PyXAB

Release 0.3.0

Wenjie Li

Aug 15, 2023

GETTING STARTED

PyXAB is a Python open-source library for X-armed bandit algorithms, a prestigious set of optimizers for online black-box optimization and hyperparameter optimization.

PyXAB includes implementations of different algorithms for X-armed bandit, such as Zooming, StoSOO, and HCT, and the most recent works such as GPO and VHCT. PyXAB also provides the most commonly-used synthetic objectives to evaluate the performance of different algorithms and the implementations for different hierarchical partitions

PyXAB is featured for:

- **User-friendly APIs, clear documentation, and detailed examples**
- **Comprehensive library** of optimization algorithms, partitions and synthetic objectives
- **High standard code quality and high testing coverage**
- **Low dependency** for flexible combination with other packages such as PyTorch, Scikit-Learn

Reminder: The algorithms are maximization algorithms!

CHAPTER

ONE

QUICK EXAMPLE

PyXAB follows a natural and straightforward API design completely aligned with the online blackbox optimization paradigm. The following is a simple 6-line usage example.

First, we define the parameter domain and the algorithm to run. At every round `t`, call `algo.pull(t)` to get a point and call `algo.receive_reward(t, reward)` to give the algorithm the objective evaluation (reward)

```
domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
algo = T_HOO(rounds=1000, domain=domain)
for t in range(1000):
    point = algo.pull(t)
    reward = 1                  #TODO: User-defined objective returns the reward
    algo.receive_reward(t, reward)
```

CHAPTER

TWO

CITATIONS

If you use our package in your research or projects, we kindly ask you to cite our work

```
@misc{Li2023PyXAB,
    doi = {10.48550/ARXIV.2303.04030},
    url = {https://arxiv.org/abs/2303.04030},
    author = {Li, Wenjie and Li, Haoze and Honorio, Jean and Song, Qifan},
    title = {PyXAB -- A Python Library for $\mathcal{X}$-Armed Bandit and Online␣
↪Blackbox Optimization Algorithms},
    publisher = {arXiv},
    year = {2023},
 }
```

We would appreciate it if you could cite our related works.

```
@article{li2023optimumstatistical,
    title={Optimum-statistical Collaboration Towards General and Efficient Black-box␣
↪Optimization},
    author={Wenjie Li and Chi-Hua Wang and Guang Cheng and Qifan Song},
    journal={Transactions on Machine Learning Research},
    issn={2835-8856},
    year={2023},
    url={https://openreview.net/forum?id=ClIcmwdlxn},
    note={}
}
```

```
@misc{li2022Federated,
    doi = {10.48550/ARXIV.2205.15268},
    url = {https://arxiv.org/abs/2205.15268},
    author = {Li, Wenjie and Song, Qifan and Honorio, Jean and Lin, Guang},
    title = {Federated X-Armed Bandit},
    publisher = {arXiv},
    year = {2022},
}
```

2.1 Installation

Note: The PyXAB package is under active development. Please make sure that you install the latest version of our package to enjoy all the features. At this moment, we recommend installing the package using pip.

2.1.1 PyPI Installation

To install the package using PyPI, run the following lines of code

```
pip install PyXAB                # normal install
pip install --upgrade PyXAB      # or update if needed
```

2.1.2 Git Installation

To install the package using git, run the following lines of code

```
git clone https://github.com/WilliamLwj/PyXAB.git
cd PyXAB
pip install .
```

2.1.3 Required Dependencies

- Python 3.6+
- numpy>=1.20.3
- matplotlib
- scikit-learn>=0.24.2 (if running real-life examples)

2.2 General Instructions

To use PyXAB, simply follow the instructions below. The domain and the algorithm must be defined beforehand. Hierarchical Partition is optional and normally binary partition works well. The objective must be able to evaluate each point the algorithm pulls and return the evaluated objective value.

2.2.1 Domain

The domain needs to be written in list of lists for a continuous domain. For example, if the parameter range is [0.01, 1], then the domain should be written as

```
domain = [[0.01, 1]]
```

If the parameter has two dimensions, say [-1, 1] x [2, 10], then the domain should be written as

```
domain = [[-1, 1], [2, 10]]
```

2.2.2 (Optional) Partition

The hierarchical partition is a core part of many X-armed bandit algorithms. It discretizes the infinite parameter space into finite number of arms in each layer hierarchically, so that finite-armed bandit algorithm designs can be utilized.

However, the design of the partition is completely optional and unnecessary in the experiments. PyXAB provides many designs in the package for the users to choose from, e.g., a standard binary partition would be

```
from PyXAB.partition.BinaryPartition import BinaryPartition
partition = BinaryPartition
```

By default, the standard binary partition will be used for all the algorithms if unspecified.

2.2.3 (User Defined) Objective

Note: The objective function `f` should be bounded by -1 and 1 for the best performance of most algorithms, i.e., `-1 <= f(x) <= 1`

Note: It is unnecessary to define the objective function in the following way, but for consistency we recommend doing so. As long as the objective function can return a reward to each point pulled by the algorithm, then the optimization process could run.

The objective function has an attribute `fmax`, which is the maximum reward obtainable. Besides, the objective function should have a function `f(x)`, which will return the reward of the point `x`. See the following simple example for a better illustration.

```
from PyXAB.synthetic_obj.Objective import Objective
import numpy as np

# The sine function f(x) = sin(x)
class Sine(Objective):
    def __init__(self):
        self.fmax = 1

    def f(self, x):
```

(continues on next page)

(continued from previous page)

```
        x = np.array(x)
        return np.sin(x)
```

2.2.4 Algorithm

Note: The point returned by the algorithm will be a list. Make sure your objective can deal with this data type. For example, if it wants the objective value at the point x = 0.8, it will return [0.8]. If the algorithm wants the objective value at x = (0, 0.5), the algorithm will return [0, 0.5].

Algorithms will always have one function named `pull` that outputs a point for evaluation, and the other function named `receive_reward` to get the feedback. Therefore, in the online learning process, the following lines of code should be used.

```
from PyXAB.algos.HOO import T_HOO
T = 1000
algo = T_HOO(rounds=T, domain=domain, partition=partition)
target = Sine()

# either for-loop or while-loop
for t in range(1, T+1):
    point = algo.pull(t)
    reward = target.f(point) + np.random.uniform(-0.1, 0.1)
    algo.receive_reward(t, reward)
```

Note: If the objective function is not defined by inheriting the *PyXAB.synthetic_obj.Objective.Objective* class, simply change the second last line in the above snippet to the evaluation of the objective.

2.3 Usage Examples

Below are examples of using the PyXAB package

2.3.1 1-D Example

In this example, we run the T_HOO algorithm on the Garland objective. First, import all the functions needed

```
from PyXAB.synthetic_obj.Garland import Garland                # the objective
from PyXAB.algos.HOO import T_HOO                              # the algorithm
from PyXAB.partition.BinaryPartition import BinaryPartition    # the partition

# the other useful packages/functions
import numpy as np
from PyXAB.utils.plot import plot_regret
```

Define the number of rounds, the target, the domain, the partition, and the algorithm for the learning process

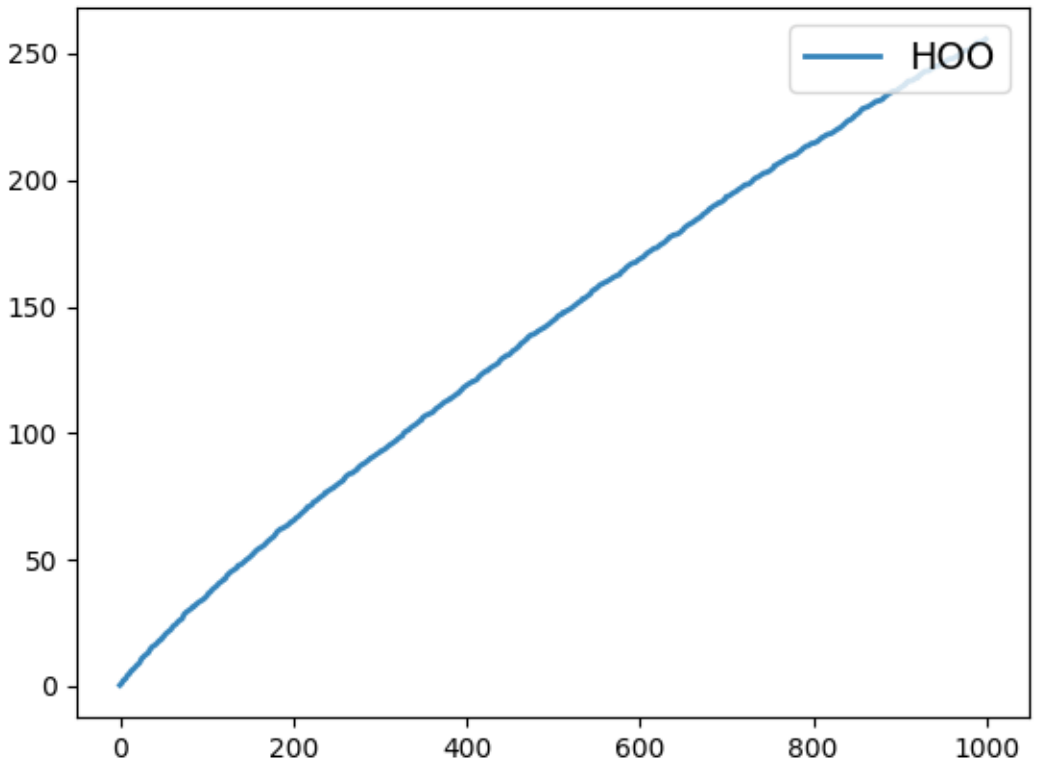
```
T = 1000                                                    # the number of rounds is 1000
target = Garland()                                          # the objective to optimize is␣
↪Garland
domain = [[0, 1]]                                           # the domain is [[0, 1]]
partition = BinaryPartition                                 # the partition chosen is␣
↪BinaryPartition
algo = T_HOO(rounds=1000, domain=domain, partition=partition)    # the algorithm is T_HOO
```

To plot the regret, we can initialize the cumulative regret and the cumulative regret list

```
cumulative_regret = 0
cumulative_regret_list = []
```

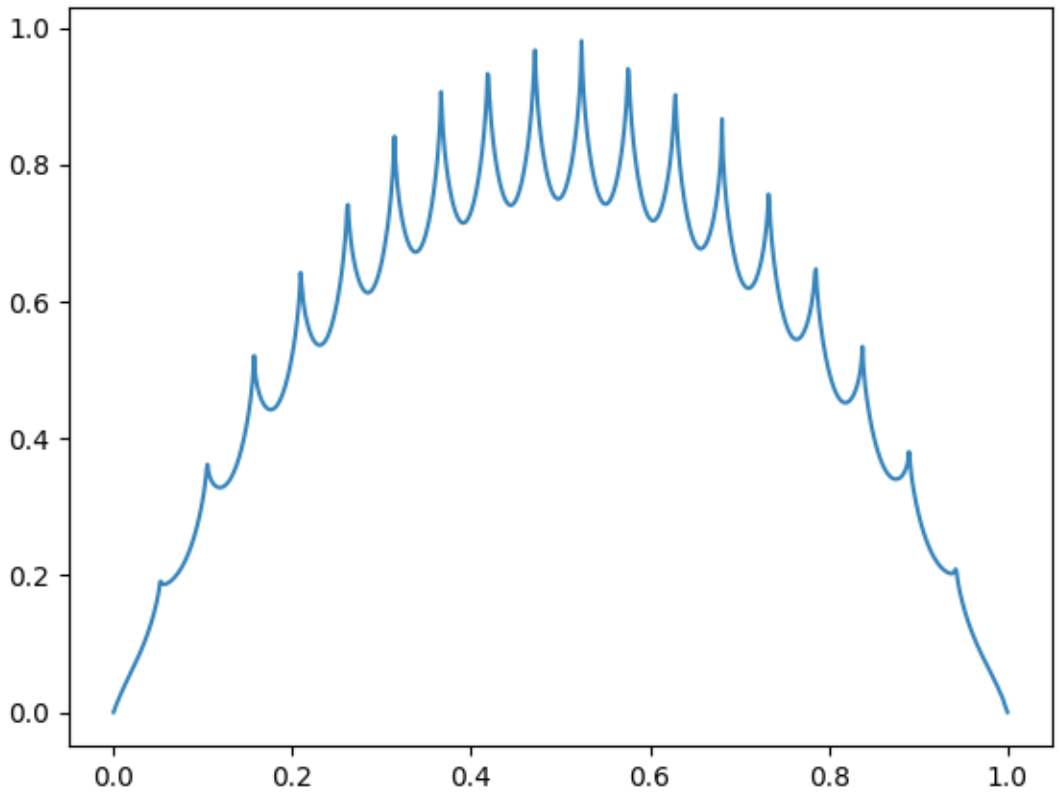
In each iteration of the learning process, the algorithm calls the `pull(t)` function to obtain a point, and then the reward for the point is returned to the algorithm by calling `receive_reward(t, reward)`. For a stochastic learning process, uniform noise is added to the reward.

```
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target.f(point) + np.random.uniform(-0.1, 0.1)     # uniform noise
    algo.receive_reward(t, reward)

    # the following lines are for the regret
    inst_regret = target.fmax - target.f(point)
    cumulative_regret += inst_regret
    cumulative_regret_list.append(cumulative_regret)


# plot the regret
plot_regret(np.array(cumulative_regret_list), name='HOO')
```



The following lines of code are only for creating thumbnails and do not need to be used

```
# sphinx_gallery_thumbnail_number = 2
import matplotlib.pyplot as plt

fig = plt.figure()
ax = fig.add_subplot(111)

x = np.linspace(domain[0][0], domain[0][1], 1000)
z = x * (1-x) * (4 - np.sqrt(np.abs(np.sin(60 * x))))
ax.plot(x, z, alpha=0.9)
fig.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)
```



Total running time of the script: (0 minutes 2.105 seconds)

2.3.2 2-D Example

In this example, we run the HCT algorithm on the normalized Himmelblau objective. First, import all the functions needed

```
from PyXAB.synthetic_obj.Himmelblau import Himmelblau_Normalized        # the objective
from PyXAB.algos.HCT import HCT                                        # the algorithm
from PyXAB.partition.BinaryPartition import BinaryPartition            # the partition

# the other useful packages/functions
import numpy as np
from PyXAB.utils.plot import plot_regret
```

Define the number of rounds, the target, the domain, the partition, and the algorithm for the learning process

```
T = 1000                                                # the number of rounds is␣
↪1000
target = Himmelblau_Normalized()                        # the objective to optimize␣
↪is the normalized Himmelblau
domain = [[-5, 5], [-5, 5]]                             # the domain is [[-5, 5], [-
↪5, 5]]
```

(continues on next page)

(continued from previous page)

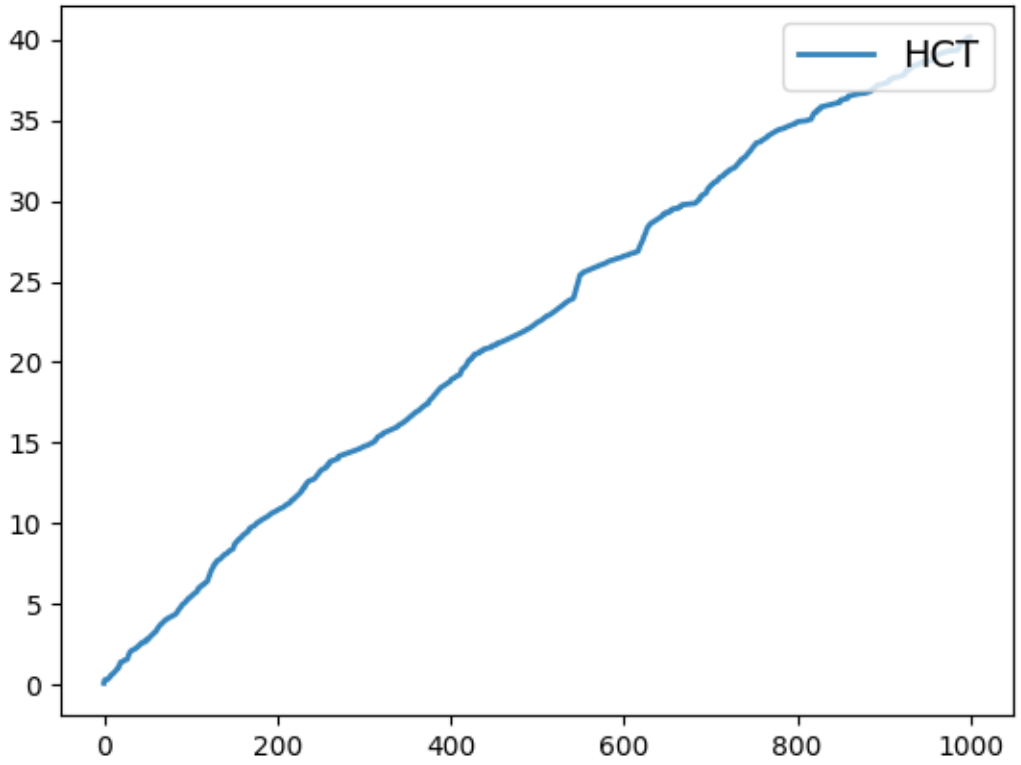
```
partition = BinaryPartition                                    # the partition chosen is␣
↪BinaryPartition
algo = HCT(domain=domain, partition=partition)                 # the algorithm is HCT
```

