# FLEXS

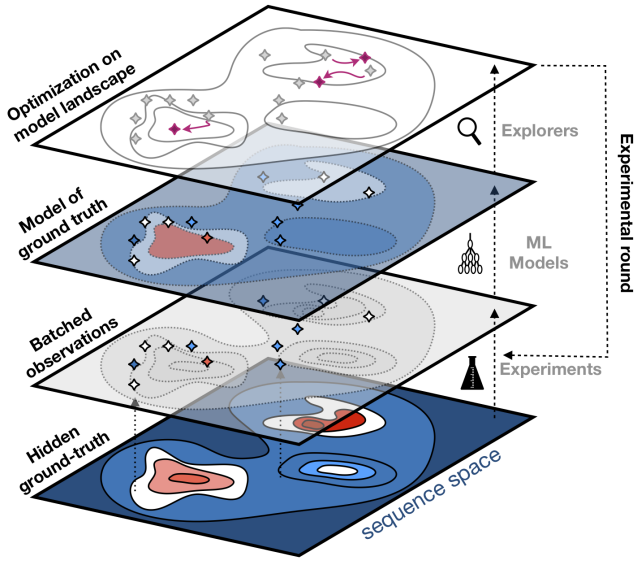
**Sam Sinai, Richard Wang, Alexander Whatley, Elina Locane, Stew**

**Oct 06, 2020**

# CONTENTS

# Fitness Landscape EXploration Sandbox

*for model-guided biological sequence design*




FLEXS is an open-source simulation environment that enables you to develop and compare model-guided biological sequence design algorithms. This project was developed with support from Dyno Therapeutics.

CHAPTER

ONE

# INSTALLATION

FLEXS is available on PyPI and can be installed with `pip install flexs`.

There are two optional, but very useful dependencies, ViennaRNA (for RNA binding landscapes) and PyRosetta (for protein design landscapes). These can both be installed with conda:

```
$ conda install -c bioconda viennarna
$ conda install pyrosetta  # Set up RosettaCommons conda channel first (http://www.
↪pyrosetta.org/dow)
```

Note that PyRosetta requires a commercial license if not being used for academic purposes.

If contributing or running paper code/experiments, we recommend that you install the dependencies for the sandbox in a conda virtual environment. You can update an existing conda environment with `conda env update --name {name} --file environment.yml` or initialize a new one with `conda env create -f environment.yml`. Then install the local version of `flexs` with `pip install -e .` in the root directory.

CHAPTER

TWO

# OVERVIEW

Biological sequence design through machine-guided directed evolution has been of increasing interest. This process often involves two closely connected steps:

- Models `f` that attempt to learn the ground truth sequence to function relationship `g(x) = y`.
- Algorithms that explore the sequence space with the help of the trained model `f`.

  While in some cases, these two steps are learned simultaneously, it is fairly common to have access to a well-trained model `f` which is *not* invertible. Namely, given a sequence `x`, the model can estimate `y'` (with variable accuracy), but it cannot generate a sequence `x'` associated with a specific function `y`. Therefore it is valuable to develop exploration algorithms `E(f)` that make use of the model `f` to propose sequences `x'`.

  We implement a simulation environment that allows you to develop or port landscape exploration algorithms for a variety of challenging tasks. Our environment allows you to abstract away the model `f = Noisy_abstract_model(g)` or employ empirical models (like Keras/Pytorch or Sklearn models). You can see how these work in the **tutorial**.

Our abstraction is comprised of four levels:

## 2.1 1. Fitness Landscapes

These oracles `g` are simulators that are assumed as ground truth, i.e. when queried, they return the true value `y_i` associated with a sequence `x_i`. Currently we have four classes of ground truth oracles implemented.

- **Transcription factor binding data**. This is comprised of 158 (experimentally) fully characterized landscapes.
- **RNA landscapes**. A set of curated and increasingly challenging RNA binding landscapes as simulated with ViennaRNA.
- **AAV Additive Tropism**. A hypothesized noisy additive protein landscape based on tissue tropism of single mutant AAV2 capsid protein.
- **GFP fluorescence**. Fluorescence of GFP protein as predicted by TAPE transformer model.
- **Rosetta-based design**. Rosetta-based design task for 3MSI anti-freeze protein.

For all landscapes we also provide a fixed set of initial points with different degrees of previous optimization, so that the relative strength of algorithms when starting from locations near or far away from peaks can be evaluated.

## 2.2 2. Noisy oracles

Noisy oracles are (approximate) models `f` of the original ground truth landscape `g`. These allow for the exploration algorithm to screen sequences virtually, before committing to making expensive queries to `g`. We implement two flavors of these

- Noisy abstract models: Noise corrupted version of `g` (this allows for independent study of exploration algorithms).
- Empirical models: `f` is learned directly from the data that was collected so far.

## 2.3 3. Exploration algorithms

> Exploration algorithms have access to `f` with some limit on the number of queries to this oracle `virtual_screen`. Once they have queried that many samples, they would commit to measuring `batch_size` from the ground truth, which incurrs a real cost. The class `base_explorer` implements the housekeeping tasks, and new exploration algorithms can be implemented by inheriting from it.

## 2.4 4. Evaluators

We also implement a suite of evaluation modules that automatically collect data that is necessary for evaluating algorithms on different performance criteria.

- *robustness*: Produces data for analyzing how explorer performance changes given different quality of models.
- *efficiency*: Produces data for analyzing how explorer performance changes when more computational evaluations are allowed.
- *adaptivity*: Produces data for analyzing how the explorer is sensitive to the number of batches it is allowed to sample, given a fixed total budget.

See the tutorial for an example of how these can be run.

CHAPTER

# THREE

# CONTRIBUTIONS AND CREDITS

Your PR and contributions to this sandbox are most welcome. If you make use of data or algorithms in this sandbox, please ensure that you cite the relevant original articles upon which this work was made possible (we provide links in this readme). To cite the sandbox itself:

```
@article{sinai2020adalead,
  title={AdaLead: A simple and robust adaptive greedy search algorithm for sequence␣
↪design},
  author={Sinai, Sam and Wang, Richard and Whatley, Alexander and Slocum, Stewart and␣
↪Locane, Elina and Kelsic, Eric},
  journal={arXiv preprint},
  year={2020}
}
```

FLEXS 0.2.1 was developed by Sam Sinai, Richard Wang, Alexander Whatley, Elina Locane, and Stewart Slocum.

