'search_computation' is set to 'cpu' with K processes, the library will attempt to launch K different processes trying to use all GPUs without explicit device allocation, which can lead to out of memory situation

2.6.1 Local CPU

This is the default computation option in all models. That is:

```
params['search_computation'] = ('cpu', 8)
params['finetune_computation'] = ('cpu',)
evaluate(..., computation=('cpu',))
predict(..., computation=('cpu',))
```

2.6.2 Local GPU

Sometimes it is desirable to mix CPU and GPU computation when both versions of tensorflow are available. For example, in HeMLGOP or its variants (HoMLGOP, HeMLRN, HoMLRN), the search procedure solves a least square problem with computation implemented on CPU. Thus it might be desirable to use CPU during the search procedure to avoid data transfer between CPU and GPU and use GPU during the finetuning step. This can be done by setting the *computation* as follows:

```
params['search_computation'] = ('cpu', K) # with K is the number of cores on the␣
↪system
params['finetune_computation'] = ('gpu', [0,1]) # finetuning with 2 GPUs
```

However, with algorithms relying only on Back Propagation during the search procedure such as POP, POPfast or POPmemO and POPmemH, it is desirable to use GPUs to perform the operator set search procedure:

```
# using 4 GPUs to perform operator set searching
# the implementation launches 4 processes, each of which sees only 1 GPU device
params['search_computation'] = ('gpu', [0,1,2,3])
```

2.6.3 Cluster Computation

The library also supports running the operator set searching procedure on a SLURM cluster. Since the search procedure involves evaluating each operator set independently, which is highly parallelizable. Thus, on a cluster running SLURM, the user can instruct the library to run the search procedure on many machines by submitting new batch jobs. Note that this assumes the all nodes share the same disk system since they will try to access *tmp_dir/model_name* as specified in **params** (see *Common parameters*)

To allow the search procedure to submit new batch jobs, set

```
params['cluster'] = True
```

In addition, a dictionary called **batchjob_parameters** that contains the configuration of the batch job file must be given to **params** as:

```
params['batchjob_parameters'] = batchjob_parameters
```

batchjob_parameters must have the following (key, value):

- **name**: String that specifies the name of the job. This corresponds to option *#SBATCH -j*

- **mem**: Int that specifies the amount of memory (or RAM) requested for each node in GB. This corresponds to option *#SBATCH –mem*
- **core**: Int that specifies the number of cores requested for each node. This corresponds to option *#SBATCH -c*
- **partition**: String that specifies the name of the partition to request the nodes. This corresponds to option *#SBATCH -p*
- **time**: String that specifies the maximum allowed time for each node, e.g., '2-00:00:00' indicates 2 days. This corresponds to option *#SBATCH -t*
- **no_machine**: Int that specifies the number of parallel nodes requested. e.g., if no_machine=4 and there are 72 operator sets, each node will process 18 different operator sets. This corresponds to option *#SBATCH -array*
- **python_cmd**: String that specifies the command how to run python on bash. In many cases, it is simply just 'python' if python is in the $PATH. In some systems, this involves calling 'srun python'

In addition, two optional keys can be set to allow specific configurations:

- **constraint**: String that specifies the constraint on the node, e.g. 'hsw' might indicate only request for Haswell architectures or 'gpu' only request for GPU nodes. This corresponds to option *#SBATCH –constraint*
- **configuration**: String that specifies all the necessary setup befores launching a python script on a node, e.g., this can be the setup of $PATH or module load, etc.

Note that similar to the local case, both CPU and GPU can be used during the search procedure using the cluster. However, **batchjob_parameters** *must be carefully set in accordance with all computation parameters setup*

- *If* **params['search_computation']** *indicate CPU*, **batchjob_parameters** *must be set so that the requested nodes and its configuration allow running tensorflow cpu version.*
- *If* **params['search_computation']** *indicate GPU*, **batchjob_parameters** *must be set so that the requested nodes allow the access to the specified GPU device list and tensorflow-gpu can be invoked*

In addition, the main script, which creates a model instance and operates on the model instance, is usually run on a node on the cluster, so ensure that **params['finetune_computation']** *and other computation arguments used in evaluate(), predict() are set in accordance with the node configuration itself*

2.7 Algorithms

This section briefly describes each algorithm implemented in the library as well as detail description of algorithm-specific parameters.