To plot the regret, we can initialize the cumulative regret and the cumulative regret list

```
cumulative_regret = 0
cumulative_regret_list = []
```

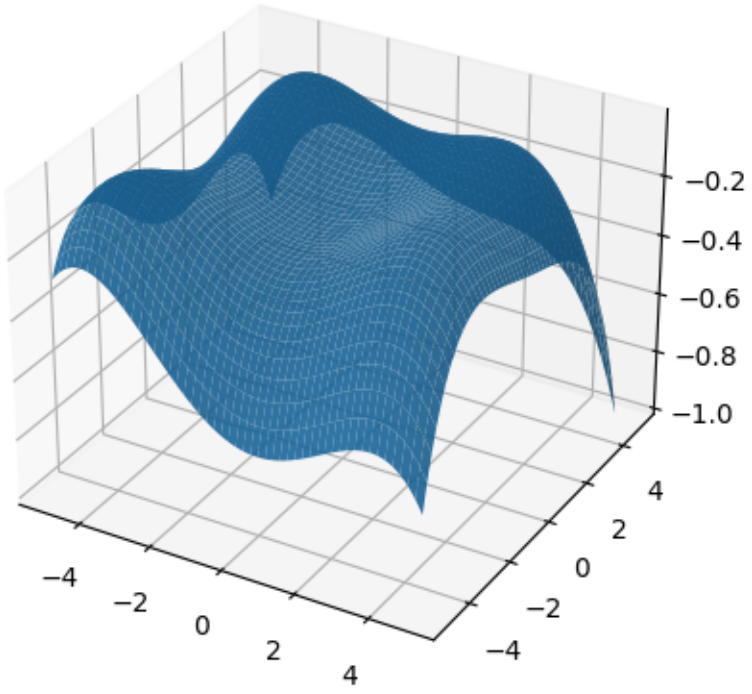
In each iteration of the learning process, the algorithm calls the `pull(t)` function to obtain a point, and then the reward for the point is returned to the algorithm by calling `receive_reward(t, reward)`. For a stochastic learning process, uniform noise is added to the reward.

```
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target.f(point) + np.random.uniform(-0.1, 0.1)          # uniform noise
    algo.receive_reward(t, reward)

    # the following lines are for the regret
    inst_regret = target.fmax - target.f(point)
    cumulative_regret += inst_regret
    cumulative_regret_list.append(cumulative_regret)


# plot the regret
plot_regret(np.array(cumulative_regret_list), name='HCT')
```



The following lines of code are only for creating thumbnails and do not need to be used

```
# sphinx_gallery_thumbnail_number = 2
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.linspace(domain[0][0], domain[0][1], 1000)
y = np.linspace(domain[0][0], domain[0][1], 1000)
xx, yy = np.meshgrid(x, y)
z = (- (xx ** 2 + yy - 11) ** 2 - (xx + yy ** 2 - 7) ** 2) / 890
ax.plot_surface(xx, yy, z, alpha=0.9)
fig.subplots_adjust(left=0.1, right=0.9, top=0.9, bottom=0.1)
```



Total running time of the script: (0 minutes 0.881 seconds)

2.3.3 Real-life Data

In this example, we run the VHCT algorithm to tune hyperparameters for Support Vector Machine (SVM)

```
from PyXAB.algos.VHCT import VHCT
from PyXAB.partition.BinaryPartition import BinaryPartition
from PyXAB.utils.plot import plot_regret

# Useful functions and classes for the learning process
from sklearn import svm
from sklearn.metrics import roc_auc_score
import numpy as np
import pickle
```

We first define the objective as maximizing the ROC_AUC score on the testing dataset after training the SVM using the hyperparameters with the training dataset.

```
class obj_func_landmine():

    def __init__(self, X_train, Y_train, X_test, Y_test):
```

(continues on next page)

(continued from previous page)

```
        self.X_train = X_train              # Training X
        self.Y_train = Y_train              # Training Y
        self.X_test = X_test                # Testing X
        self.Y_test = Y_test                # Testing Y
        self.fmax = 1

    def f(self, point):
        C = point[0]                        # First parameter
        gam = point[1]                      # Second parameter

        clf = svm.SVC(kernel="rbf", C=C, gamma=gam, probability=True)       # The␣
↪machine learning model is SVM
        clf.fit(self.X_train, self.Y_train)                                 # Fit the␣
↪model using training data
        pred = clf.predict_proba(self.X_test)                               # Make␣
↪prediction on the testing
        score = roc_auc_score(self.Y_test, pred[:, 1])                      # The reward␣
↪is the ROC_AUC score

        return score
```

Input the data and then split the data into the training dataset and the testing dataset. Then define the objective function using the datasets.

```
landmine_data = pickle.load(open("../../PyXAB/landmine/landmine_formated_data.pkl", "rb
↪"))
all_X_train, all_Y_train, all_X_test, all_Y_test = landmine_data["all_X_train"],␣
↪landmine_data["all_Y_train"], \
                                                   landmine_data["all_X_test"],␣
↪landmine_data["all_Y_test"]

X_train = all_X_train[0]
Y_train = np.squeeze(all_Y_train[0])
X_test = all_X_test[0]
Y_test = np.squeeze(all_Y_test[0])

target = obj_func_landmine(X_train=X_train, X_test=X_test, Y_train=Y_train, Y_test=Y_
↪test)
```

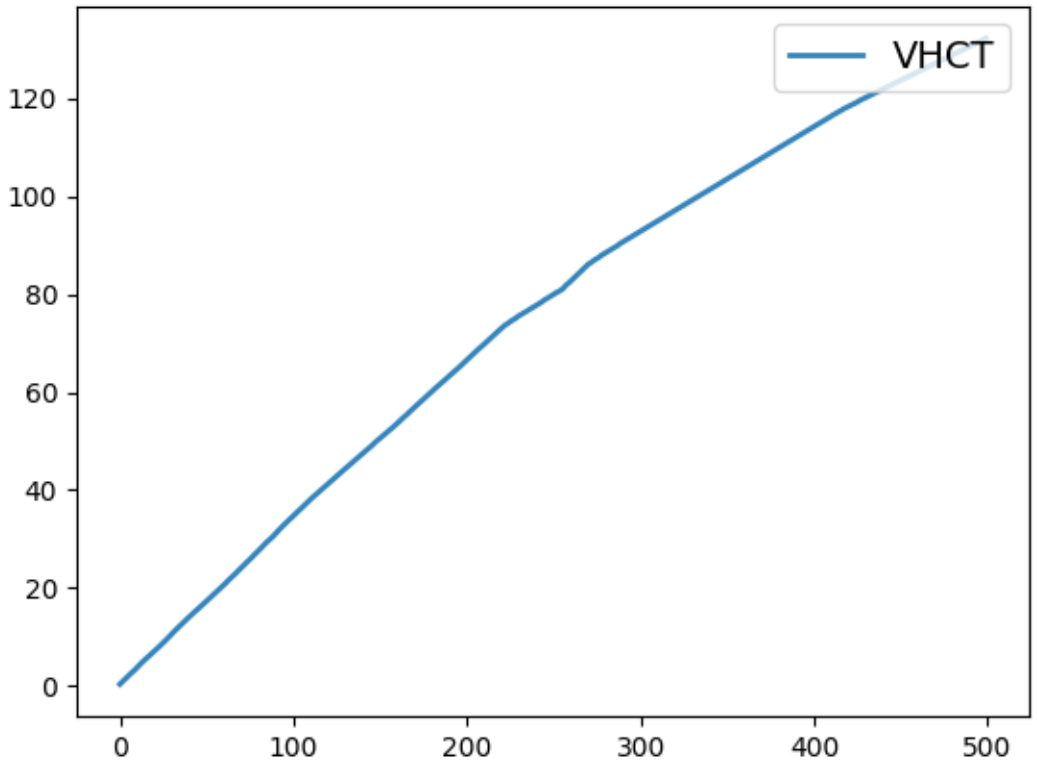
Define the number of rounds, the domain, the partition, and the algorithm for the learning process

```
T = 500
domain = [[1e-4, 10.0], [1e-2, 10.0]]
partition = BinaryPartition
algo = VHCT(domain=domain, rho=0.5, partition=partition)

# To plot the regret, we can initialize the cumulative regret and the cumulative regret␣
↪list

cumulative_regret = 0
cumulative_regret_list = []
```

In each iteration of the learning process, the algorithm calls the `pull(t)` function to obtain a point, and then the reward

for the point is returned to the algorithm by calling `receive_reward(t, reward)`.

```
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target.f(point)
    algo.receive_reward(t, reward)
    inst_regret = target.fmax - target.f(point)
    cumulative_regret += inst_regret
    cumulative_regret_list.append(cumulative_regret)


# plot the regret
plot_regret(np.array(cumulative_regret_list), name='VHCT')
```



Total running time of the script: (0 minutes 22.267 seconds)

2.4 Algorithms

Our implemented X-Armed Bandit algorithms can be classified into different categories according to different features in the algorithm design.

Algorithm	Research	Stochastic	Cumulative	Anytime
DiRect	paper			
DOO	DOO paper			
SOO	SOO paper			
Zooming	Zooming paper			
T-HOO	T-HOO paper			
StoSOO	StoSOO paper			
HCT	HCT paper			
POO*	POO paper			
GPO*	GPO paper			
PCT	GPO paper			
SequOOL	SequOOL paper			
StroquOOL	StroquOOL paper			
VROOM	VROOM paper			
VHCT	VHCT paper			
VPCT	N.A.			

- **(Stochastic)** For some algorithms such as T_HOO and HCT, they perform well in the stochastic X-Armed Bandit setting when there is noise in the problem. However for some of the algorithms, e.g., DOO, they only work in the noise-less (deterministic) setting.
- **(Cumulative)** For some algorithms such as T_HOO and HCT, they are designed to optimize the cumulative regret, i.e., the performance over the whole learning process. However for algorithms such as StoSOO and StroquOOL, they will optimize the simple regret, i.e., the final-round/last output performance.
- **(Anytime)** For some algorithms such as SequOOL and StroquOOL, they need the total number of rounds (budget) information to run the algorithm, but for algorithms such as T_HOO and HCT, they do not need such information.

Note: Please refer to the following details for more information.

2.4.1 Zooming Algorithm

Introduction

paper, code

Title: Multi-Armed Bandits in Metric Spaces

Authors: Robert Kleinberg, Aleksandrs Slivkins, Eli Upfal

Abstract: In a multi-armed bandit problem, an online algorithm chooses from a set of strategies in a sequence of n trials so as to maximize the total payoff of the chosen strategies. While the performance of bandit al- gorithms with a small finite strategy set is quite well understood, bandit problems with large strategy sets are still a topic of very active investigation, motivated by practical applications such as online auctions and web advertisement. The goal of such research is to identify broad and natural classes of strategy sets and payoff functions which enable the design of efficient solutions. In this work we study a very general setting for the multi-armed bandit problem in which the strategies form a metric space, and the payoff function satisfies a Lipschitz condition with respect to the metric. We refer to this problem as the Lipschitz MAB problem. We present a solution for the multi- armed problem in this setting. That is, for every metric space (L,X) we define an isometry invariant MaxMinCOV(X) which bounds from below the performance of Lipschitz MAB algorithms for X, and we present an algorithm which comes arbitrarily close to meeting this bound. Furthermore, our technique gives even better results for benign payoff functions.

Algorithm 2.3 (Zooming Algorithm)**.** *Each phase i runs for 2^i rounds. In the beginning of the phase no strategies are active. In each round do the following:*

1. *If some strategy is not covered, make it active.*
2. *Play an active strategy with the maximal index (3); break ties arbitrarily.*

We formulate the main result of this section as follows:

Theorem 2.4. *Consider the standard Lipschitz MAB problem. Let $\mathcal{A}$ be Algorithm 2.3. Then $\forall\, C > 0$*

$$R_{\mathcal{A}}(t) \leq O(C \log t)^{1/(2+d)} \times t^{1-1/(2+d)} \text{ for all } t, \tag{4}$$

where d is the C-zooming dimension of the problem instance.

Algorithm Parameters

- *nu (float)* – parameter nu of the Zooming algorithm
- *rho (float)* – parameter rho of the Zooming algorithm
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.Zooming import Zooming

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = Zooming(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

2.4.2 DOO Algorithm

Introduction

paper, code

Title: Optimistic Optimization of a Deterministic Function without the Knowledge of its Smoothness

Authors: Remi Munos

Abstract: We consider a global optimization problem of a deterministic function f in a semi-metric space, given a finite budget of n evaluations. The function f is assumed to be locally smooth (around one of its global maxima) with respect to a semi-metric l. We describe two algorithms based on optimistic exploration that use a hierarchical partitioning of the space at all scales. A first contribution is an algorithm, DOO, that requires the knowledge of l. We report a finite-sample performance bound in terms of a measure of the quantity of near-optimal states. We then define a second algorithm, SOO, which does not require the knowledge of the semi- metric l under which f is smooth, and whose performance is almost as good as DOO optimally-fitted.

Initialization: $\mathcal{T}_1 = \{(0,0)\}$ (root node)
for $t = 1$ to n **do**
  Select the leaf $(h,j) \in \mathcal{L}_t$ with maximum $b_{h,j} \stackrel{\text{def}}{=} f(x_{h,j}) + \delta(h)$ value.
  Expand this node: add to $\mathcal{T}_t$ the K children of (h,j)
end for
Return $x(n) = \arg\max_{(h,i)\in\mathcal{T}_n} f(x_{h,i})$

Figure 1: Deterministic optimistic optimization (DOO) algorithm.

Algorithm Parameters

- *n (int)* – The total number of rounds (budget)
- *delta (function)* – The function to compute the delta value for each depth
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.DOO import DOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = DOO(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.3 SOO Algorithm

Introduction

paper, code

Title: Optimistic Optimization of a Deterministic Function without the Knowledge of its Smoothness

Authors: Remi Munos

Abstract: We consider a global optimization problem of a deterministic function f in a semi-metric space, given a finite budget of n evaluations. The function f is assumed to be locally smooth (around one of its global maxima) with respect to a semi-metric l. We describe two algorithms based on optimistic exploration that use a hierarchical partitioning of the space at all scales. A first contribution is an algorithm, DOO, that requires the knowledge of l. We report a finite-sample performance bound in terms of a measure of the quantity of near-optimal states. We then define a second algorithm, SOO, which does not require the knowledge of the semi- metric l under which f is smooth, and whose performance is almost as good as DOO optimally-fitted.

The maximum depth function $t \mapsto h_{\max}(t)$ is a parameter of the algorithm.
Initialization: $\mathcal{T}_1 = \{(0,0)\}$ (root node). Set $t = 1$.
while True **do**
  Set $v_{\max} = -\infty$.
  for $h = 0$ to $\min(\text{depth}(\mathcal{T}_t), h_{\max}(t))$ **do**
    Among all leaves $(h,j) \in \mathcal{L}_t$ of depth h, select $(h,i) \in \arg\max_{(h,j)\in\mathcal{L}_t} f(x_{h,j})$
    if $f(x_{h,i}) \geq v_{\max}$ **then**
      Expand this node: add to $\mathcal{T}_t$ the K children $(h+1, i_k)_{1\leq k\leq K}$
      Set $v_{\max} = f(x_{h,i})$, Set $t = t+1$
      if $t = n$ **then Return** $x(n) = \arg\max_{(h,i)\in\mathcal{T}_n} x_{h,i}$
    end if
  end for
end while.