CHAPTER

FOUR

# COMPONENTS

## 4.1 Transcription Factor Binding

Barrera et al. (2016) surveyed the binding affinity of more than one hundred and fifty transcription factors (TF) to all possible DNA sequences of length 8. Since the ground truth is entirely characterized, and biological, it is a relevant benchmark for our purpose. These generate the full picture for landscapes of size `4^8`. We shift the function distribution such that `y` is within `[0,1]`, and therefore `optimal(y)=1`. We also provide 15 initiation sequences with different degrees of optimization across landscapes. The sequence `TTAATTAA` for instance is a famous binding site that is a global peak in 20 of these landscapes, and a local peak (above all its single mutant neighbors) in 96 landscapes overall. `GCTCGAGC` is a local peak in 106 landscapes, whereas `AAAGAGAG` is not a peak in any of the 158 landscapes. It is notable that while complete, these landscapes are generally easy to optimize on due to their size. So we recommend that they are tested in very low-budget setting or additional classes of landscapes are used for benchmarking.

```
@article{barrera2016survey,
  title={Survey of variation in human transcription factors reveals prevalent DNA␣
↪binding changes},
  author={Barrera, Luis A and Vedenko, Anastasia and Kurland, Jesse V and Rogers,␣
↪Julia M and Gisselbrecht, Stephen S and Rossin, Elizabeth J and Woodard, Jaie and␣
↪Mariani, Luca and Kock, Kian Hong and Inukai, Sachi and others},
  journal={Science},
  volume={351},
  number={6280},
  pages={1450--1454},
  year={2016},
  publisher={American Association for the Advancement of Science}
}
```

## 4.2 RNA Landscapes

Predicting RNA secondary structures is a well-studied problem. There are efficient and accurate dynamic programming approaches to calculates secondary structure of short RNA sequences. These landscapes give us a good proxy for a consistent oracle over entire domain of large landscapes. We use the ViennaRNA package to simulate binding landscapes of RNA sequences as a ground truth oracle.

Our sandbox allows for constructing arbitrarily complex landscapes (although we discourage large RNA sequences as the accuracy of the simulator deteriorates above 200 nucleotides). As benchmark, we provide a series of 36 increasingly complex RNA binding landscapes. These landscapes each come with at least 5 suggested starting sequences, with various initial optimization.

The simplest landscapes are binding landscapes with a single hidden target (often larger than the design sequence resulting in multiple peaks). The designed sequences is meant to be optimized to bind the target with the minimum binding energy (we use duplex energy as our objective). We estimate `optimal(y)` by computing the binding energy of the perfect complement of the target and normalize the fitnesses using that measure (hence this is only an approximation and often a slight underestimate). RNA landscapes show many local peaks, and often multiple global peaks due to symmetry.

Additionally, we construct more complex landscapes by increasing the number of hidden targets, enforcing specific conservation patterns, and composing the scores of each landscapes multiplicatively. See multi-dimensional models for the generic class that allows composing landscapes.

```
@article{lorenz2011viennarna,
  title={{ViennaRNA} Package 2.0},
  author={Lorenz, Ronny and Bernhart, Stephan H and Zu Siederdissen, Christian H{\"o}
↪ner and Tafer, Hakim and Flamm, Christoph and Stadler, Peter F and Hofacker, Ivo L},
  journal={Algorithms for molecular biology},
  volume={6},
  number={1},
  pages={26},
  year={2011},
  publisher={Springer}
}
```

## 4.3 Additive AAV landscapes

Ogden et al. (2019) perform a comprehensive single mutation scan of AAV2 capsid protein, assaying tropism for five different target tissues. The authors show that an additive model is informative about the local structure of the landscape. Here we use the data from the single mutations to generate a toy additive model. Here `y' := sum(s_i)+ e`, where `i` indicates the position across the sequences, and `s_i` indicates a sequence with mutation `s` at position `i` and `e` indicates iid Gaussian noise. This construct is also known as "Rough Mt. Fuji" (RMF) and many empirical fitness landscapes are consistent with an RMF local structure around viable natural sequences with unpredictable regions in between. In the noise-free setting, the RMF landscape is convex with a single peak. We allow the construction of multiple target tissues, and different design lengths (tasks ranging from desiging short region of the protein to tasks that encompass designing the full protein). The scores are normalized between `[0,1]`.

```
@article{ogden2019comprehensive,
  title={Comprehensive AAV capsid fitness landscape reveals a viral gene and enables␣
↪machine-guided design},
  author={Ogden, Pierce J and Kelsic, Eric D and Sinai, Sam and Church, George M},
  journal={Science},
  volume={366},
  number={6469},
  pages={1139--1143},
  year={2019},
  publisher={American Association for the Advancement of Science}
}
```

## 4.4 GFP

In TAPE, the authors benchmark multiple machine learning methods on a set of tasks including GFP fluorescence prediction. The GFP task is comprised of training and predicting fluorescence values on approximately 52,000 protein sequences of length 238 which are derived from the naturally occurring GFP in *Aequorea victoria* (See this paper). Downloading and doing inference with this model is memory and time intensive. These landscapes are not normalized and therefore scores higher than 1 are possible (we do not know the maximum activation for the model).

```
@inproceedings{tape2019,
    author = {Rao, Roshan and Bhattacharya, Nicholas and Thomas, Neil and Duan, Yan␣
↪and Chen, Xi and Canny, John and Abbeel, Pieter and Song, Yun S},
    title = {Evaluating Protein Transfer Learning with TAPE},
    booktitle = {Advances in Neural Information Processing Systems}
    year = {2019}
}

@article{sarkisyan2016local,
  title={Local fitness landscape of the green fluorescent protein},
  author={Sarkisyan, Karen S and Bolotin, Dmitry A and Meer, Margarita V and Usmanova,
↪ Dinara R and Mishin, Alexander S and Sharonov, George V and Ivankov, Dmitry N and␣
↪Bozhanova, Nina G and Baranov, Mikhail S and Soylemez, Onuralp and others},
  journal={Nature},
  volume={533},
  number={7603},
  pages={397--401},
  year={2016},
  publisher={Nature Publishing Group}
}
```