2.7.1 Progressive Operational Perceptron (POP)

Description

POP defines a maximum network template, which specifies the maximum number of layers and the size of each layer. The algorithm then learns one hidden layer at a time. All GOP neurons in the same layer share the same operator set and all layers are GOP layers, including the output layer. When learning a new hidden layer, all previous layers that have been learned are fixed. POP uses a two-pass Greedy Iterative Search (GIS) algorithm:

- Randomly select operator set (nodal, pool, activation) for the new hidden layer, let say *h*. For every operator set *op* in the library, assign *op* to the output layer and train the network (new hidden layer, output layer) for E epochs. Select the best performing operator set *o-1* according to **params['convergence_measure']** for the output layer.

- Fix *o-1* for the output layer, for every operator set *op* in the library, assign *op* to the new hidden layer and train the network for E epochs. Select the best operator set *h-1* for the hidden layer.
- Repeat step 1, however with *h-1* as the operator set for the hidden layer. This produces *o-2* for the output layer.
- Repeat step 2, however with *o-2* as the operator set for the output layer. This produces *h-2*.

GIS outputs *h-2* and *o-2* as the operator set for the hidden and output layer with current synaptic weights. The algorithm continues adding new hidden layer when the performance stops improving, given a threshold. After learning the architecture of the network (number of layer and operator set assignments), POP finetunes the synaptic weights of the whole architecture using Back Propagation.

Parameters

The following parameters are specific to POP that should be set in **params** dictionary:

- **max_topology**: List of ints that specifies the maximum network topology, default [40,40,40,40], which means 4 hidden layers with 40 GOPs each
- **layer_threshold**: Float that specifies the threshold on the relative improvement, default 1e-4. See equation (8) in here for more details.

2.7.2 Heterogeneous Multilayer Operational Perceptron (HeMLGOP)

Description

HeMLGOP learns a heterogeneous layers of GOPs in a block-wise manner. At each step, the algorithm adds a new block to the current hidden layer by searching for the suitable operator set and its synaptic weights. When the performance saturates in the current hidden layer, HeMLGOP constructs a new hidden layer composed of one block. The progression in the new hidden layer continues until reaching saturation. HeMLGOP then evaluates if adding this new hidden layer improves the performance. The algorithm terminates when adding new hidden layer stops improving the performance, given a threshold.

Different than POP, HeMLGOP assumes linear output layer. To evaluate an operator set assignment to the new block, HeMLGOP assigns random weights to the new block and optimizes the weights of the output layer by solving a least-square problem. After selecting the best operator set for the new block, HeMLGOP performs the weight update of the new block and output layer through Back Propagation. Once a block is learned, it is fixed (operator set assignment and weights). Similar to POP, once the network architecture is learned in the progressive learning step, HeMLGOP finetunes all the weights in the network.

Parameters

The following parameters are specific to HeMLGOP that should be set in **params** dictionary:

- **block_size**: Int that specifies the number of neurons in a new block, default 20
- **max_block**: Int that specifies the maximum number of blocks in a hidden layer, default 5
- **max_layer**: Int that specifies the maximum number of layers, default 4
- **block_threshold**: Float that specifies the threshold on the relative performance improvement when adding new block, default 1e-4. See equation (7) in here for more details.
- **layer_threshold**: Float that specifies the threshold on the relative performance improvement when evaluating new hidden layer, default 1e-4. See equation (8) in here for more details.
- **least_square_regularizer**: Float that specifies the coefficient of regulariztion when solving least square problem, default 0.1

2.7.3 Homogeneous Multilayer Operational Perceptron (HoMLGOP)

Description

HoMLGOP is a variant of HeMLGOP with the difference that all blocks in the same layer share the same operator set. That means the operator set searching procedure is only made when adding the 1st block of a new hidden layer. 2nd, 3rd... blocks make the same operator set assignment and only update the block weights. There is also the weights finetuning step in HoMLGOP when the architecture is defined after the progressive learning step.

2.7.4 Heterogeneous Multilayer Randomized Network (HeMLRN)

Description

HeMLRN is also a variant of HeMLGOP with the difference is that there is no intermediate weight update steps through Back Propagation in HeMLRN. The algorithm adds a new block by using random weights and only optimize the output layer weights. After that, the random weight of the new block is fixed and another new block is learned by evaluating all operator set assignment in the same manner. All weights finetuning is also done after the network architecture is defined.