Figure 2: Simultaneous Optimistic Optimization (SOO) algorithm.

Algorithm Parameters

- *n (int)* – The total number of rounds (budget)
- *h_max (int)* – The largest searching depth
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.SOO import SOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = SOO(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.4 StoSOO Algorithm

Introduction

paper, code

Title: Stochastic Simultaneous Optimistic Optimization

Authors: Michal Valko, Alexandra Carpentier, Remi Munos

Abstract: We study the problem of global maximization of a function f given a finite number of evaluations perturbed by noise. We consider a very weak assumption on the function, namely that it is locally smooth (in some precise sense) with respect to some semi-metric, around one of its global maxima. Compared to previous works on bandits in general spaces (Kleinberg et al., 2008; Bubeck et al., 2011a) our algorithm does not require the knowledge of this semi-metric. Our algorithm, StoSOO, follows an optimistic strategy to iteratively construct upper confidence bounds over the hierarchical partitions of the function domain to decide which point to sample next. A finite-time analysis of StoSOO shows that it performs almost as well as the best specifically-tuned algorithms even though the local smoothness of the function is not known.

Algorithm 1 `StoSOO`

Stochastic Simultaneous Optimistic Optimization

Parameters: number of function evaluations n, maximum number of evaluations per node $k > 0$, maximum depth $h_{\max}$, and $\delta > 0$.

Initialization:

$\mathcal{T} \leftarrow \{\circ[0,0]\}$ {root node}

$t \leftarrow 0$ {number of evaluations}

while $t \leq n$ **do**

$b_{\max} \leftarrow -\infty$

for $h = 0$ to $\min(\text{depth}(\mathcal{T}), h_{\max})$ **do**

if $t \leq n$ **then**

For each leaf $\circ[h,j] \in \mathcal{L}$, compute its b-value:

$b_{h,j}(t) = \hat{\mu}_{h,j}(t) + \sqrt{\log(nk/\delta)/(2T_{h,j}(t))}$

Among leaves $\circ[h,j] \in \mathcal{L}_t$ at depth h, select

$$\circ[h,i] \in \arg\max_{\circ[h,j] \in \mathcal{L}} b_{h,j}(t)$$

if $b_{h,i}(t) \geq b_{\max}$ **then**

if $T_{h,i}(t) < k$ **then**

Evaluate (sample) state $x_t = x_{h,i}$.

Collect reward r_t (s.t. $\mathbb{E}[r_t | x_t] = f(x_t)$).

$t \leftarrow t + 1$

else {i.e. $T_{h,i}(t) \geq k$, expand this node}

Add the K children of $\circ[h,i]$ to $\mathcal{T}$

$b_{\max} \leftarrow b_{h,i}(t)$

end if

end if

end if

end for

end while

Output: The representative point with the highest $\hat{\mu}_{h,j}(n)$ among the deepest expanded nodes:

$$x(n) = \arg\max_{x_{h,j}} \hat{\mu}_{h,j}(n) \text{ s.t. } h = \text{depth}(\mathcal{T} \setminus \mathcal{L}).$$

Algorithm Parameters

- *n (int)* – The total number of rounds (budget)
- *k (int)* - The maximum number of pulls per node
- *h_max (int)* – The largest searching depth
- *delta (float)* - The confidence parameter delta
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.StoSOO import StoSOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = StoSOO(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.5 T-HOO Algorithm

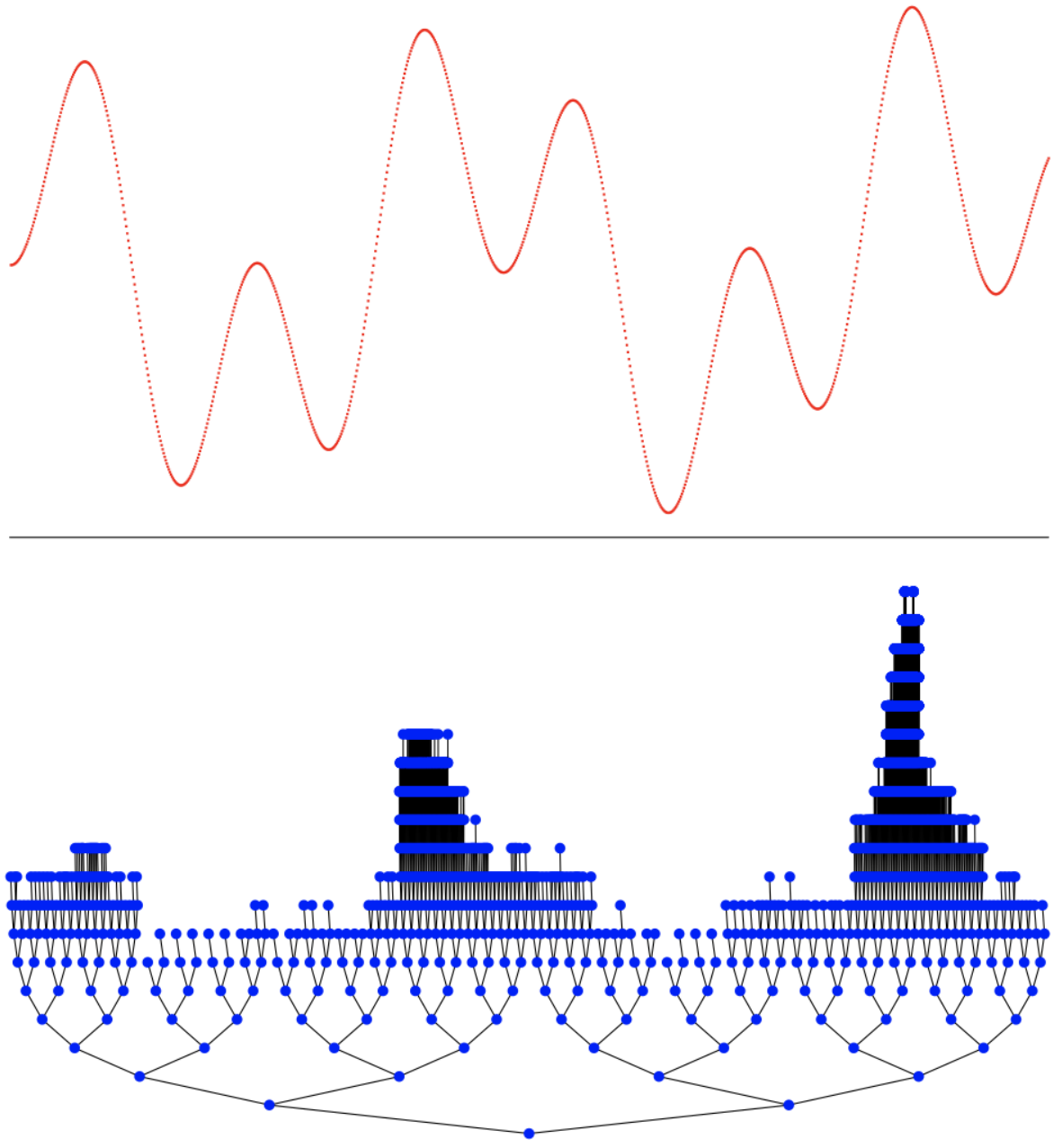
Introduction

paper, code

Title: X -Armed Bandits

Authors: Sebastien Bubeck, Remi Munos, Gilles Stoltz, Csaba Szepesvari

Abstract: We consider a generalization of stochastic bandits where the set of arms, X, is allowed to be a generic measurable space and the mean-payoff function is "locally Lipschitz" with respect to a dissimilarity function that is known to the decision maker. Under this condition we construct an arm selection policy, called HOO (hierarchical optimistic optimization), with improved regret bounds compared to previous results for a large class of problems. In particular, our results imply that if X is the unit hypercube in a Euclidean space and the mean-payoff function has a finite number of global maxima around which the behavior of the function is locally continuous with a known smoothness degree, then the expected regret of HOO is bounded up to a logarithmic factor by n, that is, the rate of growth of the regret is independent of the dimension of the space. We also prove the minimax optimality of our algorithm when the dissimilarity is a metric. Our basic strategy has quadratic computational complexity as a function of the number of time steps and does not rely on the doubling trick. We also introduce a modified strategy, which relies on the doubling trick but runs in linearithmic time. Both results are improvements with respect to previous approaches.



Algorithm Parameters

- *nu (float)* – parameter nu of the T_HOO algorithm
- *rho (float)* – parameter rho of the T_HOO algorithm
- *rounds (int)* - total number of rounds
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.HOO import T_HOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = T_HOO(rounds=rounds, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

2.4.6 HCT Algorithm

Introduction

paper, code

Title: Online Stochastic Optimization under Correlated Bandit Feedback

Authors: Mohammad Gheshlaghi Azar, Alessandro Lazaric, Emma Brunskill

Abstract: In this paper we consider the problem of online stochastic optimization of a locally smooth function under bandit feedback. We introduce the high-confidence tree (HCT) algorithm, a novel anytime X-armed bandit algorithm, and derive regret bounds matching the performance of state- of-the-art algorithms in terms of the dependency on number of steps and the near-optimality dimension. The main advantage of HCT is that it handles the challenging case of correlated bandit feedback (reward), whereas existing methods require rewards to be conditionally independent. HCT also improves on the state-of-the-art in terms of the memory requirement, as well as requiring a weaker smoothness assumption on the mean-reward function in comparison with the existing anytime algorithms. Finally, we discuss how HCT can be applied to the problem of policy search in reinforcement learning and we report preliminary empirical results.

Algorithm 1 The *HCT* algorithm.

Require: Parameters $\nu_1 > 0$, $\rho \in (0,1)$, $c > 0$, tree structure $(\mathcal{P}_{h,i})_{h\geq 0, 1\leq i\leq 2^i}$ and confidence δ.
Initialize $t = 1$, $\mathcal{T}_t = \{(0,1),(1,1),(1,2)\}$, $H(t) = 1$, $U_{1,1}(t) = U_{1,2}(t) = +\infty$,
loop
  if $t = t^+$ **then** ▷ Refresh phase
    for all $(h,i) \in \mathcal{T}_t$ **do**
      $U_{h,i}(t) \leftarrow \widehat{\mu}_{h,i}(t) + \nu_1 \rho^h + \sqrt{\frac{c^2 \log(1/\tilde{\delta}(t^+))}{T_{h,i}(t)}}$
    end for;
    for all $(h,i) \in \mathcal{T}_t$ Backward from $H(t)$ **do**
      if $(h,i) \in$ leaf$(\mathcal{T}_t)$ **then**
        $B_{h,i}(t) \leftarrow U_{h,i}(t)$
      else
        $B_{h,i}(t) \leftarrow \min\left[U_{h,i}(t), \max_{j\in\{2i-1,2i\}} B_{h+1,j}(t)\right]$
      end if
    end for
  end if;
  $\{(h_t,i_t), P_t\} \leftarrow$ OptTraverse$(\mathcal{T}_t)$
  if Algorithm ***HCT*-iid** **then**
    Pull arm x_{h_t,i_t} and observe r_t
    $t = t+1$
  else if Algorithm ***HCT*-Γ** **then**
    $T_{cur} = T_{h_t,i_t}(t)$
    while $T_{h_t,i_t}(t) < 2T_{cur}$ **AND** $t < t^+$ **do**
      Pull arm $x_{h,i}$ and observe r_t
      $(h_{t+1}, i_{t+1}) = (h_t, i_t)$
      $t = t+1$
    end while
  end if
  Update counter $T_{h_t,i_t}(t)$ and empirical average $\widehat{\mu}_{h_t,i_t}(t)$
  $U_{h_t,i_t}(t) \leftarrow \widehat{\mu}_{h_t,i_t}(t) + \nu_1\rho^h + \sqrt{\frac{c^2\log(1/\tilde{\delta}(t^+))}{T_{h_t,i_t}(t)}}$
  UpdateB$(\mathcal{T}_t, P_t, (h_t,i_t))$
  $\tau_h(t) = \frac{c^2\log(1/\tilde{\delta}(t^+))}{\nu_1^2}\rho^{-2h_t}$
  if $T_{h_t,i_t}(t) \geq \tau_{h_t}(t)$ AND $(h_t,i_t) =$leaf$(\mathcal{T})$ **then**
    $\mathcal{I}_t = \{(h_t+1, 2i_t-1), (h_t+1, 2i_t)\}$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{I}_t$
    $U_{h_t+1,2i_t-1}(t) = U_{h_t+1,2i_t}(t) = +\infty$
  end if
end loop

Algorithm Parameters

- *nu (float)* – parameter nu of the HCT algorithm
- *rho (float)* – parameter rho of the HCT algorithm
- *c (float)* – parameter c of the HCT algorithm
- *delta (float)* – confidence parameter delta of the HCT algorithm
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.HCT import HCT

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = HCT(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

2.4.7 POO Algorithm

Introduction

paper, code

Title: Black-box optimization of noisy functions with unknown smoothness

Authors: Jean-Bastien Grill, Michael Valko, Remi Munos

Abstract: We study the problem of black-box optimization of a function f of any dimension, given function evaluations perturbed by noise. The function is assumed to be locally smooth around one of its global optima, but this smoothness is unknown. Our contribution is an adaptive optimization algorithm, POO or parallel optimistic optimization, that is able to deal with this setting. POO performs almost as well as the best known algorithms requiring the knowledge of the smoothness. Furthermore, POO works for a larger class of functions than what was previously considered, especially for functions that are difficult to optimize, in a very precise sense. We provide a finite-time analysis of POO's performance, which shows that its error after n evaluations is at most a factor of ln n away from the error of the best known optimization algorithms using the knowledge of the smoothness.

Algorithm 1 POO

Parameters: K, $\mathcal{P} = \{\mathcal{P}_{h,i}\}$
  Optional parameters: $\rho_{\max}, \nu_{\max}$
Initialization:
  $D_{\max} \leftarrow \ln K / \ln(1/\rho_{\max})$
  $n \leftarrow 0$ {number of evaluation performed}
  $N \leftarrow 1$ {number of HOO instances}
  $\mathcal{S} \leftarrow \{(\nu_{\max}, \rho_{\max})\}$ {set of HOO instances}
while computational budget is available **do**
  while $N \geq \frac{1}{2} D_{\max} \ln(n/(\ln n))$ **do**
    for $i \leftarrow 1, \ldots, N$ **do** {start new HOOs}
      $s \leftarrow \left(\nu_{\max}, {\rho_{\max}}^{2N/(2i+1)}\right)$
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
      Perform $\frac{n}{N}$ function evaluation with HOO(s)
      Update the average reward $\widehat{\mu}[s]$ of HOO(s)
    end for
    $n \leftarrow 2n$
    $N \leftarrow 2N$
  end while{ensure there is enough HOOs}
  for $s \in \mathcal{S}$ **do**
    Perform a function evaluation with HOO(s)
    Update the average reward $\widehat{\mu}[s]$ of HOO(s)
  end for
  $n \leftarrow n + N$
end while
$s^\star \leftarrow \text{argmax}_{s \in \mathcal{S}} \widehat{\mu}[s]$
Output: A random point evaluated by HOO($s^\star$)

Note: Make sure to use *get_last_point()* to get the final output

Algorithm Parameters

- numax (float) – parameter nu_max in the algorithm
- rhomax (float) – parameter rho_max in the algorithm, the maximum rho used
- rounds (int) – the number of rounds/budget
- domain (list(list)) – the domain of the objective function
- partition – the partition used in the optimization process
- algo – the baseline algorithm used by the wrapper, such as T_HOO or HCT

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.POO import POO
from PyXAB.algos.HOO import T_HOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = POO(rounds=rounds, domain=domain, algo=T_HOO)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.8 GPO Algorithm

Introduction

paper, code

Title: General parallel optimization a without metric

Authors: Xuedong Shang, Emilie Kaufmann, Michal Valko

Abstract: Hierarchical bandits are an approach for global optimization of emph{extremely} irregular functions. This paper provides new elements regarding POO, an adaptive meta-algorithm that does not require the knowledge of local smoothness of the target function. We first highlight the fact that the subroutine algorithm used in POO should have a small regret under the assumption of emph{local smoothness with respect to the chosen partitioning}, which is unknown if it is satisfied by the standard subroutine HOO. In this work, we establish such regret guarantee for HCT,

which is another hierarchical optimistic optimization algorithm that needs to know the smoothness. This confirms the validity of POO. We show that POO can be used with HCT as a subroutine with a regret upper bound that matches the one of best-known algorithms using the knowledge of smoothness up to a log n factor. On top of that, we further propose a more general wrapper, called GPO, that can cope with algorithms that only have simple regret guarantees.