## 4.5 Rosetta-based Design

Rosetta is a protein modeling software suite used for *de novo* design and structure prediction. Based on the principle that structure determines function, the Rosetta design process begins with a desired 3-D protein conformation and tries to find amino acid sequences that are likely to fold to that structure. While the dynamics of protein folding are still poorly understood, this approach has proven remarkably effective in practice, and so we find it an acceptable analogue to the true fitness landscape. To keep our experiments computationally feasible, we omit the expensive step of side-chain packing and use the simplified centroid scoring frounction as our objective. We use the PyRosetta Python interface to Rosetta. The Rosetta design objective function is a scaled estimate of the folding energy, which has been found to be an indicator of the probability that a sequence will fold to the desired structure. As an example, we provide an optimization challenge for the structure of 3MSI, a 66 amino acid antifreeze protein found in the ocean pout starting from 5 sequences with 3-27 mutations from the wildtype. Here, we normalize energy scores by scaling and shifting their distribution and then applying the sigmoid function.

```
@article{chaudhury2010pyrosetta,
  title={PyRosetta: a script-based interface for implementing molecular modeling␣
↪algorithms using Rosetta},
  author={Chaudhury, Sidhartha and Lyskov, Sergey and Gray, Jeffrey J},
  journal={Bioinformatics},
  volume={26},
  number={5},
  pages={689--691},
  year={2010},
```

(continues on next page)

(continued from previous page)

```
  publisher={Oxford University Press}
}
```

### 4.5.1 Noisy Oracles

## 4.6 Noisy Abstract Models

These models get access to the ground truth `g`, but do not allow the explorer to access `g` directly. They corrupt the signal from `g` but adding noise to it, proportional to the distance of the query from the (nearest) observed data. The parameter `signal_strength` which is between 0 (no signal) and 1 (perfect model) determines the rate of decay.

## 4.7 Empirical Models

These models train a standard algorithm on the observed data. Some baseline models can be found in https://github.com/samsinai/FLEXS/blob/master/flexs/baselines/models. All landscapes and models can also be ensembled using the ensemble class. Ensembles also have the ability to be *adaptive* i.e. the models within an ensemble will be reweighted based on their accuracy on the last measured set.

### 4.7.1 Exploration Algorithms

## 4.8 Bring your own explorer

Exploration algorithms are search methods that use noisy oracles to select the next batch of samples from the landscape. This is the main service of this sandbox, you can implement your own explorer by simply inheriting from the Base Explorer, and implementing a single method:

```
class MyExplorer(flexs.Explorer):
    """Your explorer here"""
    def __init__(self,
                 model,
                 rounds,
                 starting_sequence,
                 sequences_batch_size,
                 model_queries_per_batch,
                 **kwargs):

        name = f"MyExplorer_{**kwargs}"
        super().__init__(
            model,
            name,
            rounds,
            sequences_batch_size,
            model_queries_per_batch,
            starting_sequence,
        )
        "Your custom attributes here"

    def propose_sequences(self, measured_sequences_data):
```

(continues on next page)

(continued from previous page)

```
        """
        Your method implementation overriding the main explorer.
        It is allowed to make *model_queries_per_batch* queries to the model
        and make *sequences_batch_size* proposals in return.
        """

        return sequences, scores
```

## 4.9 Baseline Explorers

- Random Explorer: A baseline random explorer.

## 4.10 Evolutionary Algorithms

- Naive Genetic Algorithm, Wright-Fisher: A standard Wright-Fisher process that has access to an oracle for pre-screening.
- CMA-ES: The CMA-ES algorithm that optimizes a continuous relaxation of one-hot vectors encoding sequences (another evolutionary baseline).
- ADALEAD : ADALEAD is our recommended "benchmark" algorithm as it is robust to hyperparameters, and is relatively fast in execution. It also compares strongly to other state of the art algorithms.

```
@article{sinai2020adalead,
  title={AdaLead: A simple and robust adaptive greedy search algorithm for sequence␣
↪design},
  author={Sinai, Sam and Wang, Richard and Whatley, Alexander and Slocum, Stewart and␣
↪Locane, Elina and Kelsic, Eric},
  journal={arXiv preprint},
  year={2020}
}
```

## 4.11 DbAS and CbAS

- CbAS and DbAS

```
@article{brookes2019conditioning,
  title={Conditioning by adaptive sampling for robust design},
  author={Brookes, David H and Park, Hahnbeom and Listgarten, Jennifer},
  journal={arXiv preprint arXiv:1901.10060},
  year={2019}
}
@article{brookes2018design,
  title={Design by adaptive sampling},
  author={Brookes, David H and Listgarten, Jennifer},
  journal={arXiv preprint arXiv:1810.03714},
  year={2018}
}
```

## 4.12 Reinforcement Learning Algorithms

- DQN
- PPO
- DyNAPPO: See the following citation.

```
@inproceedings{angermueller2019model,
title={Model-based reinforcement learning for biological sequence design},
author={Angermueller, Christof and Dohan, David and Belanger, David and Deshpande,
↪ Ramya and Murphy, Kevin and Colwell, Lucy},
booktitle={International Conference on Learning Representations},
year={2019}
}
```

## 4.13 Bayesian Optimization

- Evolutionary/Enumerative BO: Bayesian optimization with sparse sampling of the mutation space. A fully enumerated (when possible) is also implemented mutation space.

CHAPTER

FIVE

# API DOCUMENTATION

## 5.1 flexs

The FLEXS (Fitness Landscape EXploration Sandbox) package.

### 5.1.1 flexs.ensemble

Defines the Ensemble class.

**class** flexs.ensemble.**Ensemble**(*models*, *combine_with=<function Ensemble.<lambda>>*)

Bases: *flexs.model.Model*

Class to ensemble models or landscapes together.

**models**

List of landscapes/models being ensembled.

**Type** List[flexs.Landscape]

**combine_with**

Function to combine ensemble predictions.

**Type** Callable[[np.ndarray], np.ndarray]

**train**(*sequences*, *labels*)

Train each model in *self.models*.

**Parameters**

- **sequences** (Union[List[str], ndarray]) – Training sequences
- **labels** (ndarray) – Training labels

### 5.1.2 flexs.evaluate

A small set of evaluation metrics to benchmark explorers.

flexs.evaluate.**adaptivity**(*landscape, make_explorer, num_rounds=[1, 10, 100], total_ground_truth_measurements=1000, total_model_queries=10000*)

For a fixed total budget of ground truth measurements and model queries, run with different numbers of rounds.

**Parameters**

- **landscape** (*Landscape*) – Ground truth fitness landscape.