2.7.5 Homogeneous Multilayer Randomized Network (HoMLRN)

Description

HoMLRN is also a variant of the above three algorithms. HoMLRN is similar to HoMLGOP in that all blocks in the same layer share the same operator set assignment. Different from HoMLGOP, HoMLRN has no intermediate weight update steps by Back Propagation. So the 2nd, 3rd... blocks of all hidden layers are assigned random weights and at each step, only the weights of output layer is optimized by solving least square problem and the performance is recorded for the new block. Whole network finetuning is also done in HoMLRN after the network growth stops.

Note that since HoMLGOP, HeMLRN, HoMLRN are variants of HeMLGOP, they share the same parameters as HeMLGOP described above

2.7.6 Fast Progressive Operational Perceptron (POPfast)

Description

POPfast is a fast and simplified version of POP. When adding a new hidden layer, POP has to search for the operator sets of both the hidden and output layer, which involves a large search space. POPfast simply assumes a linear output layer, i.e. *multiplcation* as the nodal operator and *summation* as the pooling operator. This constraint reduces the search problem to only the new hidden layer. The progression in POPfast is similar to POP, that is the network is grown layer-wise with a predefined maximum topology. Parameters that are specific to POP are also applied to POPfast.

2.7.7 Progressive Operational Perceptron with Memory (POPmem)

Description

POPmem uses a similar search procedure as in POPfast with the assumption of a linear output layer. The idea of POPmem is to augment the network growing procedure by enhancing the representation in the network. POPmem aims to address the following problem:

> *When adding a new hidden layer, POP or POPfast aims to learn a better transformation of the data by only using the output of the previous transformation (the current hidden layer), and using this (potentially better) transformation to learn a decision function (the output layer). Thus, the new hidden layer has no direct access to previously extracted hidden representations, and the output layer also has no direct access to these information*

There are two memory schemes which are denoted as POPmemH and POPmemO that was proposed to address the above problem:

- In POPmemH, before adding a new hidden layer, the previous hidden representation is linearly projected to a meaningful subspace such as PCA or LDA and concatenated to the current hidden representation as input to the new hidden layer.
- In POPmemO, when adding a new hidden layer, the *current* hidden representation is linearly projected to a meaningful subspace such as PCA or LDA. This compact representation is concanated with the new hidden layer to form an enhanced hidden representation, which is connected to the output layer.

The motivation and discussion of two memory schemes are discussed in details in here. Generally, POPmem can be understood as positing the layer addition as: given all the previously extracted hidden representations, find a new hidden layer *and* the output layer configuration that improves the performance.

Parameters

The following parameters are specific to POPmem that should be set in **params** dictionary:

- **max_topology**: List of ints that specifies the maximum network topology, default [40,40,40,40], which means 4 hidden layers with 40 GOPs each
- **layer_threshold**: Float that specifies the threshold on the relative improvement, default 1e-4. See equation (8) in *here <https://arxiv.org/pdf/1804.05093.pdf>* for more details.
- **memory_type**: String that specifies the type of linear projection, either 'PCA' or 'LDA', default 'PCA'. Note that 'LDA' should be used in classification problem only. The dimension of the subspace in PCA is chosen so that at least 98% of the energy of the data is preserved. For 'LDA', the subspace dimension is 'output_dim'-1.
- **memory_regularizer**: Float that specifies the regularization coefficient when calculating the projection, default 0.1

2.8 Customization

This section describes how to customize PyGOP to your need!

2.8.1 Custom loss

The loss function is specified through **params['loss']** (see *Common parameters*), which is 'mse' (mean square error) by default. All the loss functions defined by Keras is supported in PyGOP. These includes:

- *'mean_squared_error'*
- *'mean_absolute_error'*
- *'mean_absolute_percentage_error'*
- *'mean_squared_logarithmic_error'*
- *'squared_hinge'*
- *'hinge'*

- *'categorical_hinge'*
- *'logcosh'*
- *'categorical_crossentropy'*
- *'sparse_categorical_crossentropy'*
- *'binary_crossentropy'*
- *'kullback_leibler_divergence'*
- *'poisson'*
- *'cosine_proximity'*