Algorithm 5: General parallel optimization (GPO)

Input : base algorithm $\mathcal{A}$, budget n, $\rho_{\max}, \nu_{\max}$, K

Compute $N = \lceil (1/2) D_{\max} \ln((n/2)/\ln(n/2)) \rceil$ the number of instances

for $i \leftarrow 1, \ldots, N$ **do**

  $s \leftarrow \left(\nu_{\max}, {\rho_{\max}}^{2N/(2i+1)}\right)$

  Run $\mathcal{A}(s)$ for $\lfloor n/(2N) \rfloor$ time steps and output a recommendation $\widetilde{x}_s$

  Get $\lfloor n/(2N) \rfloor$ noisy evaluations of $f(\widetilde{x}_s)$ and compute their average $V[s]$

end

$s^\star \leftarrow \arg\max_s V[s]$

return $\widetilde{x}_{s^\star}$

Algorithm Parameters

- numax (float) – parameter nu_max in the algorithm
- rhomax (float) – parameter rho_max in the algorithm, the maximum rho used
- rounds (int) – the number of rounds/budget
- domain (list(list)) – the domain of the objective function
- partition – the partition used in the optimization process. Default: BinaryPartition
- algo – the baseline algorithm used by the wrapper, such as T_HOO or HCT

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.GPO import GPO
from PyXAB.algos.HOO import T_HOO

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = GPO(rounds=rounds, domain=domain, algo=T_HOO)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

(continues on next page)

(continued from previous page)

```
algo.get_last_point()
```

2.4.9 PCT Algorithm

Introduction

paper, code

Title: General parallel optimization a without metric

Authors: Xuedong Shang, Emilie Kaufmann, Michal Valko

Abstract: Hierarchical bandits are an approach for global optimization of emph{extremely} irregular functions. This paper provides new elements regarding POO, an adaptive meta-algorithm that does not require the knowledge of local smoothness of the target function. We first highlight the fact that the subroutine algorithm used in POO should have a small regret under the assumption of emph{local smoothness with respect to the chosen partitioning}, which is unknown if it is satisfied by the standard subroutine HOO. In this work, we establish such regret guarantee for HCT, which is another hierarchical optimistic optimization algorithm that needs to know the smoothness. This confirms the validity of POO. We show that POO can be used with HCT as a subroutine with a regret upper bound that matches the one of best-known algorithms using the knowledge of smoothness up to a log n factor. On top of that, we further propose a more general wrapper, called GPO, that can cope with algorithms that only have simple regret guarantees.

Algorithm 5: General parallel optimization (GPO)

Input : base algorithm $\mathcal{A}$, budget n, $\rho_{\max}$, $\nu_{\max}$, K

Compute $N = \lceil (1/2) D_{\max} \ln((n/2)/\ln(n/2)) \rceil$ the number of instances

for $i \leftarrow 1, \ldots, N$ **do**

  $s \leftarrow \left(\nu_{\max}, {\rho_{\max}}^{2N/(2i+1)}\right)$

  Run $\mathcal{A}(s)$ for $\lfloor n/(2N) \rfloor$ time steps and output a recommendation $\widetilde{x}_s$

  Get $\lfloor n/(2N) \rfloor$ noisy evaluations of $f(\widetilde{x}_s)$ and compute their average $V[s]$

end

$s^\star \leftarrow \arg\max_s V[s]$

return $\widetilde{x}_{s^\star}$

Algorithm Parameters

- numax (float) – parameter nu_max in the algorithm
- rhomax (float) – parameter rho_max in the algorithm, the maximum rho used
- rounds (int) – the number of rounds/budget
- domain (list(list)) – the domain of the objective function
- partition – the partition used in the optimization process. Default: BinaryPartition

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.PCT import PCT

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = PCT(rounds=rounds, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.10 SequOOL Algorithm

Introduction

paper, code

Title: A simple parameter-free and adaptive approach to optimization under a minimal local smoothness assumption

Authors: Peter L. Bartlett, Victor Gabillon, Michal Valko

Abstract: We study the problem of optimizing a function under a budgeted number of evaluations. We only assume that the function is locally smooth around one of its global optima. The difficulty of optimization is measured in terms of 1) the amount of noise b of the function evaluation and 2) the local smoothness, d, of the function. A smaller d results in smaller optimization error. We come with a new, simple, and parameter-free approach. First, for all values of b and d, this approach recovers at least the state-of-the-art regret guarantees. Second, our approach additionally obtains these results while being agnostic to the values of both b and d. This leads to the first algorithm that naturally adapts to an unknown range of noise b and leads to significant improvements in a mod- erate and low-noise regime. Third, our approach also obtains a remarkable improvement over the state-of-the-art SOO algorithm when the noise is very low which includes the case of optimization under deterministic feedback (b = 0). There, under our minimal local smoothness assumption, this improvement is of exponential magnitude and holds for a class of functions that covers the vast ma- jority of functions that practitioners optimize (d = 0). We show that our algorithmic improvement is borne out in experiments as we empirically show faster convergence on common benchmarks.

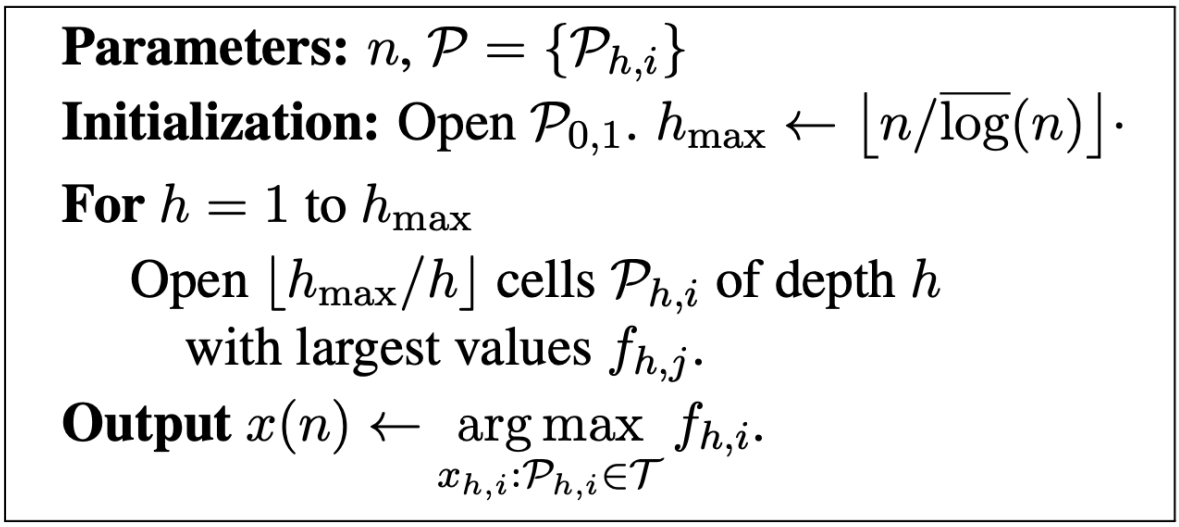


Figure 1: The SequOOL algorithm

Algorithm Parameters

- *n (int)* – The total number of rounds (budget)
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.SequOOL import SequOOL

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = SequOOL(n=rounds, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.11 StroquOOL Algorithm

Introduction

paper, code

Title: A simple parameter-free and adaptive approach to optimization under a minimal local smoothness assumption

Authors: Peter L. Bartlett, Victor Gabillon, Michal Valko

Abstract: We study the problem of optimizing a function under a budgeted number of evaluations. We only assume that the function is locally smooth around one of its global optima. The difficulty of optimization is measured in terms of 1) the amount of noise b of the function evaluation and 2) the local smoothness, d, of the function. A smaller d results in smaller optimization error. We come with a new, simple, and parameter-free approach. First, for all values of b and d, this approach recovers at least the state-of-the-art regret guarantees. Second, our approach additionally obtains these results while being agnostic to the values of both b and d. This leads to the first algorithm that naturally adapts to an unknown range of noise b and leads to significant improvements in a mod- erate and low-noise regime. Third, our approach also obtains a remarkable improvement over the state-of-the-art SOO algorithm when the noise is very low which includes the case of optimization under deterministic feedback (b = 0). There, under our minimal local smoothness assumption, this improvement is of exponential magnitude and holds for a class of functions that covers the vast ma- jority of functions that practitioners optimize (d = 0). We show that our algorithmic improvement is borne out in experiments as we empirically show faster convergence on common benchmarks.

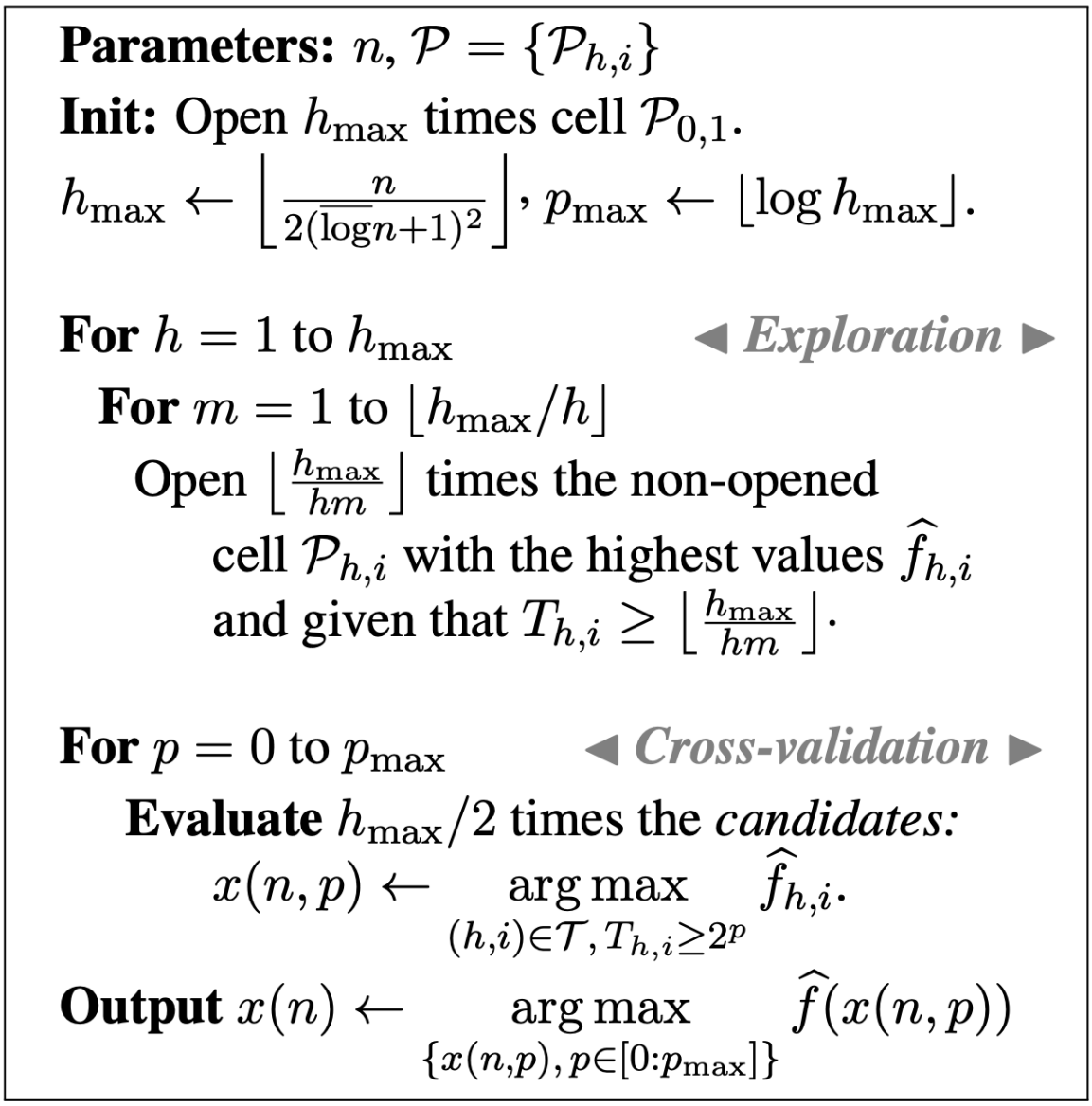


Figure 2: The StroquOOL algorithm

Algorithm Parameters

- *n (int)* – The total number of rounds (budget)
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.StroquOOL import StroquOOL

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = StroquOOL(n=rounds, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.12 VROOM Algorithm

Introduction

paper, code

Title: Derivative-Free & Order-Robust Optimisation

Authors: Victor Gabillon, Rasul Tutunov, Michal Valko, Haitham Bou Ammar

Abstract: In this paper, we formalise order-robust optimisation as an instance of online learning minimising simple regret, and propose VROOM, a zeroth order optimisation algorithm capable of achieving vanishing regret in non-stationary environments, while recovering favorable rates under stochastic reward-generating processes. Our results are the first to target simple regret definitions in adversarial scenarios unveiling a challenge that has been rarely considered in prior work.

Parameters: $\mathcal{P} = \{\mathcal{P}_{h,i}\}, b, n, f_{\max}$
Set $\delta = \frac{4b}{f_{\max}\sqrt{n}}$.

For $t = 1, \ldots, n$ ◀ *Exploration* ▶

For each depth $h \in [\lfloor \log_K(n) \rfloor]$, rank[a] the cells by decreasing order of $\widehat{f^-_{h,i}}(t-1)$: Rank cell $\mathcal{P}_{h,i}$ as $\widehat{\langle i \rangle}_{h,t}$.

$x_t \sim \mathcal{U}_{\mathcal{P}}(\mathcal{P}_{h_t,i_t})$ where $\mathcal{P}_{h_t,i_t}$ is sampled so that for any $h \in [\lfloor \log_K(n) \rfloor]$ and any $i \in [K^h]$,

$$\boldsymbol{p}_{h,i,t} \triangleq \mathbb{P}(\mathcal{P}_{h_t,i_t} = \mathcal{P}_{h,i}) \triangleq \frac{1}{h\widehat{\langle i \rangle}_{h,t}\overline{\log}_K(n)}$$

and where $\overline{\log}_K(n) = \sum_{h=1}^{\lfloor \log_K(n) \rfloor} \sum_{i=1}^{K^h} \frac{1}{hi}$.

Output $x(n) \sim \mathcal{U}_{\mathcal{P}}(\mathcal{P}_{h(n),i(n)})$ where $(h(n), i(n)) \leftarrow \arg\max_{(h,i)} \widetilde{F}_{h,i}(n) - B_{h,i}(n)$

[a] Equalities between cells or comparisons with cells that have not been pulled yet are broken arbitrarily.