- **make_explorer** (Callable[[int, int, int], *Explorer*]) – A function that takes in a number of rounds, a *sequences_batch_size* and a *model_queries_per_batch* and returns an explorer.
- **num_rounds** (List[int]) – A list of number of rounds to run the explorer with.
- **total_ground_truth_measurements** (int) – Total number of ground truth measurements across all rounds (*sequences_batch_size * rounds*).
- **total_model_queries** (int) – Total number of model queries across all rounds (*model_queries_per_round * rounds*).

flexs.evaluate.**efficiency**(*landscape, make_explorer, budgets=[(100, 500), (100, 5000), (1000, 5000), (1000, 10000)]*)

Evaluate explorer outputs as a function of the number of allowed ground truth measurements and model queries per round.

**Parameters**

- **landscape** (*Landscape*) – Ground truth fitness landscape.
- **make_explorer** (Callable[[int, int], *Explorer*]) – A function that takes in a *sequences_batch_size* and a *model_queries_per_batch* and returns an explorer.
- **budgets** (List[Tuple[int, int]]) – A list of tuples (*sequences_batch_size*, *model_queries_per_batch*).

flexs.evaluate.**robustness**(*landscape, make_explorer, signal_strengths=[0, 0.5, 0.75, 0.9, 1], verbose=True*)

Evaluate explorer outputs as a function of the noisyness of its model.

It runs the same explorer with *flexs.NoisyAbstractModel*'s of different signal strengths.

**Parameters**

- **landscape** (*Landscape*) – The landscape to run on.
- **make_explorer** (Callable[[*Model*, float], *Explorer*]) – A function that takes in a model and signal strength (for potential bookkeeping/logging purposes) and an explorer.
- **signal_strengths** (List[float]) – A list of signal strengths between 0 and 1.

### 5.1.3 flexs.explorer

Defines abstract base explorer class.

**class** flexs.explorer.**Explorer**(*model*, *name*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *log_file=None*)

Bases: abc.ABC

Abstract base explorer class.

Run explorer through the *run* method. Implement subclasses by overriding *propose_sequences* (do not override *run*).

**abstract propose_sequences**(*measured_sequences_data*)

Propose a list of sequences to be measured in the next round.

This method will be overriden to contain the explorer logic for each explorer.

**Parameters**

- **measured_sequences_data** (DataFrame) – A pandas dataframe of all sequences that have been

- **by the ground truth so far. Has columns "sequence",** (*measured*) –
- **"model_score", and "round".** (*"true_score",*) –

**Return type** Tuple[ndarray, ndarray]

**Returns**

**A tuple containing the proposed sequences and their scores** (according to the model).

**run**(*landscape*, *verbose=True*)
Run the exporer.

**Parameters**

- **landscape** (*Landscape*) – Ground truth fitness landscape.
- **verbose** (bool) – Whether to print output or not.

**Return type** Tuple[DataFrame, Dict]

### 5.1.4 flexs.landscape

Defines the Landscape class.

**class** flexs.landscape.**Landscape**(*name*)
Bases: abc.ABC

Base class for all landscapes and for *flexs.Model*.

**cost**
Number of sequences whose fitness has been evaluated.

**Type** int

**name**
A human-readable name for the landscape (often contains parameter values in the name) which is used when logging explorer runs.

**Type** str

**get_fitness**(*sequences*)
Score a list/numpy array of sequences.

This public method should not be overriden – new landscapes should override the private *_fitness_function* method instead. This method increments *self.cost* and then calls and returns *_fitness_function*.

**Parameters** **sequences** (Union[List[str], ndarray]) – A list/numpy array of sequence strings to be scored.

**Return type** ndarray

**Returns** Scores for each sequence.

### 5.1.5 flexs.model

Defines base Model class.

**class** flexs.model.**LandscapeAsModel**(*landscape*)
Bases: *flexs.model.Model*

This simple class wraps a *flexs.Landscape* in a *flexs.Model* to allow running experiments against a perfect model.

This class's *_fitness_function* simply calls the landscape's *_fitness_function*.

**train**(*sequences*, *labels*)
No-op.

**class** flexs.model.**Model**(*name*)
Bases: *flexs.landscape.Landscape*, abc.ABC

Base model class. Inherits from *flexs.Landscape* and adds an additional *train* method.

**abstract train**(*sequences*, *labels*)
Train model.

This function is called whenever you would want your model to update itself based on the set of sequences it has measurements for.

### 5.1.6 flexs.types

Types definitions for the flexs package.

### 5.1.7 flexs.baselines

Baselines module containing robust implementations of various models and explorers.

#### 5.1.7.1 flexs.baselines.explorers

FLEXS *explorers* module

#### flexs.baselines.explorers.adalead

Defines the Adalead explorer class.

**class** flexs.baselines.explorers.adalead.**Adalead**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *mu=1*, *recomb_rate=0*, *threshold=0.05*, *rho=0*, *eval_batch_size=20*, *log_file=None*)
Bases: *flexs.explorer.Explorer*

Adalead explorer.

**Algorithm works as follows:**

**Initialize set of top sequences whose fitnesses are at least** (1 - threshold) of the maximum fitness so far

**While we can still make model queries in this batch** Recombine top sequences and append to parents Rollout from parents and append to mutants

**propose_sequences**(*measured_sequences*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.bo

BO explorer.

**class** flexs.baselines.explorers.bo.**BO**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*, *method='EI'*, *recomb_rate=0*)
Bases: *flexs.explorer.Explorer*

Evolutionary Bayesian Optimization (Evo_BO) explorer.

**Algorithm works as follows:**

**for N experiment rounds**

**recombine samples from previous batch if it exists and measure them,** otherwise skip

Thompson sample starting sequence for new batch while less than B samples in batch

Generate *model_queries_per_batch/sequences_batch_size* samples If variance of ensemble models is above twice that of the starting

sequence

Thompson sample another starting sequence

**EI**(*vals*)
Compute expected improvement.

**static Thompson_sample**(*measured_batch*)
Pick a sequence via Thompson sampling.

**static UCB**(*vals*)
Upper confidence bound.

**initialize_data_structures**()
Initialize.

**pick_action**(*all_measured_seqs*)
Pick action.

**propose_sequences**(*measured_sequences*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

**sample_actions**()
Sample actions resulting in sequences to screen.

**train_models**()
Train the model.

**class** flexs.baselines.explorers.bo.**GPR_BO**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*, *seq_proposal_method='Thompson'*)
Bases: *flexs.explorer.Explorer*

Explorer using GP-based Bayesian Optimization.