The above strings can be set to **params['loss']** to indicate the loss function. User can also define the definition of the loss function and assign the callable to **params['loss']**. A custom loss function should follow this template

```
def cutom_loss(y_true, y_pred):
    # calculation of the loss using tensorflow or keras backend operations

    return loss # loss should be a scalar
```

Note that the computation in the loss function must be expressed by tensorflow or keras operations. Below gives an example of a custom mean squared error that only calculate the error if two corresponding elements have the same sign

```
"""An example of custom loss function for PyGOP models
"""
import tensorflow as tf

def custom_mse(y_true, y_pred):
# assume 1st dimension is the number of samples
    mask = y_true * y_pred > 0
    mse = tf.reduce_sum(tf.flatten(mask * (y_true - y_pred)**2))

    return mse


# set the custom loss to the dictionary of model parameters
params['loss'] = custom_mse
```

2.8.2 Custom metrics

This refers to **params['metrics']**, which are standard metrics that can be accumulated over mini-batches. Similar to loss function, we can also define a list of metrics that has both built-in metrics and custom metrics. Note that all the loss strings listed above can also be used as metrics. In addition, Keras has the following built-in metrics:

- *'binary_accuracy'*
- *'categorical_accuracy'*
- *'sparse_categorical_accuracy'*
- *'top_k_categorical_accuracy'*
- *'sparse_top_k_categorical_accuracy'*

The custom metric can be defined in exactly the same manner as custom loss. Suppose we need to monitor 'accuracy', mean_squared_error and 'custom_mse' defined above, we can set metrics as follows:

```
"""An example that defines custom metric for PyGOP models
"""
import tensorflow as tf

def custom_mse(y_true, y_pred):
# assume 1st dimension is the number of samples
    mask = y_true * y_pred > 0
    mse = tf.reduce_sum(tf.flatten(mask * (y_true - y_pred)**2))

    return mse

# set the metrics to the dictionary of model parameters
params['metrics'] = ['accuracy', 'mean_squared_error', custom_mse]
```

2.8.3 Special Metrics

This refers to **params['special_metrics']**, which categorizes those metrics that require y_true and y_pred of the full batch to evaluate. Special metrics are given as a list of user-defined functions, which use should take **numpy arrays** as input to compute the metrics. Examples of special metrics are precision, recall or f1. Below gives an example that defines average precision, average recall and average f1 as special metrics:

```
"""An example that defines average precision, recall and f1 using sklearn metrics
   and use this metrics as special metrics in PyGOP
"""
from sklearn import metrics

def custom_precision(y_true, y_pred):
# assume 1st dimension is the number of samples
    y_true_lb = np.argmax(y_true, axis=-1)
    y_pred_lb = np.argmax(y_pred, axis=-1)

    return metrics.f1_score(y_true_lb, y_pred_lb, average='macro')

def custom_recall(y_true, y_pred):
# assume 1st dimension is the number of samples
    y_true_lb = np.argmax(y_true, axis=-1)
    y_pred_lb = np.argmax(y_pred, axis=-1)

    return metrics.f1_score(y_true_lb, y_pred_lb, average='macro')

def custom_f1(y_true, y_pred):
# assume 1st dimension is the number of samples
    y_true_lb = np.argmax(y_true, axis=-1)
    y_pred_lb = np.argmax(y_pred, axis=-1)

    return metrics.f1_score(y_true_lb, y_pred_lb, average='macro')

# set special metrics to the dictionary of model paramters
params['special_metrics'] = [custom_precision, custom_recall, custom_f1]
```

If params['convergence_measure'] is one of the custom metrics or special metrics, it should be specified as the name of the function, e.g. params['convergence_measure'] = 'custom_f1'