Figure 2: The VROOM algorithm

Algorithm Parameters

- *n: int*- The total number of rounds (budget)
- *h_max: int* - The maximum depth of the partition
- *b: float* - The parameter that measures the variation of the function
- *f_max: float* - An upper bound of the objective function
- *domain: list(list)*- The domain of the objective to be optimized
- *partition* - The partition choice of the algorithm

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.VROOM import VROOM

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = VROOM(n=rounds, b=1, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)

algo.get_last_point()
```

2.4.13 VHCT Algorithm

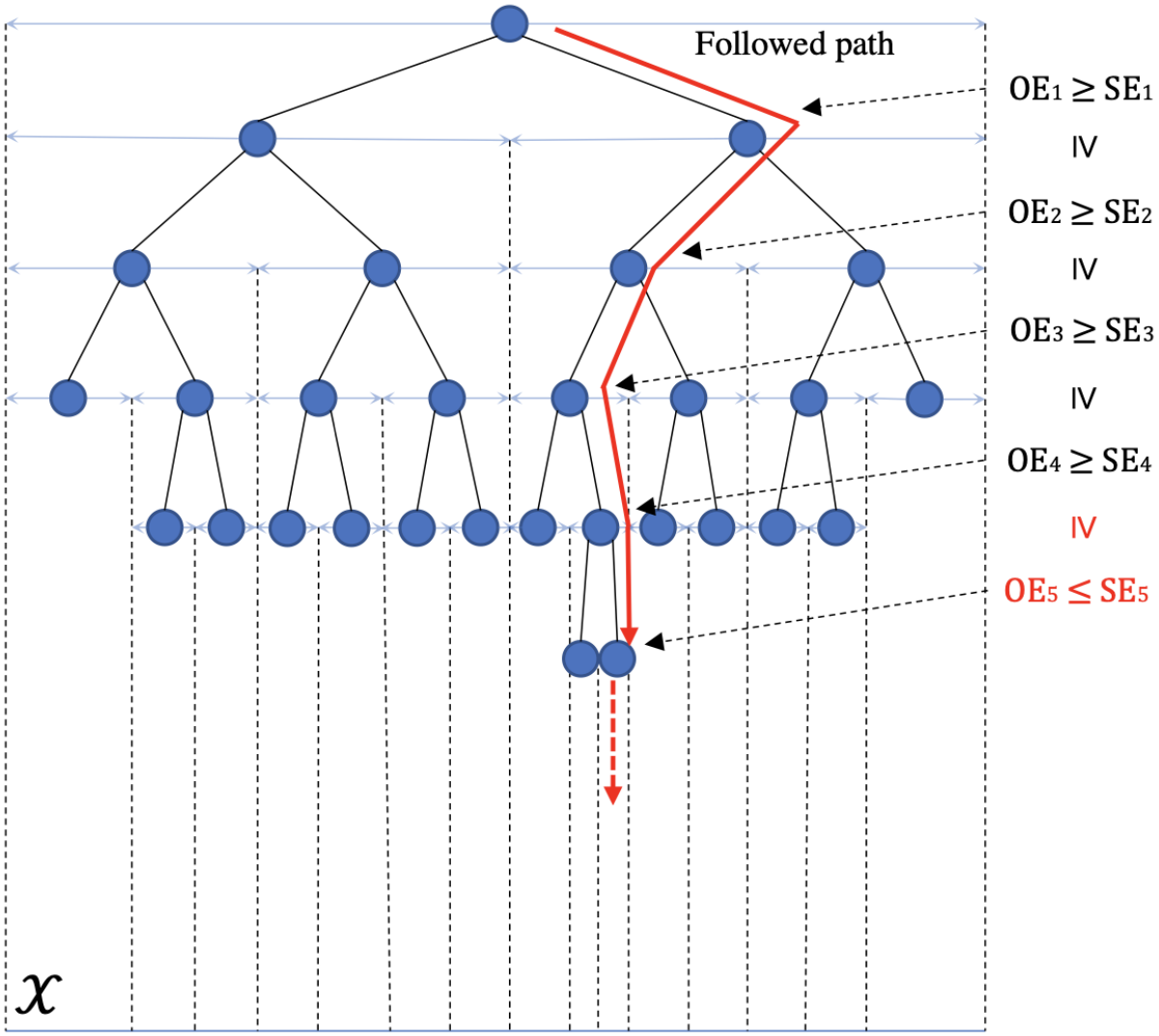
Introduction

paper, code

Title: Optimum-statistical Collaboration Towards General and Efficient Black-box Optimization

Authors: Wenjie Li, Chi-Hua Wang, Guang Cheng, Qifan Song

Abstract: In this paper, we make the key delineation on the roles of resolution and statistical uncertainty in hierarchical bandits-based black-box optimization algorithms, guiding a more general analysis and a more efficient algorithm design. We introduce the optimum-statistical collaboration, an algorithm framework of managing the interaction between optimization error flux and statistical error flux evolving in the optimization process. We provide a general analysis of this framework without specifying the forms of statistical error and uncertainty quantifier. Our framework and its analysis, due to their generality, can be applied to a large family of functions and partitions that satisfy different local smoothness assumptions and have different numbers of local optimums, which is much richer than the class of functions studied in prior works. Our framework also inspires us to propose a better measure of the statistical uncertainty and consequently a variance-adaptive algorithm VHCT. In theory, we prove the algorithm enjoys rate-optimal regret bounds under different local smoothness assumptions; in experiments, we show the algorithm outperforms prior efforts in different settings.



Algorithm Parameters

- *nu (float)* – parameter nu of the VHCT algorithm
- *rho (float)* – parameter rho of the VHCT algorithm
- *c (float)* – parameter c of the VHCT algorithm
- *delta (float)* – confidence parameter delta of the VHCT algorithm
- *bound (float)* – the noise upper bound parameter bound
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.VHCT import VHCT

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
algo = VHCT(domain=domain)

for t in range(1000):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

2.4.14 VPCT Algorithm

Introduction

code

We introduce VPCT by combining VHCT with the GPO algorithm for a smoothness-agnostic algorithm

Algorithm Parameters

- *nu_max (float)* – parameter nu_max of the VPCT algorithm
- *rho_max (float)* – parameter rho of the VPCT algorithm
- *rounds (int)* - the number of rounds/budget
- *domain (list(list))* – The domain of the objective to be optimized
- *partition* – The partition choice of the algorithm. Default: BinaryPartition.

Usage Example

Note: Make sure to use *get_last_point()* to get the final output

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.VPCT import VPCT

domain = [[0, 1]]               # Parameter is 1-D and between 0 and 1
target = Garland()
rounds = 1000
algo = VHCT(rounds=rounds, domain=domain)

for t in range(rounds):
    point = algo.pull(t)
    reward = target(point)
    algo.receive_reward(t, reward)
```

(continues on next page)

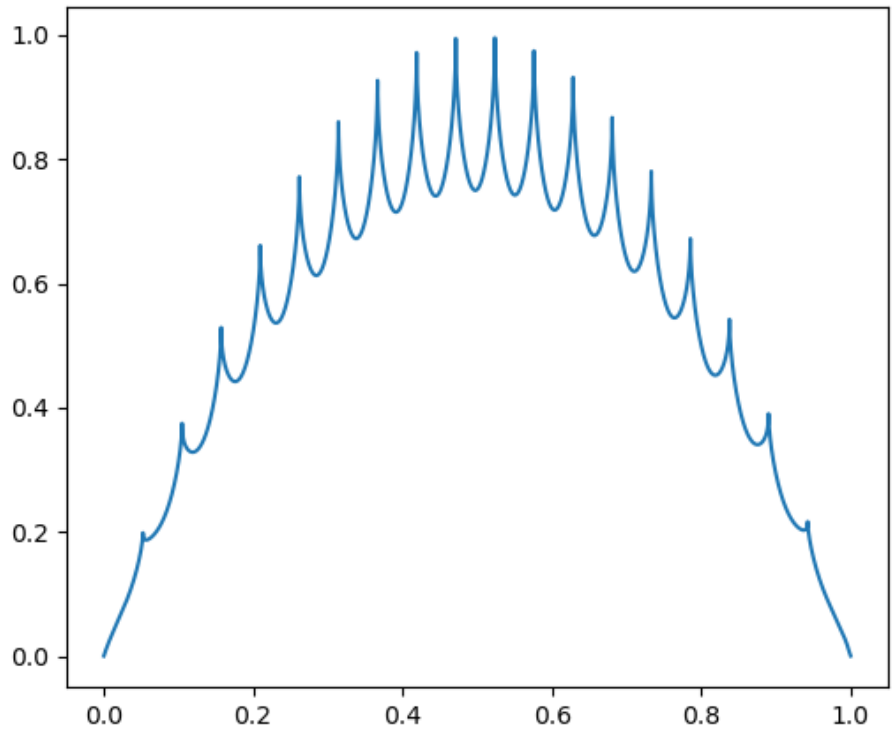
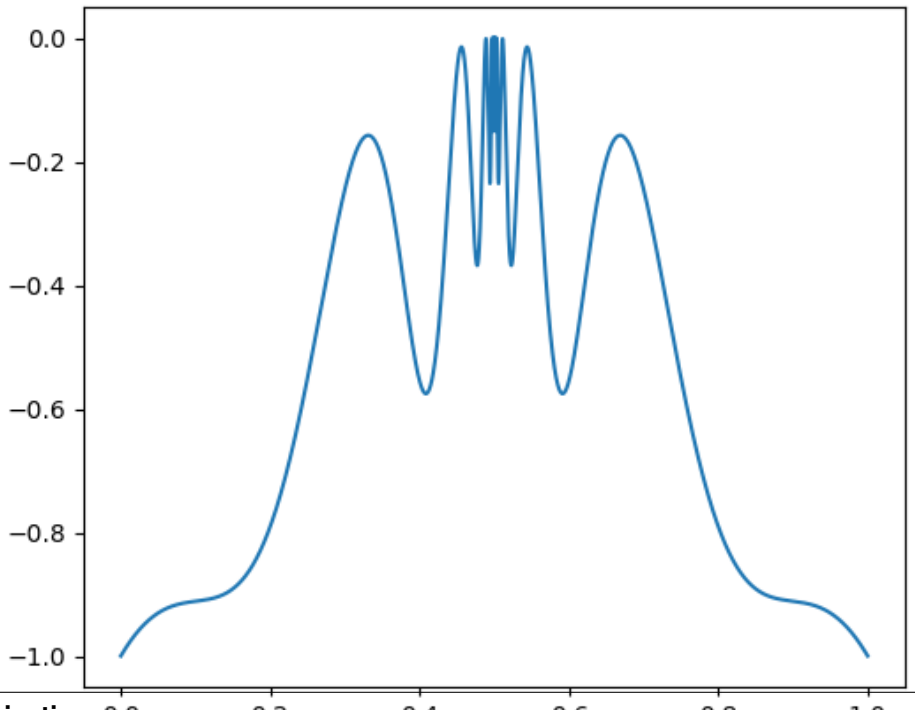
(continued from previous page)

```
algo.get_last_point()
```

2.5 Synthetic Objectives

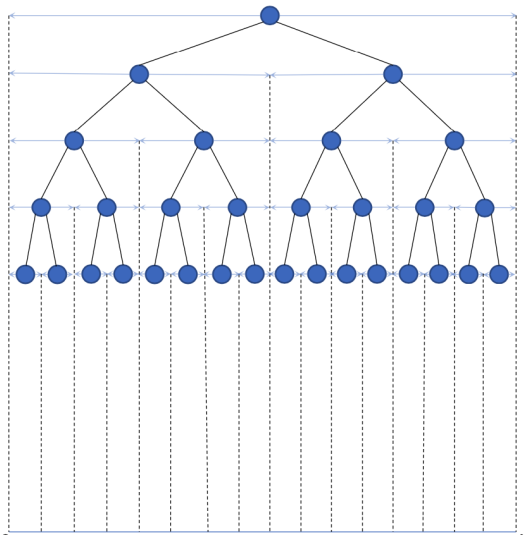
Synthetic objectives that can be used to evaluate the performance of X-armed bandit algorithms

Note: Some of these objectives can be found on Wikipedia

Objec-tives	Image
Garland	
Double-Sine	

2.6 Hierarchical Partition

We provide several choices of the hierarchical partition that separates the parameter space into multiple pieces.



Partition	Description
BinaryPartition	Equal-size binary partition of the parameter space, the split dimension is chosen uniform randomly
RandomBinaryPartition	Random-size binary partition of the parameter space, the split dimension is chosen uniform randomly
DimensionBinaryPartition	Equal-size partition of the space with a binary split on each dimension, the number of children of one node is 2^d
KaryPartition	Equal-size K-ary partition of the parameter space, the split dimension is chosen uniform randomly
RandomKaryPartition	Random-size K-ary partition of the parameter space, the split dimension is chosen uniform randomly

2.7 API Cheatsheet

We list all the important (abstract) functions from the base classes as follows.

2.7.1 Algorithm

- *PyXAB.algos.Algo.Algorithm.pull()*: Generate a point for each time step to be evaluated
- *PyXAB.algos.Algo.Algorithm.receive_reward()*: After receiving the reward, update the parameters of the algorithm
- *PyXAB.algos.Algo.Algorithm.get_last_point()*: The function to retrieve the last output of the algorithm

2.7.2 Partition

- *PyXAB.partition.Partition.Partition.make_children()*: Make children for one node

2.7.3 Objective

- *PyXAB.synthetic_obj.Objective.Objective.f()*: Evaluate the point and return the reward (stochastic or deterministic)

Note: The general base classes and the implemented functions are listed as follows

---

2.7.4 PyXAB.algos.Algo.Algorithm

Base class for all X-armed Bandit algorithms

class PyXAB.algos.Algo.**Algorithm**

Bases: abc.ABC

Abstract class for X-armed bandit algorithms.

abstract __init__()

Initialization for the algorithm

abstract get_last_point()

Every algorithm needs a function to get the last point

Returns **chosen_point** – The point chosen by the algorithm

Return type list

abstract pull(*time*)

Every algorithm needs a function to pull a node.

Parameters **time** (*int*) – The time step of the online process.

Returns **chosen_point** – The point chosen by the algorithm

Return type list

abstract receive_reward(*time*, *reward*)

Every algorithm needs a function to receive the reward.

Parameters

- **time** (*int*) – The time step of the online process.
- **reward** (*float*) – The (Stochastic) reward of the pulled point

2.7.5 PyXAB.synthetic_obj.Objective.Objective

Base class for any objective

class PyXAB.synthetic_obj.Objective.**Objective**

Bases: abc.ABC

Abstract class for general blackbox objectives

abstract f(*x*)

Evaluation of the chosen point

Parameters **x** (*list*) – one input point in the form of x = [x_1, x_2, … x_d]

Returns **y** – Evaluated value of the function at the particular point

Return type float

2.7.6 PyXAB.partition.Partition.Partition

Base class for any partition

class PyXAB.partition.Partition.**Partition**(*domain*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Bases: abc.ABC

Abstract class for partition of the parameter domain

__init__(*domain*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Initialization of the partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **node** – The node used in the partition, with the default choice to be P_node.

deepen()

The function to deepen the partition by one layer by making children to every node in the last layer

get_depth()

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

get_layer_node_list(*depth*)

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

get_node_list()

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

get_root()

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

abstract make_children(*parent*, *newlayer=False*)

The function to make children for the parent node

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

2.7.7 PyXAB.partition.Node.P_node

Base class for any node inside a partition

class PyXAB.partition.Node.**P_node**(*depth*, *index*, *parent*, *domain*)

Bases: object

The most basic node class that contains everything needed to be inside a partition

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the P_node class

Parameters

- **depth** (*int*) – The depth of the node
- **index** (*int*) – The index of the node
- **parent** – The parent node of the P_node
- **domain** (*list(list)*) – The domain that this P_node represents

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_parent()

The function to get the parent of the node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

2.8 API Reference

This documentation lists all the API reference for the implemented features in our package, including the algorithms, the synthetic functions, and the hierarchical partitions. Click on each of the class to see the functions implemented.

2.8.1 X-Armed Bandit Algorithms

The API references for X-armed bandit algorithms. Please see the general algorithm class in *API Cheatsheet*.

Zooming Algorithm

class PyXAB.algos.Zooming.**Zooming**(*nu=1*, *rho=0.9*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the Zooming algorithm

__init__(*nu=1*, *rho=0.9*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the Zooming algorithm

Parameters

- **nu** (*float*) – smoothness parameter nu of the Zooming algorithm
- **rho** (*float*) – smoothness parameter rho of the Zooming algorithm
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point of Zooming

Returns **chosen_point** – The point chosen by the algorithm

Return type list

make_active(*node*)

The function to make a node (an arm in the node) active

Parameters **node** – the node to be made active

pull(*time*)

The pull function of Zooming that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of Zooming to obtain the reward and update the statistics, then expand the active arms

Parameters

- **time** (*int*) – time stamp parameter
- **reward** (*float*) – the reward of the evaluation

Truncated HOO Algorithm

class PyXAB.algos.HOO.**HOO_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the HOO_node

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the HOO node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

compute_u_value(*nu*, *rho*, *rounds*)

The function to compute the u_{h,i} value of the node

Parameters

- **nu** (*float*) – parameter nu in the HOO algorithm
- **rho** (*float*) – parameter rho in the HOO algorithm
- **rounds** (*int*) – the number of rounds in the HOO algorithm

get_b_value()

The function to get the b_{h,i} value of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean reward of the node

get_parent()

The function to get the parent of the node

get_u_value()

The function to get the u_{h,i} value of the node

get_visited_times()

The function to get the number of visited times of the node

update_b_value(*b_value*)

The function to update the b_{h,i} value of the node

Parameters **b_value** (*float*) – The new b_{h,i} value to be updated

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.HOO.**T_HOO**(*nu=1*, *rho=0.5*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of the T_HOO algorithm

__init__(*nu=1*, *rho=0.5*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the T_HOO algorithm

Parameters

- **nu** (*float*) – parameter nu of the T_HOO algorithm
- **rho** (*float*) – parameter rho of the T_HOO algorithm
- **rounds** (*int*) – total number of rounds
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

expand(*parent*)

The function to expand the tree after pulling the parent node

Parameters **parent** – The parent node to be expanded

get_last_point()

The function to get the last point of HOO

Returns **chosen_point** – The point chosen by the algorithm

Return type list

optTraverse()

The function to traverse the exploration tree to find the best path and the best node to pull at this moment.