Uses Gaussian process with RBF kernel on black box function. IMPORTANT: This explorer is not limited by any virtual screening restriction, and is used to find the unrestricted performance of Bayesian Optimization techniques in small landscapes.

Reference: http://krasserm.github.io/2018/03/21/bayesian-optimization/

**propose_sequences**(*measured_sequences*)
Propose *batch_size* samples.

**Return type** Tuple[ndarray, ndarray]

**propose_sequences_via_greedy**()
Propose a batch of new sequences.

Based on greedy in the expectation of the Gaussian posterior.

**propose_sequences_via_thompson**()
Propose a batch of new sequences. Based on Thompson sampling with a Gaussian posterior.

**propose_sequences_via_ucb**()
Propose a batch of new sequences. Based on upper confidence bound.

**reset**()
Reset.

### flexs.baselines.explorers.cbas_dbas

CbAS and DbAS explorers.

**class** flexs.baselines.explorers.cbas_dbas.**CbAS**(*model*, *generator*, *rounds*, *starting_sequence*, *sequences_batch_size*, *model_queries_per_batch*, *alphabet*, *algo='cbas'*, *Q=0.7*, *cycle_batch_size=100*, *mutation_rate=0.2*, *log_file=None*)

Bases: *flexs.explorer.Explorer*

CbAS and DbAS explorers.

**propose_sequences**(*measured_sequences_data*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.cmaes

CMAES explorer.

**class** flexs.baselines.explorers.cmaes.**CMAES**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *population_size=15*, *max_iter=400*, *initial_variance=0.2*, *log_file=None*)

Bases: *flexs.explorer.Explorer*

An explorer which implements the covariance matrix adaptation evolution strategy (CMAES).

Optimizes a continuous relaxation of the one-hot sequence that we use to construct a normal distribution around, sample from, and then argmax to get sequences for the objective function.

http://blog.otoro.net/2017/10/29/visual-evolution-strategies/ is a helpful guide.

**propose_sequences**(*measured_sequences*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.dqn

DQN explorer.

**class** flexs.baselines.explorers.dqn.**DQN**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*, *memory_size=100000*, *train_epochs=20*, *gamma=0.9*, *device='cpu'*)
Bases: *flexs.explorer.Explorer*

DQN explorer class.

DQN Explorer implementation, based off https://colab.research.google.com/drive/1NsbSPn6jOcaJB_mp9TmkgQX7UrRIrTi0.

Algorithm works as follows: for N experiment rounds

collect samples with policy policy updates using Q network:

Q(s, a) <- Q(s, a) + alpha * (R(s, a) + gamma * max Q(s, a) - Q(s, a))

**calculate_next_q_values**(*state_v*)
Calculate the next Q values.

**get_action_and_mutant**(*epsilon*)
Return an action and the resulting mutant.

**initialize_data_structures**()
Initialize internal data structures.

**pick_action**(*all_measured_seqs*)
Pick an action.

Generates a new string representing the state, along with its associated reward.

**propose_sequences**(*measured_sequences_data*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

**q_network_loss**(*batch*)
Calculate MSE.

Computes between actual state action values, and expected state action values from DQN.

**sample**()
Sample a random *batch_size* subset of the memory.

**train_actor**(*train_epochs*)
Train the Q Network.

**class** flexs.baselines.explorers.dqn.**Q_Network**(*sequence_len*, *alphabet_len*)
Bases: torch.nn.modules.module.Module

Q Network implementation, used in DQN Explorer.

**forward**(*x*)
Take a forward step.

**training = None**

flexs.baselines.explorers.dqn.**build_q_network**(*sequence_len*, *alphabet_len*, *device*)
Build the Q Network.

### flexs.baselines.explorers.dyna_ppo

DyNA-PPO explorer.

**class** flexs.baselines.explorers.dyna_ppo.**DynaPPO**(*landscape*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*, *model=None*, *num_experiment_rounds=10*, *num_model_rounds=1*, *env_batch_size=4*)

Bases: *flexs.explorer.Explorer*

Explorer which implements DynaPPO.

**This RL-based sequence design algorithm works as follows:**

**for r in rounds:** train_policy(experimental_data_rewards[r]) for m in model_based_rounds:

train_policy(model_fitness_rewards[m])

An episode for the agent begins with an empty sequence, and at each timestep, one new residue is generated and added to the sequence until the desired length of the sequence is reached. The reward is zero at all timesteps until the last one, when the reward is *reward = lambda * sequence_density + sequence_fitness* where sequence density is the density of nearby sequences already proposed.

As described above, this explorer generates sequences *constructively*.

Paper: https://openreview.net/pdf?id=HklxbgBKvr

**add_last_seq_in_trajectory**(*experience*, *new_seqs*)
Add the last sequence in an episode's trajectory.

Given a trajectory object, checks if the object is the last in the trajectory. Since the environment ends the episode when the score is non-increasing, it adds the associated maximum-valued sequence to the batch.

If the episode is ending, it changes the "current sequence" of the environment to the next one in *last_batch*, so that when the environment resets, mutants are generated from that new sequence.

**propose_sequences**(*measured_sequences_data*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

**class** flexs.baselines.explorers.dyna_ppo.**DynaPPOEnsemble**(*seq_len*, *alphabet*, *r_squared_threshold=0.5*, *models=None*)

Bases: *flexs.model.Model*

Ensemble from DyNAPPO paper.

Ensembles many models together but only uses those with an $r^2$ above a certain threshold (on validation data) at test-time.

**train**(*sequences*, *labels*)
Train the ensemble, calculating $r^2$ values on a holdout set.

**class** flexs.baselines.explorers.dyna_ppo.**DynaPPOMutative**(*landscape*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*, *model=None*, *num_experiment_rounds=10*, *num_model_rounds=1*)

Bases: *flexs.explorer.Explorer*

Explorer which implements DynaPPO.

Note that unlike the other DynaPPO explorer, this one is mutative rather than constructive. Specifically, instead of starting from an empty sequence and generating residues one-by-one, this explorer starts from a complete sequence (fitness thresholds to start with good sequences) and mutates it until the mutant's fitness has started to decrease. Then it ends the episode.

This has proven to be a stronger algorithm than the original DyNAPPO.

Paper: https://openreview.net/pdf?id=HklxbgBKvr

**add_last_seq_in_trajectory**(*experience*, *new_seqs*)
Add the last sequence in an episode's trajectory.

Given a trajectory object, checks if the object is the last in the trajectory. Since the environment ends the episode when the score is non-increasing, it adds the associated maximum-valued sequence to the batch.