2.8.4 Custom Operators

While PyGOP specifies built-in library of operators as defined in Table 1 in here, custom operators can be defined by users and given to train a model. All custom operators must be implemented using tensorflow or keras operators. Below gives the templates on how to define custom nodal, pooling or activation operators that can be used by PyGOP

```
"""Template of custom operators
"""

def custom_nodal(x, w):
"""Description of custom nodal operator format

All nodal operators must take as input two tensors x and w, which are the input and␣
↪weights
x and w are assumed to have compatible shape so that the element-wise multiplication␣
↪(x*w) is valid

All nodal operations should be element-wise operation, meaning that each individual␣
↪input signal x[i] is
modified by the corresponding weight element w[i]

Here we give as an example the 'multiplication' operator

"""

return x*w


def custom_pool(z):
"""Description of the pooling operator format

All pooling operators must take only one input z, which is the output of the nodal␣
↪operation
z has specific shape of [N, D, d]
- N is the number of samples
- D is the number of neurons in the previous layer (number of input signals)
- d is the number of neurons in the current layer

The pooling operation performs pooling over D input signals, thus the pooling is␣
↪performed on axis=1
The output y of the pooling operator should has shape [N, d]

Here we give as an example of the 'sum' operator

"""

y = tf.reduce_sum(z, axis=1)
return y

def custom_activation(y):
"""Description of the activation operator format

All activation operators should be element-wise operation.
Here we give as an example the 'sigmoid' operator
"""

return tf.sigmoid(y)
```

After defining the custom operators, these functions can be included in the list of **params['nodal_set']**, **params['pool_set']**, **params['activation_set']** together with/without other built-in operators. See *Common parameters* for the default built-in values

2.9 Change Log

All notable changes to PyGOP will be documented here.

2.9.1 [0.2.2] 2019-04-24

Added

- Test Suite: unit test, PEP8 verification
- Continuous Integration (Travis)
- Mnist Example
- Instructions on Contributions
- Improved Documentation
- requirements.txt
- Functionality to specify optimizer's parameters

Fixed

- learning rate update in utility/gop_utils.network_trainer()
- improve coding style according to PEP8

2.9.2 [0.2.1] 2018-12-29

Added

- new wheel for version 0.2.1 in releases

Fixed

- move the tmp files removal step from progressive_learn to the end of fit() in all models (hemlgop.py, homlgop.py, hemlrn.py homlrn.py, pop.py, popfast.py, popmemo.py, popmemh.py). This fixes the bug that removes train_states.pickle before finetuning.
- change file opening option from 'r' to 'rb' in utility/misc.initialize_states() to read train_states.pickle in Python3

2.9.3 [0.2.0] 2018-12-17

Added

- utility/block_update.py

- utility/calculate_memory.py
- Added CHANGELOG.md to keep track of major changes
- Added releases directory to keep track of different wheel versions.
- Functionalities to spawn new process when calculating memory block (utility/gop_utils.calculate_memory_block_standalone() and utility/calculate_memory.py). This prevents potential OOM errors when tensorflow-gpu does not release memory right after the block finishes.
- Functionalities to spawn new process when finetuning some blocks (utility/gop_utils.block_update_standalone() and utility/block_update.py). This also prevents potential OOM errors mentioned above.

Fixed

- utility/misc.check_model_parameters()
- models/_model.print_performance()

2.10 Contributions

PyGOP is open for contributions. Especially if you are developing GOP-based algorithms and would like to catch public attention or integrate your algorithm to our codebase to enable reproducible research, you are welcome! In addition, new features, documentations, tests or bug reports are also welcome. Following is the guide to contribute to PyGOP:

2.10.1 Bug Report

If you have determined that your code doesn't work and it comes from PyGOP, please follow the following step to report a bug.

1. Make sure your code is up-to-date with our current code on the master branch.
2. Make sure to check the issue section before filing an issue on our github project. The bug might already be reported.
3. Create an issue in issue section . Give detailed description of your system configuration: What OS you are using? What is the python version? What is the PyGOP version? What is your computation device (cpu/gpu)? If you are using GPU, what is the model of GPU and the version of CUDA? In addition, make sure to include a complete example which we can use to trigger the bug.
4. If you know how to fix the bug, please make a request to contribute!

2.10.2 Pull Request

If you want to contribute to our codebase, please send an email to viebboy@gmail.com to with subject "PYGOP DEVELOPMENT" to discuss your plan which should include the description of the contribution (bug fix, interface improvement, new feature, new test or documentation) and potential interface (for code contribution). Once we reach the agreement on how to proceed, you can

1. Fork PyGOP on github
2. Clone your fork to local machine
3. Inside the top level project directory, install the project for development purpose:

```
pip install -e .
```

4. Make a new branch for development
5. Hook your Travis CI to your development branch to enable continuous testing
6. Write code in the development branch
7. Write unit tests in tests/ and run pytest to test your code locally
8. Write documentation in docs/
9. Make a Pull Request here

2.11 License