Returns

- **curr_node** (*Node*) – The last node selected by the algorithm
- **path** (*List of Node*) – The best path to traverse the partition selected by the algorithm

pull(*time*)

The pull function of T_HOO that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of T_HOO to obtain the reward and update the Statistics

Parameters

- **time** (*int*) – time stamp parameter
- **reward** (*float*) – the reward of the evaluation

updateAllTree(*path*, *reward*)

The function to update everything in the tree

Parameters

- **path** (*list*) – the path from the root to the chosen node
- **reward** (*float*) – the reward to update

updateBackwardTree()

The function to update all the b_{h,i} value backwards in the tree

updateRewardTree(*path*, *reward*)

The function to update the reward of each node in the path

Parameters

- **path** (*list*) – the path to find the best node
- **reward** (*float*) – the reward to update

updateUvalueTree()

The function to update the u_{h,i} value in the whole tree

DOO Algorithm

class PyXAB.algos.DOO.**DOO_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the node in the DOO algorithm

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the DOO node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

compute_b_value(*delta*)

The function to compute the b_{h,i} value of the node

Parameters **delta** (*float*) – The delta value in the b_{h,i} term, which depends on the depth of the node (Munos, 2011)

get_b_value()

The function to get the b_{h,i} value of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_parent()

The function to get the parent of the node

get_reward()

The function to get the reward of the node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward of the node

Parameters **reward** (*float*) – the reward for evaluating the node

visit()

The function to visit the node

class PyXAB.algos.DOO.**DOO**(*n=100*, *delta=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the DOO algorithm (Munos, 2011)

__init__(*n=100*, *delta=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the DOO algorithm

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **delta** (*function*) – The function to compute the delta value for each depth
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

delta_init(*h*)

The default delta function used in the algorithm (Munos, 2011)

Parameters **h** (*int*) – The depth parameter

Returns **max_value** – The delta value in that depth

Return type float

get_last_point()

The function to get the last point in DOO

Returns **point** – The output of the DOO algorithm at last

Return type list

pull(*time*)

The pull function of DOO that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of DOO to obtain the reward and update Statistics

Parameters

- **time** (*int*) – The time stamp parameter
- **reward** (*float*) – The reward of the evaluation

SOO Algorithm

class PyXAB.algos.SOO.**SOO_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the node in the SOO algorithm

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the SOO node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_parent()

The function to get the parent of the node

get_reward()

The function to get the reward of the node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward of the node

Parameters **reward** (*float*) – the reward for evaluating the node

visit()

The function to visit the node

class PyXAB.algos.SOO.**SOO**(*n=100*, *h_max=100*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the SOO algorithm (Munos, 2011)

__init__(*n=100*, *h_max=100*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the SOO algorithm

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **h_max** (*int*) – The largest searching depth
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point in SOO

Returns **point** – The output of the SOO algorithm at last

Return type list

pull(*time*)

The pull function of SOO that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of SOO to obtain the reward and update Statistics (for current node)

Parameters

- **time** (*int*) – The time stamp parameter
- **reward** (*float*) – The reward of the evaluation

StoSOO Algorithm

class PyXAB.algos.StoSOO.**StoSOO_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the node in the StoSOO algorithm

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the StoSOO node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

compute_b_value(*n*, *k*, *delta*)

The function to compute the b_{h,i} value of the node

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **k** (*int*) – The maximum number of pulls per node
- **delta** (*float*) – The confidence parameter

get_b_value()

The function to get the b_{h,i} value of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean reward of the node

get_parent()

The function to get the parent of the node

get_visited_times()

The function to get the number of visited times of the node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.StoSOO.**StoSOO**(*n=100*, *k=None*, *h_max=100*, *delta=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the StoSOO algorithm (Valko et al., 2013)

__init__(*n=100*, *k=None*, *h_max=100*, *delta=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the StoSOO algorithm

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **k** (*int*) – The maximum number of pulls per node
- **h_max** (*int*) – The maximum depth limit
- **delta** (*float*) – The confidence parameter delta
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point in StoSOO

Returns **point** – The output of the StoSOO algorithm at last

Return type list

pull(*time*)

The pull function of StoSOO that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of StoSOO to obtain the reward and update the Statistics

Parameters

- **time** (*int*) – The time stamp parameter
- **reward** (*float*) – the reward of the evaluation

HCT Algorithm

class PyXAB.algos.HCT.**HCT_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of HCT node

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the HCT node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

compute_u_value(*nu*, *rho*, *c*, *delta_tilde*)

The function to compute the u_{h,i} value of the node

Parameters

- **nu** (*float*) – parameter nu in the HOO algorithm
- **rho** (*float*) – parameter rho in the HOO algorithm
- **rounds** (*int*) – the number of rounds in the HOO algorithm

get_b_value()

The function to get the b_{h,i} value of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean reward of the node

get_parent()

The function to get the parent of the node

get_u_value()

The function to get the u_{h,i} value of the node

get_visited_times()

The function to get the number of visited times of the node

update_b_value(*b_value*)

The function to update the b_{h,i} value of the node

Parameters **b_value** (*float*) – The new b_{h,i} value to be updated

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.HCT.**HCT**(*nu=1*, *rho=0.5*, *c=0.1*, *delta=0.01*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of the HCT algorithm

__init__(*nu=1*, *rho=0.5*, *c=0.1*, *delta=0.01*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the HCT algorithm

Parameters

- **nu** (*float*) – parameter nu of the HCT algorithm
- **rho** (*float*) – parameter rho of the HCT algorithm
- **c** (*float*) – parameter c of the HCT algorithm
- **delta** (*float*) – confidence parameter delta of the HCT algorithm
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

expand(*parent*)

The function to expand the tree at the parent node

Parameters **parent** – The parent node to be expanded

get_last_point()

The function to get the last point of HCT

Returns **chosen_point** – The point chosen by the algorithm

Return type list

optTraverse()

The function to traverse the exploration tree to find the best path and the best node to pull at this moment.

Returns

- **curr_node** (*Node*) – The last node selected by the algorithm

- **path** (*List of Node*) – The best path to traverse the partition selected by the algorithm

pull(*time*)

The pull function of HCT that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of HCT to obtain the reward and update the Statistics

Parameters

- **time** (*int*) – time stamp parameter
- **reward** (*float*) – the reward of the evaluation

updateAllTree(*path*, *reward*)

The function to update everything in the tree

Parameters

- **path** (*list*) – the path from the root to the chosen node
- **reward** (*float*) – the reward to update

updateBackwardTree()

The function to update all the b_{h,i} value backwards in the tree

updateRewardTree(*path*, *reward*)

The function to update the reward of each node in the path

Parameters

- **path** (*list*) – the path to find the best node
- **reward** (*float*) – the reward to update

updateUvalueTree()

The function to update the u_{h,i} value in the whole tree

POO Algorithm

class PyXAB.algos.POO.**POO**(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*, *algo=None*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of the Parallel Optimistic Optimization (POO) algorithm (Grill et al., 2015), with the general definition in Shang et al., 2019.

__init__(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*, *algo=None*)

Parameters

- **numax** (*float*) – parameter nu_max in the algorithm
- **rhomax** (*float*) – parameter rho_max in the algorithm, the maximum rho used

- **rounds** (*int*) – the number of rounds/budget
- **domain** (*list(list)*) – the domain of the objective function
- **partition** – the partition used in the optimization process
- **algo** – the baseline algorithm used by the wrapper, such as T_HOO or HCT

get_last_point()

The function that returns the last point chosen by POO

pull(*time*)

The pull function of POO that returns a point to be evaluated

Parameters **time** (*int*) – The time step of the online process.

Returns **point** – The point chosen by the POO algorithm

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of POO to receive the reward for the chosen point

Parameters

- **time** (*int*) – The time step of the online process.
- **reward** (*float*) – The (Stochastic) reward of the pulled point

---

GPO Algorithm

class PyXAB.algos.GPO.**GPO**(*numax=1.0*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*, *algo=None*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of the General Parallel Optimization (GPO) algorithm (Shang et al., 2019)

__init__(*numax=1.0*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*, *algo=None*)

Initialization of the wrapper algorithm

Parameters

- **numax** (*float*) – parameter nu_max in the algorithm (default 1.0)
- **rhomax** (*float*) – parameter rho_max in the algorithm, the maximum rho used (default 0.9)
- **rounds** (*int*) – the number of rounds/budget (default 1000)
- **domain** (*list(list)*) – the domain of the objective function
- **partition** – the partition used in the optimization process
- **algo** – the baseline algorithm used by the wrapper, such as T_HOO or HCT

get_last_point()

The function to get the last point in GPO

pull(*time*)

The pull function of GPO that returns a point to be evaluated

Parameters **time** (*int*) – The time step of the online process.

Returns **point** – The point chosen by the GPO algorithm

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of GPO to receive the reward for the chosen point

Parameters

- **time** (*int*) – The time step of the online process.
- **reward** (*float*) – The (Stochastic) reward of the pulled point

---

PCT Algorithm

class PyXAB.algos.PCT.**PCT**(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of Parallel Confidence Tree (Shang et al., 2019) algorithm

__init__(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the PCT algorithm

Parameters

- **numax** (*float*) – parameter nu_max in the algorithm
- **rhomax** (*float*) – parameter rho_max in the algorithm, the maximum rho used
- **rounds** (*int*) – the number of rounds/budget
- **domain** (*list(list)*) – the domain of the objective function
- **partition** – the partition used in the optimization process

get_last_point()

The function to get the last point for PCT

pull(*time*)

The pull function of PCT that returns a point to be evaluated

Parameters **time** (*int*) – The time step of the online process.

Returns **point** – The point chosen by the PCT algorithm

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of PCT to receive the reward for the chosen point

Parameters

- **time** (*int*) – The time step of the online process.
- **reward** (*float*) – The (Stochastic) reward of the pulled point

SequOOL Algorithm

class PyXAB.algos.SequOOL.**SequOOL_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the SequOOL node

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the SequOOL node :param depth: fepth of the node :type depth: int :param index: index of the node :type index: int :param parent: parent node of the current node :param domain: domain that this node represents :type domain: list(list)

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_parent()

The function to get the parent of the node

get_reward()

The function to get the reward of the node

not_opened()

The function to get the status of the node (opened or not)

open()

The function to open a node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.SequOOL.**SequOOL**(*n=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the SequOOL algorithm (Barlett, 2019)

__init__(*n=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the SequOOL algorithm

Parameters

- **n** (*int*) – The totdal number of rounds (budget)
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point in SequOOL

Returns **point** – The output of the SequOOL algorithm at last

Return type list

static harmonic_series_sum(*n*)

A static method for computing the summation of harmonic series

Parameters **n** (*int*) – The number of terms in the summation

Returns **res** – The sum of the series

Return type float

pull(*t*)

The pull function of SequOOL that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*t*, *reward*)

The receive_reward function of SequOOL to obtain the reward and update Statistics

Parameters

- **t** (*int*) – The time stamp parameter
- **reward** (*float*) – The reward of the evaluation

StroquOOL Algorithm

class PyXAB.algos.StroquOOL.**StroquOOL_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the node in the StroquOOL algorithm

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the StroquOOL node

depth: int depth of the node

index: int index of the node

parent: parent node of the current node

domain: list(list) domain that this node represents

compute_mean_reward()

The function to compute the mean of the reward list of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean of the reward list of the node

get_parent()

The function to get the parent of the node

get_visited_times()

The function to get the number of visited times of the node

not_opened()

The function to get the status of the node (opened or not)

open_node()

The function to open a node

remove_reward()

The function to clear the reward list of a node

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.StroquOOL.**StroquOOL**(*n=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the StroquOOL algorithm (Bartlett, 2019)

__init__(*n=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the StroquOOL algorithm

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **domain** (*list(list)*) – The domain of the objective to be optimized

- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point in StroquOOL

Returns **point** – The output of the StroquOOL algorithm at last

Return type list

static harmonic_series_sum(*n*)

A static method for computing the summation of harmonic series

Parameters **n** (*int*) – The number of terms in the summation

Returns **res** – The sum of the series

Return type float

pull(*time*)

The pull function of StroquOOL that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

receive_reward(*t*, *reward*)

The receive_reward function of StroquOOL to obtain the reward and update Statistics. If the algorithm has ended but there is still time left, then this function just passes

Parameters

- **t** (*int*) – The time stamp parameter
- **reward** (*float*) – The reward of the evaluation

reset_p()

The function to reset p for current situation

VROOM Algorithm

class PyXAB.algos.VROOM.**VROOM_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of the node in the VROOM algorithm

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the VROOM node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

add_rank(*rank*)

The method to set the rank of the cell

Parameters **rank** (*int*) – the rank of the cell at current depth

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_eval_time()

The function to get the evaluation time of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean of the reward of the node

get_parent()

The function to get the parent of the node

get_rank()

The function to get the rank of the cell

Returns **rank** – the rank of the cell at current depth

Return type int

get_reward_tilde()

The function to get the reward tilde statistic of the node

sample_uniform()

The function to uniformly sample a point from the domain of the node

Returns **res** – the point sampled by the sampler

Return type list

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward of the node

Parameters **reward** (*float*) – the reward for evaluating the node

update_reward_tilde(*reward*)

The function to update the reward tilde of the node

Parameters **reward** (*float*) – the reward tilde statistc of the node

class PyXAB.algos.VROOM.**VROOM**(*n=100*, *h_max=100*, *b=None*, *f_max=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the VROOM algorithm (Ammar, Haitham, et al., 2020)

__init__(*n=100*, *h_max=100*, *b=None*, *f_max=None*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

The initialization of the VROOM algorithm

Parameters

- **n** (*int*) – The total number of rounds (budget)
- **h_max** (*int*) – The number bounds the depth of the searching tree
- **b** (*float*) – The parameter that measures the variation of the function
- **f_max** (*float*) – An upper bound of the objective function
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

get_last_point()

The function to get the last point in VROOM

Returns **point** – The output of the VROOM algorithm at last

Return type list

pull(*time*)

The pull function of VROOM that returns a point in every bound

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

rank(*nodes*)

The rank function of VROOM that rank nodes at the same depth

Parameters **nodes** (*list*) – a list of node at the same depth

receive_reward(*time*, *reward*)

The receive_reward function of VROOM to obtain the reward and update Statistics (for current node)

Parameters

- **time** (*int*) – The time stamp parameter
- **reward** (*float*) – The reward of the evaluation

VHCT Algorithm

class PyXAB.algos.VHCT.**VHCT_node**(*depth*, *index*, *parent*, *domain*)

Bases: *PyXAB.partition.Node.P_node*

Implementation of VHCT node

__init__(*depth*, *index*, *parent*, *domain*)

Initialization of the VHCT node

Parameters

- **depth** (*int*) – depth of the node
- **index** (*int*) – index of the node
- **parent** – parent node of the current node
- **domain** (*list(list)*) – domain that this node represents

compute_tau_hi_value(*nu*, *rho*, *c*, *bound*, *delta_tilde*)

The function to compute the threshold tau_hi value for the VHCT node

Parameters

- **nu** (*float*) – parameter nu of the VHCT algorithm
- **rho** (*float*) – parameter rho of the VHCT algorithm
- **c** (*float*) – parameter c of the VHCT algorithm
- **bound** (*float*) – parameter bound of the VHCT algorithm, the noise bound
- **delta_tilde** (*float*) – modified confidence parameter delta_tilde of the VHCT algorithm

compute_u_value(*nu*, *rho*, *c*, *bound*, *delta_tilde*)

The function to compute the u_{h,i} value of the node

Parameters

- **nu** (*float*) – parameter nu in the HOO algorithm
- **rho** (*float*) – parameter rho in the HOO algorithm
- **rounds** (*int*) – the number of rounds in the HOO algorithm

get_b_value()

The function to get the b_{h,i} value of the node

get_children()

The function to get the children of the node

get_cpoint()

The function to get the center point of the domain

get_depth()

The function to get the depth of the node

get_domain()

The function to get the domain of the node

get_index()

The function to get the index of the node

get_mean_reward()

The function to get the mean reward of the node

get_parent()

The function to get the parent of the node

get_tau_hi_value()

The function to get the tau_hi value of the node

get_u_value()

The function to get the u_{h,i} value of the node

get_visited_times()

The function to get the number of visited times of the node

update_b_value(*b_value*)

The function to update the b_{h,i} value of the node

Parameters **b_value** (*float*) – The new b_{h,i} value to be updated

update_children(*children*)

The function to update the children of the node

Parameters **children** – The children nodes to be updated

update_reward(*reward*)

The function to update the reward list of the node

Parameters **reward** (*float*) – the reward for evaluating the node

class PyXAB.algos.VHCT.**VHCT**(*nu=1*, *rho=0.5*, *c=0.1*, *delta=0.01*, *bound=1*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

The implementation of the Variance High Confidence Tree algorithm

__init__(*nu=1*, *rho=0.5*, *c=0.1*, *delta=0.01*, *bound=1*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the VHCT algorithm

Parameters

- **nu** (*float*) – parameter nu of the VHCT algorithm
- **rho** (*float*) – parameter rho of the VHCT algorithm
- **c** (*float*) – parameter c of the VHCT algorithm
- **delta** (*float*) – confidence parameter delta of the VHCT algorithm
- **bound** (*float*) – the noise upper bound parameter bound
- **domain** (*list(list)*) – The domain of the objective to be optimized
- **partition** – The partition choice of the algorithm

expand(*parent*)

The function to expand the tree at the parent node

Parameters **parent** – The parent node to be expanded

`get_last_point()`

The function to get the last point of HCT

Returns **chosen_point** – The point chosen by the algorithm

Return type list

`optTraverse()`

The function to traverse the exploration tree to find the best path and the best node to pull at this moment.