If the episode is ending, it changes the "current sequence" of the environment to the next one in *last_batch*, so that when the environment resets, mutants are generated from that new sequence.

**propose_sequences**(*measured_sequences_data*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.genetic_algorithm

Define a baseline genetic algorithm implementation.

**class** flexs.baselines.explorers.genetic_algorithm.**GeneticAlgorithm**(*model*, *rounds*, *starting_sequence*, *sequences_batch_size*, *model_queries_per_batch*, *alphabet*, *population_size*, *parent_selection_strategy*, *children_proportion*, *log_file=None*, *parent_selection_proportion=None*, *beta=None*, *seed=None*)

Bases: *flexs.explorer.Explorer*

A genetic algorithm explorer with single point mutations and recombination.

Based on the *parent_selection_strategy*, this class implements one of three genetic algorithms:

1. If *parent_selection_strategy == 'top-k'*, we have a traditional genetic algorithm where the top-k scoring sequences in the population become parents.
2. If *parent_selection_strategy == 'wright-fisher'*, we have a genetic algorithm based off of the Wright-Fisher model of evolution, where members of the population become parents with a probability exponential to their fitness (softmax the scores then sample).

**propose_sequences**(*measured_sequences*)

Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.ppo

PPO explorer.

**class** flexs.baselines.explorers.ppo.**PPO**(*model*, *rounds*, *sequences_batch_size*, *model_queries_per_batch*, *starting_sequence*, *alphabet*, *log_file=None*)

Bases: *flexs.explorer.Explorer*

Explorer which uses PPO.

**The algorithm is:**

**for N experiment rounds** collect samples with policy train policy on samples

A simpler baseline than DyNAPPOMutative with similar performance.

**add_last_seq_in_trajectory**(*experience*, *new_seqs*)

Add the last sequence in an episode's trajectory.

Given a trajectory object, checks if the object is the last in the trajectory. Since the environment ends the episode when the score is non-increasing, it adds the associated maximum-valued sequence to the batch.

If the episode is ending, it changes the "current sequence" of the environment to the next one in *last_batch*, so that when the environment resets, mutants are generated from that new sequence.

**propose_sequences**(*measured_sequences_data*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.random

Defines the Random explorer class.

**class** flexs.baselines.explorers.random.**Random**(*model*, *rounds*, *starting_sequence*, *sequences_batch_size*, *model_queries_per_batch*, *alphabet*, *mu=1*, *elitist=False*, *seed=None*, *log_file=None*)

Bases: *flexs.explorer.Explorer*

A simple random explorer.

Chooses a random previously measured sequence and mutates it.

A good baseline to compare other search strategies against.

Since random search is not data-driven, the model is only used to score sequences, but not to guide the search strategy.

**propose_sequences**(*measured_sequences*)
Propose top *sequences_batch_size* sequences for evaluation.

**Return type** Tuple[ndarray, ndarray]

### flexs.baselines.explorers.environments

Reinforcement learning environments for DynaPPO and PPO explorers.

### flexs.baselines.explorers.environments.dyna_ppo

DyNA-PPO environment module.

**class** flexs.baselines.explorers.environments.dyna_ppo.**DynaPPOEnvironment**(*alphabet*, *seq_length*, *model*, *landscape*, *batch_size*)

Bases: tf_agents.environments.py_environment.PyEnvironment

DyNA-PPO environment based on TF-Agents.

**action_spec**()
Define agent actions.

**property batch_size**
Tf-agents property that return env batch size.

**batched**()
Tf-agents function that says that this env returns batches of timesteps.

**get_cached_fitness**(*seq*)
Get cached sequence fitness computed in previous episodes.

**observation_spec**()
Define environment observations.

**sequence_density**(*seq*)
Get average distance to *seq* out of all observed sequences.

**set_fitness_model_to_gt**(*fitness_model_is_gt*)
Set the fitness model to the ground truth landscape or to the model.

Call with *True* when doing an experiment-based training round and call with *False* when doing a model-based training round.

**time_step_spec**()
Define time steps.

**class** flexs.baselines.explorers.environments.dyna_ppo.**DynaPPOEnvironmentMutative**(*alphabet*, *starting_seq*, *model*, *landscape*, *max_num_ste*

Bases: tf_agents.environments.py_environment.PyEnvironment

DyNA-PPO environment based on TF-Agents.

Note that unlike the other DynaPPO environment, this one is mutative rather than constructive.

**action_spec**()
Define agent actions.

**get_state_string**()
Get sequence representing current state.

**observation_spec**()
Define environment observations.

**sequence_density**(*seq*)
Get average distance to *seq* out of all observed sequences.

**set_fitness_model_to_gt**(*fitness_model_is_gt*)
Set the fitness model to the ground truth landscape or to the model.

Call with *True* when doing an experiment-based training round and call with *False* when doing a model-based training round.

#### flexs.baselines.explorers.environments.ppo

PPO environment module.

**class** flexs.baselines.explorers.environments.ppo.**PPOEnvironment**(*alphabet*, *starting_seq*, *model*, *max_num_steps*)

Bases: tf_agents.environments.py_environment.PyEnvironment

PPO environment based on TF-Agents.

**action_spec**()
Define agent actions.

**get_state_string**()
Get sequence representing current state.

**observation_spec**()
Define environment observations.

### 5.1.7.2 flexs.baselines.models

*baselines.models* module.

#### flexs.baselines.models.adaptive_ensemble

Defines the AdaptiveEnsemble model.

**class** flexs.baselines.models.adaptive_ensemble.**AdaptiveEnsemble**(*models*, *combine_with='sum'*, *adapt_weights_with='r2_weights'*, *adaptive_val_size=0.2*)

Bases: *flexs.model.Model*

Ensemble class that weights individual model predictions adaptively, according to some reweighting function.

**train**(*sequences*, *labels*)
Train each model in the ensemble and then adaptively reweight them according to *adapt_weights_with*.

**Parameters**

- **sequences** (Union[List[str], ndarray]) – Training sequences.
- **lables** – Training sequence labels.

flexs.baselines.models.adaptive_ensemble.**r2_weights**(*model_preds*, *labels*)

**Parameters**

- **model_preds** (ndarray) – A numpy array of shape (num_models, num_samples) containing model predictions.
- **labels** (ndarray) – A numpy array of true labels.

**Return type** ndarray

**Returns** A numpy array of shape (num_models,) containing $r^2$ scores for models.

### flexs.baselines.models.cnn

Define a baseline CNN Model.