Returns

- **curr_node** (*Node*) – The last node selected by the algorithm
- **path** (*List of Node*) – The best path to traverse the partition selected by the algorithm

`pull(time)`

The pull function of VHCT that returns a point in every round

Parameters **time** (*int*) – time stamp parameter

Returns **point** – the point to be evaluated

Return type list

`receive_reward(time, reward)`

The receive_reward function of VHCT to obtain the reward and update the Statistics

Parameters

- **time** (*int*) – time stamp parameter
- **reward** (*float*) – the reward of the evaluation

`updateAllTree(path, reward)`

The function to update everything in the tree

Parameters

- **path** (*list*) – the path from the root to the chosen node
- **reward** (*float*) – the reward to update

`updateBackwardTree()`

The function to update all the b_{h,i} value backwards in the tree

`updateRewardTree(path, reward)`

The function to update the reward of each node in the path

Parameters

- **path** (*list*) – the path to find the best node
- **reward** (*float*) – the reward to update

`updateUvalueTree()`

The function to update the u_{h,i} value in the whole tree

VPCT Algorithm

class algos.VPCT.**VPCT**(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Bases: *PyXAB.algos.Algo.Algorithm*

Implementation of Variance-reduced Parallel Confidence Tree algorithm (VHCT + GPO)

__init__(*numax=1*, *rhomax=0.9*, *rounds=1000*, *domain=None*, *partition=<class 'PyXAB.partition.BinaryPartition.BinaryPartition'>*)

Initialization of the VPCT algorithm

Parameters

- **numax** (*float*) – parameter nu_max in the algorithm
- **rhomax** (*float*) – parameter rho_max in the algorithm, the maximum rho used
- **rounds** (*int*) – the number of rounds/budget
- **domain** (*list(list)*) – the domain of the objective function
- **partition** – the partition used in the optimization process

get_last_point()

The function to get the last point of VPCT.

pull(*time*)

The pull function of VPCT that returns a point to be evaluated

Parameters **time** (*int*) – The time step of the online process.

Returns **point** – The point chosen by the VPCT algorithm

Return type list

receive_reward(*time*, *reward*)

The receive_reward function of VPCT to receive the reward for the chosen point

Parameters

- **time** (*int*) – The time step of the online process.
- **reward** (*float*) – The (Stochastic) reward of the pulled point

2.8.2 Synthetic Objective Functions

The API references for synthetic objective functions. Please see the general objective class in *API Cheatsheet*.

Garland

class PyXAB.synthetic_obj.Garland.**Garland**

Bases: *PyXAB.synthetic_obj.Objective.Objective*

Garland objective implementation, with the domain [0, 1]

__init__()

Initialization with fmax = 1

f(*x*)

Evaluation of the chosen point in Garland function

Parameters **x** (*list*) – one input point in the form of x = [x1]

Returns **y** – Evaluated value of the function at the particular point x = [x1], returns x * (1 - x) * (4 - sqrt(abs(sin(60 * x))))

Return type float

DoubleSine

class PyXAB.synthetic_obj.DoubleSine.**DoubleSine**(*rho1=0.3*, *rho2=0.8*, *tmax=0.5*)

Bases: *PyXAB.synthetic_obj.Objective.Objective*

DoubleSine objective implementation, with the domain [0, 1]

__init__(*rho1=0.3*, *rho2=0.8*, *tmax=0.5*)

Initialization with fmax = 0

Parameters

- **rho1** (*float*) – The parameter rho1 between 0 and 1 to compute ep1
- **rho2** (*float*) – The parameter rho2 between 0 and 1 to compute ep2
- **tmax** (*float*) – The parameter tmax between 0 and 1 to truncate x

f(*x*)

Evaluation of the chosen point in DoubleSine function

Parameters **x** (*list*) – one input point in the form of x = [x1]

Returns **y** – Evaluated value of the function at the particular point x = [x1], returns mysin2(log(u, 2) / 2.0) * envelope_width - pow(u, self.ep2)

Return type float

HimmelBlau

class PyXAB.synthetic_obj.Himmelblau.**Himmelblau**

Bases: *PyXAB.synthetic_obj.Objective.Objective*

Himmelblau objective implementation, with the domain [-5, 5]^2

__init__()

Initialization with fmax = 0

f(*x*)

Evaluation of the chosen point in Himmelblau function

Parameters **x** (*list*) – one input point in the form of x = [x1, x2]

Returns **y** – Evaluated value of the function at the particular point x = [x1, x2], returns -((x1**2 + x2 - 11) ** 2) - (x1 + x2**2 - 7) ** 2

Return type float

Ackley

class PyXAB.synthetic_obj.Ackley.**Ackley**

Bases: *PyXAB.synthetic_obj.Objective.Objective*

Ackley objective implementation, with the domain [-1, 1]^2

__init__()

Initialization with fmax = 0

f(*x*)

Evaluation of the chosen point in Ackley function

Parameters **x** (*list*) – one input point in the form of x = [x1, x2]

Returns **y** – Evaluated value of the function at the particular point x = [x1, x2], returns 20 * exp(-0.2 * sqrt(0.5 * (x1**2 + x2**2))) + exp(0.5 * (cos(2 * pi * x1) + cos(2 * pi * x2))) - e - 20

Return type float

Rastrigin

class PyXAB.synthetic_obj.Rastrigin.**Rastrigin**

Bases: *PyXAB.synthetic_obj.Objective.Objective*

Rastrigin objective implementation, with the domain [-1, 1]^p

__init__()

Initialization with fmax = 0

f(*x*)

Evaluation of the chosen point in Rastrigin function

Parameters **x** (*list*) – one input point in the form of x = [x1, x2,...,xp]

Returns **y** – Evaluated value of the function at the particular point x = [x1, x2,...,xp], returns sum_i^p - 10 - (x[i] ** 2 - 10 * cos(2 * pi * x[i]))

Return type float

Difficult Function

class PyXAB.synthetic_obj.DifficultFunc.**DifficultFunc**

Bases: *PyXAB.synthetic_obj.Objective.Objective*

DifficultFunc objective implementation, with the domain [0, 1]

__init__()

Initialization with fmax = 1

f(*x*)

Evaluation of the chosen point in DifficultFunc function

Parameters **x** (*list*) – one input point in the form of x = [x1]

Returns **y** – Evaluated value of the function at the particular point x = [x1], returns threshold(log(y)) * (sqrt(y) - y**2) - sqrt(y)

Return type float

2.8.3 Hierarchical Partition

The API references for the hierarchical partition of the parameter space. Please see the general partition class in *API Cheatsheet*.

Binary Partition

class PyXAB.partition.BinaryPartition.**BinaryPartition**(*domain=None*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Bases: *PyXAB.partition.Partition.Partition*

Implementation of Binary Partition

__init__(*domain=None*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Initialization of the Binary Partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **node** – The node used in the partition, with the default choice to be P_node.

deepen()

The function to deepen the partition by one layer by making children to every node in the last layer

get_depth()

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

get_layer_node_list(*depth*)

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

get_node_list()

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

`get_root()`

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

`make_children(parent, newlayer=False)`

The function to make children for the parent node with a standard binary partition, i.e., split every parent node in the middle. If there are multiple dimensions, the dimension to split the parent is chosen randomly

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

---

Random Binary Partition

`class PyXAB.partition.RandomBinaryPartition.RandomBinaryPartition(domain=None, node=<class 'PyXAB.partition.Node.P_node'>)`

Bases: *PyXAB.partition.Partition.Partition*

Implementation of Random Binary Partition

`__init__(domain=None, node=<class 'PyXAB.partition.Node.P_node'>)`

Initialization of the Random Binary Partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **node** – The node used in the partition, with the default choice to be P_node.

`deepen()`

The function to deepen the partition by one layer by making children to every node in the last layer

`get_depth()`

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

`get_layer_node_list(depth)`

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

`get_node_list()`

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

`get_root()`

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

`make_children(parent, newlayer=False)`

The function to make children for the parent node with a random binary partition, i.e., split every parent node randomly into two children. If there are multiple dimensions, the dimension to split the parent is chosen randomly

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

---

Dimension-wise Binary Partition

`class PyXAB.partition.DimensionBinaryPartition.DimensionBinaryPartition(domain=None, node=<class 'PyXAB.partition.Node.P_node'>)`

Bases: *PyXAB.partition.Partition.Partition*

Implementation of Dimension-wise Binary Partition

`__init__(domain=None, node=<class 'PyXAB.partition.Node.P_node'>)`

Initialization of the Dimension-wise Binary Partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **node** – The node used in the partition, with the default choice to be P_node.

`deepen()`

The function to deepen the partition by one layer by making children to every node in the last layer

`get_depth()`

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

`get_layer_node_list(depth)`

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

get_node_list()

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

get_root()

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

make_children(*parent*, *newlayer=False*)

The function to make children for the parent node with a dimension-wise binary partition, i.e., split every parent node into 2^d children where d is the dimension of the parameter domain. Each dimension of the domain is split right in the middle.

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

---

Kary Partition

class PyXAB.partition.KaryPartition.**KaryPartition**(*domain=None*, *K=3*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Bases: *PyXAB.partition.Partition.Partition*

Implementation of K-ary Partition especially when K >= 3, i.e., Ternary, Quaternary, and so on

__init__(*domain=None*, *K=3*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Initialization of the K-ary Partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **K** (*int*) – The number of children of each parent, with the default choice to be 3
- **node** – The node used in the partition, with the default choice to be P_node.

deepen()

The function to deepen the partition by one layer by making children to every node in the last layer

get_depth()

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

get_layer_node_list(*depth*)

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

get_node_list()

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

get_root()

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

make_children(*parent*, *newlayer=False*)

The function to make children for the parent node with a standard K-ary partition, i.e., split every parent node into K children nodes of the same size. If there are multiple dimensions, the dimension to split the parent is chosen randomly

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

---

RandomKary Partition

class PyXAB.partition.RandomKaryPartition.**RandomKaryPartition**(*domain=None*, *K=3*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Bases: *PyXAB.partition.Partition.Partition*

Implementation of Random K-ary Partition especially when K >= 3, i.e., Ternary, Quaternary, and so on

__init__(*domain=None*, *K=3*, *node=<class 'PyXAB.partition.Node.P_node'>*)

Initialization of the Random K-ary Partition

Parameters

- **domain** (*list(list)*) – The domain of the objective function to be optimized, should be in the form of list of lists (hypercubes), i.e., [[range1], [range2], … [range_d]], where [range_i] is a list indicating the domain's projection on the i-th dimension, e.g., [-1, 1]
- **K** (*int*) – The number of children of each parent
- **node** – The node used in the partition, with the default choice to be P_node.

deepen()

The function to deepen the partition by one layer by making children to every node in the last layer

`get_depth()`

The function to get the depth of the partition

Returns **depth** – The depth of the partition

Return type int

`get_layer_node_list(depth)`

The function to get the all the nodes on the specified depth

Parameters **depth** (*int*) – The depth of the layer in the partition

Returns **self.node_list** – The list of nodes on the specified depth

Return type list

`get_node_list()`

The function to get the list all nodes in the partition

Returns **self.node_list** – The list of all nodes

Return type list

`get_root()`

The function to get the root of the partition

Returns The root node of the partition

Return type self.root

`make_children(parent, newlayer=False)`

The function to make children for the parent node with a random K-ary partition, i.e., split every parent node into K children nodes of different sizes. If there are multiple dimensions, the dimension to split the parent is chosen randomly

Parameters

- **parent** – The parent node to be expanded into children nodes
- **newlayer** (*bool*) – Boolean variable that indicates whether or not a new layer is created

2.9 Contributing

We appreciate all forms of help and contributions, including but not limited to

- Star and watch our project
- Open an issue for any bugs you find or features you want to add to our library
- Fork our project and submit a pull request with your valuable codes

Note: We have some TODOs listed in the Roadmap that we need help with.

2.9.1 To Implement New Features

Note: Please submit all pull requests to the `dev` branch instead of the `main` branch

Before Implementation

Please read the *API Cheatsheet* and *API Reference* for the API of our implemented classes and the abstract methods that need to be implemented when creating a new algorithm/partition/objective/node.

During Implementation

Please carefully follow our API and the *Usage Examples*. For example, every algorithm needs to inherit the class *PyXAB.algos.Algo.Algorithm*, and has to implement the abstract methods *PyXAB.algos.Algo.Algorithm.pull()*, *PyXAB.algos.Algo.Algorithm.receive_reward()* and *PyXAB.algos.Algo.Algorithm.get_last_point()*.

Documentations

We do not ask for detailed documentations, but if it is possible, please add some comments and documentations for your implemented functions/classes, following the numpy docstring style.

Testing and Debug

After implementation, please test your algorithm by running it on some of our synthetic objectives for debugging and improvements and write a test_xxx.py file.

Final Check

Before submitting the pull request, please make sure you have the following files ready

```
xxx.py
test_xxx.py
```

2.9.2 Optional Steps

Note: The following steps are optional but highly recommended

Black CodeStyle

In PyXAB, we follow the black codestyle. See more details on webpage of black and our issue. To convert your code, simply follow the instructions below.