**class** flexs.baselines.models.cnn.**CNN**(*seq_len*, *num_filters*, *hidden_size*, *alphabet*, *loss='MSE'*, *name=None*, *batch_size=256*, *epochs=20*)

Bases: *flexs.baselines.models.keras_model.KerasModel*

A baseline CNN model with 3 conv layers and 2 dense layers.

### flexs.baselines.models.global_epistasis_model

Define a global epistasis model.

**class** flexs.baselines.models.global_epistasis_model.**GlobalEpistasisModel**(*seq_len*, *hidden_size*, *alphabet*, *loss='MSE'*, *name=None*, *batch_size=256*, *epochs=20*)

Bases: *flexs.baselines.models.keras_model.KerasModel*

Global epistasis model.

Weighted sum of input features follow by several dense layers. A simple, but relatively uneffective nonlinear model.

### flexs.baselines.models.keras_model

Define the base KerasModel class.

**class** flexs.baselines.models.keras_model.**KerasModel**(*model*, *alphabet*, *name*, *batch_size=256*, *epochs=20*, *custom_train_function=None*, *custom_predict_function=None*)

Bases: *flexs.model.Model*

A wrapper around tensorflow/keras models.

**train**(*sequences*, *labels*, *verbose=False*)

Train keras model.

### flexs.baselines.models.mlp

Define a baseline multilayer perceptron model.

**class** flexs.baselines.models.mlp.**MLP**(*seq_len*, *hidden_size*, *alphabet*, *loss='MSE'*, *name=None*, *batch_size=256*, *epochs=20*)

Bases: *flexs.baselines.models.keras_model.KerasModel*

A baseline MLP with three dense layers and relu activations.

### flexs.baselines.models.noisy_abstract_model

Define the noisy abstract model class.

**class** flexs.baselines.models.noisy_abstract_model.**NoisyAbstractModel**(*landscape*, *signal_strength=0.9*)

Bases: *flexs.model.Model*

Behaves like a ground truth model.

It corrupts a ground truth model with noise, which is modulated by distance to already measured sequences.

Specifically, $hat{f}(x) = alpha^d f(x) + (1 - alpha^d) epsilon$ where $epsilon$ is drawn from an exponential distribution with mean $f(x)$ $d$ is the edit distance to the closest measured neighbor, and $alpha$ is the signal strength.

**train**(*sequences*, *labels*)

Training step simply stores sequences and labels in a dictionary for future lookup.

### flexs.baselines.models.sklearn_models

Define scikit-learn model wrappers as well a few convenient pre-wrapped models.

**class** flexs.baselines.models.sklearn_models.**LinearRegression**(*alphabet*, ***kwargs*)

Bases: *flexs.baselines.models.sklearn_models.SklearnRegressor*

Sklearn linear regression.

**class** flexs.baselines.models.sklearn_models.**LogisticRegression**(*alphabet*, ***kwargs*)

Bases: *flexs.baselines.models.sklearn_models.SklearnRegressor*

Sklearn logistic regression.

**class** flexs.baselines.models.sklearn_models.**RandomForest**(*alphabet*, ***kwargs*)

Bases: *flexs.baselines.models.sklearn_models.SklearnRegressor*

Sklearn random forest regressor.

**class** flexs.baselines.models.sklearn_models.**SklearnClassifier**(*model*, *alphabet*, *name*)

Bases: *flexs.baselines.models.sklearn_models.SklearnModel*, abc.ABC

Class for sklearn classifiers (uses *model.predict_proba(. . . )[:, 1]*).

**class** flexs.baselines.models.sklearn_models.**SklearnModel**(*model*, *alphabet*, *name*)

Bases: *flexs.model.Model*, abc.ABC

Base sklearn model wrapper.

**train**(*sequences*, *labels*)
Flatten one-hot sequences and train model using *model.fit*.

**class** flexs.baselines.models.sklearn_models.**SklearnRegressor**(*model*, *alphabet*, *name*)
Bases: *flexs.baselines.models.sklearn_models.SklearnModel*, abc.ABC

Class for sklearn regressors (uses *model.predict*).

### 5.1.8 flexs.landscapes

FLEXS landscapes module.

#### 5.1.8.1 flexs.landscapes.additive_aav_packaging

Defines the AdditiveAAVPackaging landscape and problem registry.

**class** flexs.landscapes.additive_aav_packaging.**AdditiveAAVPackaging**(*phenotype='heart'*, *minimum_fitness_multiplier=1*, *start=0*, *end=735*, *noise=0*)
Bases: *flexs.landscape.Landscape*

An Additive landscape based on data from AAV2 packaging fitness measurements.

By additive landscape, we mean that each residue at each position is given a fitness and the fitness of the sequence is the sum of these individual fitnesses. This means that the fitness contribution per residue is independent of the identities of the other residues. This makes for a very simple landscape.

**wild_type**
AAV2 wild_type substring between positions *start* and *end*.

**Type** str

**compute_max_possible**()
Compute max possible fitness of any sequence (used for normalization).

flexs.landscapes.additive_aav_packaging.**registry**()
Return a dictionary of problems of the form: ```{

**“problem name”: {** “params”: . . .

}```

where *flexs.landscapes.AdditiveAAVPackaging(**problem[“params”])* instantiates the additive AAV packaging landscape for the given set of parameters.

**Returns** Problems in the registry.

**Return type** dict

#### 5.1.8.2 flexs.landscapes.bert_gfp

Defines the BertGFPBrightness landscape.

**class** flexs.landscapes.bert_gfp.**BertGFPBrightness**
Bases: *flexs.landscape.Landscape*

Green fluorescent protein (GFP) brightness landscape.

The oracle used in this lanscape is the transformer model from TAPE (https://github.com/songlab-cal/tape).