First, run the following lines of code to install black

```
python -m pip install --upgrade pip
python -m pip install black
```

After implementing your own classes with documentations, run the following lines to change your code style

```
black PyXAB
python -m black PyXAB #if the above line does not work
```

Local Testing and Coverage

First, run the following lines of code to install pytest and coverage

```
python -m pip install --upgrade pip
python -m pip install pytest==7.1.2
python -m pip install coverage
```

To obtain the testing results and the code coverage report, run the following lines

```
coverage run --source=PyXAB -m pytest
coverage report
```

To see which lines are not covered by the tests, run the following lines

```
coverage run --source=PyXAB -m pytest
coverage report -m
```

2.10 Contributing Examples

Below are examples of how to contribute to the PyXAB package

2.10.1 New Objective

An example to implement a new blackbox objective for any PyXAB algorithm to optimize. First import all the useful packages

```
from PyXAB.synthetic_obj.Objective import Objective
from PyXAB.algos.HOO import T_HOO
from PyXAB.partition.BinaryPartition import BinaryPartition
```

We have already shown a `Sine` function example in the general instructions. Here, we give another very simple example. Suppose the objective is only a constant, i.e., `f(x) = 1` everywhere, then the class definition would be as simple as.

```
class Constant_1(Objective):
    def __init__(self):
        self.fmax = 1

    def f(self, x):
        return 1
```

The inheritance of the `Objective` class is unnecessary (but highly recommended for consistency). E.g., another possible definition could be

```
class Constant_2():

    def evaluate(self, x):
        return 1
```

Let us suppose the domain for this objective is `[0, 10]`, and we use the binary partition for the domain and the HOO algorithm to optimize the objectives

```
T = 100
target1 = Constant_1()
target2 = Constant_2()
domain = [[0, 10]]
partition = BinaryPartition
algo = T_HOO(domain=domain, partition=partition)
```

Now as can be seen below, the objectives are ready to be optimized

```
# Optimize Objective Constant_1
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target1.f(point)
    algo.receive_reward(t, reward)


# Optimize Objective Constant_2
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target2.evaluate(point)
    algo.receive_reward(t, reward)
```

Total running time of the script: (0 minutes 0.273 seconds)

2.10.2 New Algorithm

A (dummy) example to implement a new PyXAB algorithm. First import all the useful packages

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.Algo import Algorithm
from PyXAB.partition.BinaryPartition import BinaryPartition
import numpy as np
```

Now let us suppose that we want to implement a new (dummy) algorithm that always select all nodes on each layer and then evaluate them for a number of times

```
class Dummy(Algorithm):
    def __init__(self, evaluation=5, rounds=1000, domain=None, partition=None):
        super(Dummy, self).__init__()
        if domain is None:
            raise ValueError("Parameter space is not given.")
        if partition is None:
            raise ValueError("Partition of the parameter space is not given.")
        self.partition = partition(domain=domain)
        self.iteration = 0
        self.evaluation = evaluation
        self.rounds = rounds
        self.partition.deepen()

        self.iterator = 0
```

(continues on next page)

(continued from previous page)

```
    # we need to re-write the pull function
    def pull(self, time):

        depth = self.partition.get_depth()
        index = self.iterator // self.evaluation
        nodes = self.partition.get_node_list()
        point = nodes[depth][index].get_cpoint()
        self.iterator += 1                  # get a point and increase the iterator
        if self.iterator >= self.evaluation * len(nodes[depth]):     # If every point is␣
↪evaluated, deepen the partition
            self.partition.deepen()

        return point

    # we need to re-write the receive_reward function
    def receive_reward(self, time, reward):
        # No update given the reward for the dummy algorithm
        pass

    def get_last_point(self):

        return self.pull(0)
```

Define the number of rounds, the target, the domain, the partition, and the algorithm for the learning process

```
T = 100
target = Garland()
domain = [[0, 1]]
partition = BinaryPartition
algo = Dummy(rounds=T, domain=domain, partition=partition)        # The new algorithm
```

As shown below, the Dummy algorithm now optimizes the objective

```
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target.f(point) + np.random.uniform(-0.1, 0.1)     # uniform noise
    algo.receive_reward(t, reward)
    #print(point)
```

Total running time of the script: (0 minutes 0.004 seconds)

2.10.3 New Partition

An example to implement a new partition of the space for any PyXAB algorithm. First import all the useful packages

```
from PyXAB.synthetic_obj.Garland import Garland
from PyXAB.algos.HOO import T_HOO
from PyXAB.partition.Partition import Partition
from PyXAB.partition.Node import P_node
import numpy as np
```

Now let us suppose that we want to implement a new binary partition that always split the domain into two nodes that are 1/3 and 2/3 of its original size, i.e., if the original projection on the chosen dimension is `[a, b]`, we split the domain into `[a, 0.67a + 0.33b]` and `[0.67a + 0.33b, b]`.

```
class NewBinaryPartition(Partition):

    def __init__(self, domain, node=P_node):

        super(NewBinaryPartition, self).__init__(domain=domain, node=node)

    # We rewrite the make_chilren function for the new partition
    def make_children(self, parent, newlayer=False):

        parent_domain = parent.get_domain()
        dim = np.random.randint(0, len(parent_domain))
        selected_dim = parent_domain[dim]

        domain1 = parent_domain.copy()
        domain2 = parent_domain.copy()

        # New choice of the split point
        split_point = 2/3 * selected_dim[0] + 1/3 * selected_dim[1]             # split␣
↪point
        domain1[dim] = [selected_dim[0], split_point]
        domain2[dim] = [split_point, selected_dim[1]]

        # Initialization of the two new nodes
        node1 = self.node(
            depth=parent.get_depth() + 1,
            index=2 * parent.get_index() - 1,
            parent=parent,
            domain=domain1,
        )
        node2 = self.node(
            depth=parent.get_depth() + 1,
            index=2 * parent.get_index(),
            parent=parent,
            domain=domain2,
        )

        # Update the children of the parent
        parent.update_children([node1, node2])

        new_deepest = []
```

(continues on next page)

(continued from previous page)

```
        new_deepest.append(node1)
        new_deepest.append(node2)

        # If creating a new layer, use the new nodes as the first nodes in the new layer
        if newlayer:
            self.node_list.append(new_deepest)
            self.depth += 1
        # Else, just append the new nodes to the existing layer
        else:
            self.node_list[parent.get_depth() + 1] += new_deepest
```

Define the number of rounds, the target, the domain, the partition, and the algorithm for the learning process

```
T = 100
target = Garland()
domain = [[0, 1]]
partition = NewBinaryPartition                          # the new partition chosen is␣
↪NewBinaryPartition
algo = T_HOO(domain=domain, partition=partition)
```

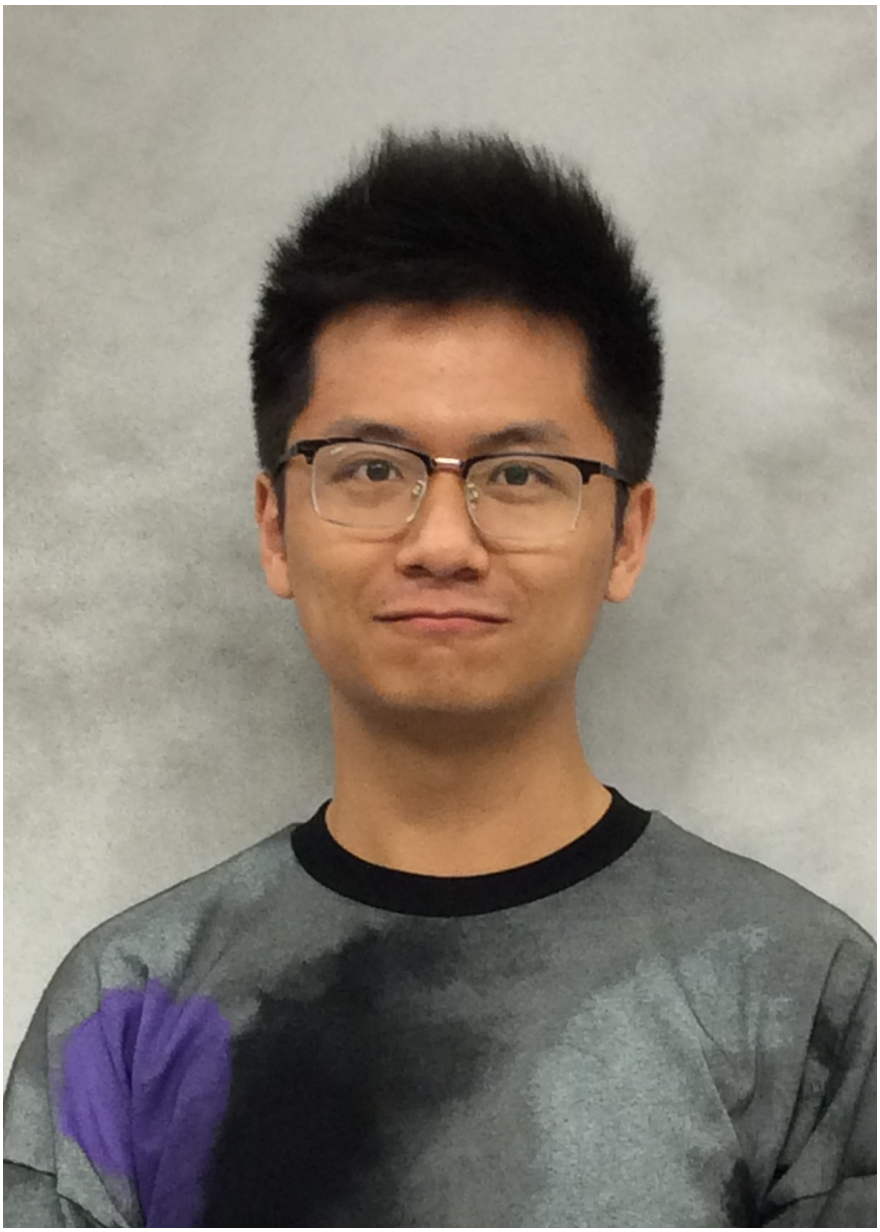
As shown below, the partition should be working

```
for t in range(1, T+1):

    point = algo.pull(t)
    reward = target.f(point) + np.random.uniform(-0.1, 0.1)      # uniform noise
    algo.receive_reward(t, reward)
```

Total running time of the script: (0 minutes 0.093 seconds)

2.11 PyXAB Development Team

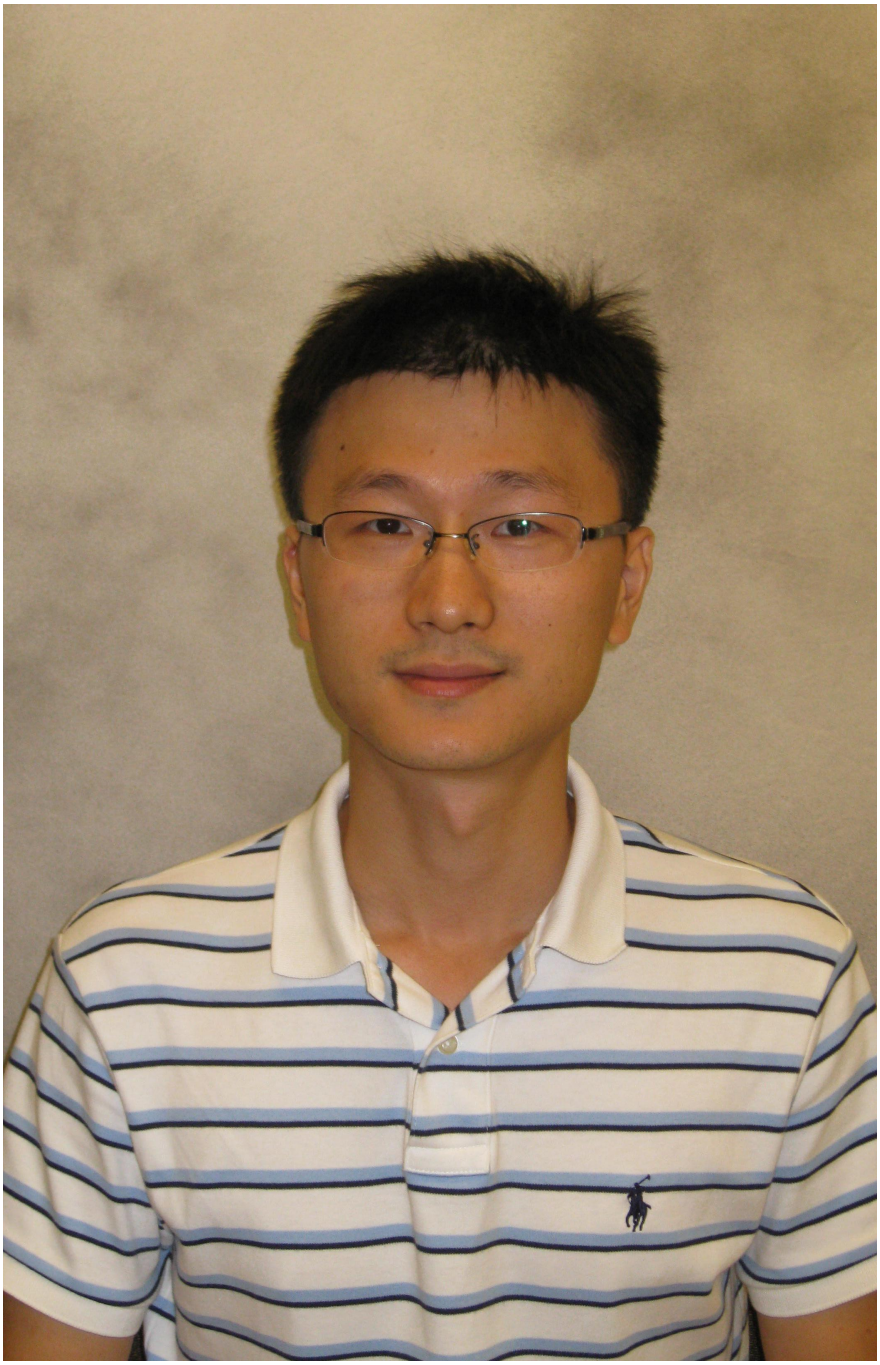
2.11.1 Coding



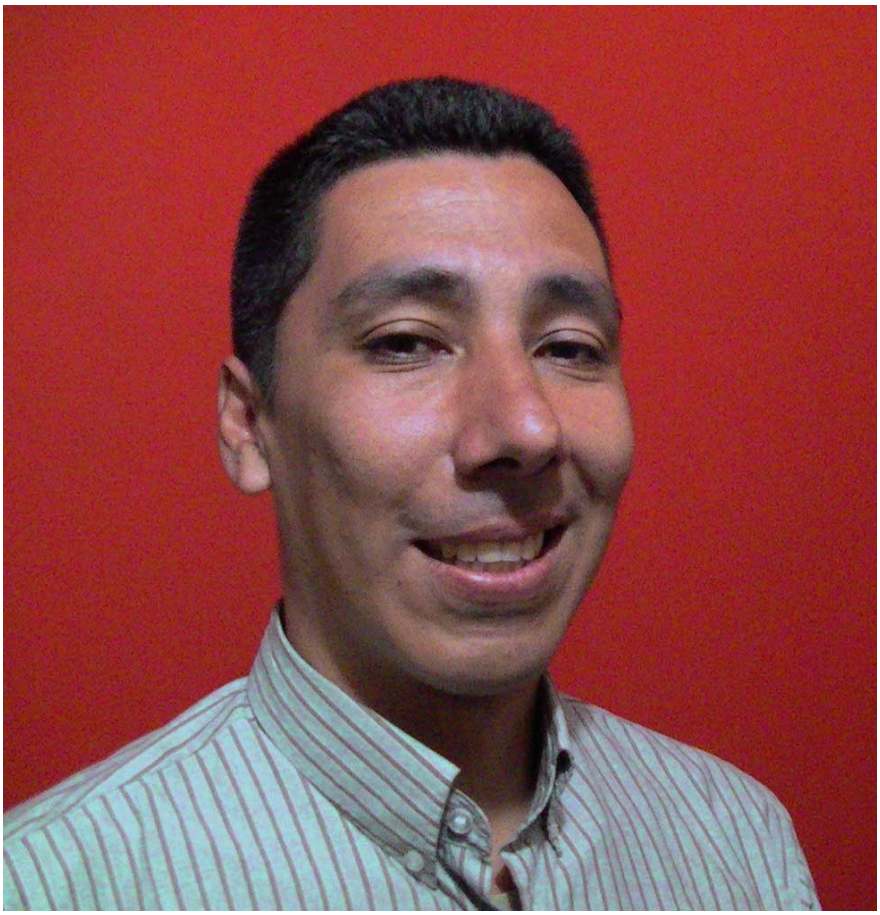
Mr. Wenjie Li is a Ph.D. Candidate at Purdue University. Mr. Wenjie Li has obtained an MS degree in Computer Science and Statistics during Ph.D. study. Mr. Wenjie Li received a B.Sc. in Mathematics from Chinese University of Hong Kong, with double stream in Computational Applied Mathematics (CAM) and Enrichment Mathematics.

Mr. Haoze Li is a Ph.D. Candidate at Purdue University. Mr. Haoze Li received a B.Sc. in Statistics from Peking University

2.11.2 Advisory



Prof. Qifan Song is an Associate Professor at Purdue University. Prof. Qifan Song obtained a doctoral degree in statistics from Texas A&M University in 2014 under the supervision of Dr. Faming Liang (who currently is a Distinguished Professor of Purdue University). Before that, Prof. Qifan Song obtained a bachelor's degree in probability and statistics from Peking University, China in 2009.



Prof. Jean Honorio is an Assistant Professor at Purdue University. Prior to joining Purdue, Prof. Jean Honorio was a postdoctoral associate at MIT CSAIL, working with Tommi Jaakkola

INDEX

Symbols

A

B

C

H

K

M

N

O

P

R

S

T

U

V

Z