To create the transformer model used here, run the command:

```tape-train transformer fluorescence –from_pretrained bert-base –batch_size 128 –gradient_accumulation_steps 10 –data_dir .```

Note that the output of this landscape is not normalized to be between 0 and 1.

**gfp_wt_sequence**
Wild-type sequence for jellyfish green fluorescence protein.

**Type** str

**gfp_wt_sequence = 'MSKGEELFTGVVPILVELDGDVNGHKFSVSGEGEGDATYGKLTLKFICTTGKLPVPWPTLVTTLSYG**

#### 5.1.8.3 flexs.landscapes.rna

Defines RNA binding landscape and problem registry.

**class** flexs.landscapes.rna.**RNABinding**(*targets*, *seq_length*, *conserved_region=None*)
Bases: *flexs.landscape.Landscape*

RNA binding landscape using ViennaRNA *duplexfold*.

**compute_min_binding_energies**()
Compute the lowest possible binding energy for each target.

**class** flexs.landscapes.rna.**RNAFolding**(*norm_value=1*)
Bases: *flexs.landscape.Landscape*

RNA folding landscape using ViennaRNA *fold*.

flexs.landscapes.rna.**registry**()
Return a dictionary of problems of the form: `{

**“problem name”: {** “params”: . . . , “starts”: . . .

}`

where *flexs.landscapes.RNABinding(**problem[“params”])* instantiates the RNA binding landscape for the given set of parameters.

**Returns** Problems in the registry.

**Return type** dict
```

#### 5.1.8.4 flexs.landscapes.rosetta

Defines the RosettaFolding landscape and problem registry.

**class** flexs.landscapes.rosetta.**RosettaFolding**(*pdb_file*, *sigmoid_center*, *sigmoid_norm_value*)

Bases: *flexs.landscape.Landscape*

This oracle scores sequences using a fixed conformation design energy. In this case, both backbone and side chain conformations are fixed (no repacking).

In this setting, we have a 3-D structure that we'd like to design for (given by the PDB file), so we look for sequences that might stably fold to the given conformation.

The best way to query how well a sequence might fold to a given conformation is to run a folding simulation, but since that is so computationally intense, it is more common to simply calculate the energy of the sequence if it was forced to fold into the target 3-D structure.

This is just a proxy for folding stability, but it is often an effective one and is the approach used by RosettaDesign.

We use Rosetta's centroid energy function instead of the full-atom one since it is less sensitive to switching out residues without repacking side-chain conformations.

We convert these energies to a maximization objective in the 0-1 scale by fitness = (-energy - *sigmoid_center*) / *sigmoid_norm_value*.

**wt_pose**
The original PyRosetta pose object from the .pdb file. Call *wt_pose.sequence()* to get the wild type sequence.

**get_folding_energy**(*sequence*)
Return rosetta folding energy of the given sequence in *self.pose*'s conformation.

flexs.landscapes.rosetta.**registry**()
Return a dictionary of problems of the form: `{

**"problem name": {** "params": . . . ,

}`

where *flexs.landscapes.RosettaFolding(**problem["params"])* instantiates the rosetta folding landscape for the given set of parameters.

**Returns** Problems in the registry.

**Return type** dict

#### 5.1.8.5 flexs.landscapes.tf_binding

Define TFBinding landscape and problem registry.

**class** flexs.landscapes.tf_binding.**TFBinding**(*landscape_file*)
Bases: *flexs.landscape.Landscape*

A landscape of binding affinity of proposed 8-mer DNA sequences to a particular transcription factor.

We use experimental data from Barrera et al. (2016), a survey of the binding affinity of more than one hundred and fifty transcription factors (TF) to all possible DNA sequences of length 8.

flexs.landscapes.tf_binding.**registry**()
Return a dictionary of problems of the form:

```python {
```

**"problem name": {** "params": . . . ,

where *flexs.landscapes.TFBinding(**problem["params"])* instantiates the transcription factor binding landscape for the given set of parameters.

**Return type** `Dict[str, Dict]`

**Returns** Problems in the registry.

### 5.1.9 flexs.utils

Utility modules.

*utils.sequence_utils* is the most important and useful.

#### 5.1.9.1 flexs.utils.VAE_utils

Utility functions for A VAE generative model.

**class** flexs.utils.VAE_utils.**Sampling**(*trainable=True*, *name=None*, *dtype=None*, *dynamic=False*, ***kwargs*)

Bases: `tensorflow.python.keras.engine.base_layer.Layer`

Uses (z_mean, z_log_var) to sample z, the vector encoding a sequence.

**call**(*inputs*)

Sample from multivariate guassian defined by *inputs = (z_mean, z_log_var)*.

**class** flexs.utils.VAE_utils.**VAE**(*seq_length*, *alphabet*, *batch_size=10*, *latent_dim=2*, *intermediate_dim=250*, *epochs=10*, *epsilon_std=1.0*, *beta=1*, *validation_split=0.2*, *verbose=True*)

Bases: `object`

VAE class wrapping *VAEModel*, exposing an interface friendly to CbAS/DbAS.

**calculate_log_probability**(*sequences*, *vae=None*)

Calculate log probability of reconstructing a sequence.

**generate**(*n_samples*, *existing_samples*, *existing_weights*)

Generate *n_samples* new samples such that none of them are in *existing_samples*.

**train_model**(*samples*, *weights*)

Train VAE on *samples* according to their *weights*.

**class** flexs.utils.VAE_utils.**VAEModel**(*original_dim*, *intermediate_dim*, *latent_dim*, ***kwargs*)

Bases: `tensorflow.python.keras.engine.training.Model`

Keras implementation of VAE for CbAS/DbAS.

**call**(*data*)

Return the VAE's reconstruction of *data*.

**generate**()

Generate a new sequence by sampling the latent space and then decoding.

**train_step**(*data*)

Define a custom train step taking in *data* and returning the loss.

flexs.utils.VAE_utils.**pwm_to_boltzmann_weights**(*prob_weight_matrix*, *temp*)

Convert pwm to boltzmann weights for categorical distribution sampling.

#### 5.1.9.2 flexs.utils.replay_buffers

Defines replay buffers used by some explorers.

**class** flexs.utils.replay_buffers.**MinSegmentTree**(*capacity*)
Bases: *flexs.utils.replay_buffers.SegmentTree*

Create SegmentTree. Taken from OpenAI baselines Github repository: https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py

**min**(*start=0*, *end=0*)
Return min(arr[start], . . . , arr[end]).

**Return type** float

**class** flexs.utils.replay_buffers.**PrioritizedReplayBuffer**(*obs_dim*, *size*, *batch_size=32*, *alpha=0.6*)
Bases: *flexs.utils.replay_buffers.ReplayBuffer*

Prioritized Replay buffer.

**max_priority**
max priority

**Type** float

**tree_ptr**
next index of tree

**Type** int

**alpha**
alpha parameter for prioritized replay buffer

**Type** float

**sum_tree**
sum tree for prior

**Type** *SumSegmentTree*

**min_tree**
min tree for min prior to get max weight

**Type** *MinSegmentTree*

**sample_batch**(*beta=0.4*)
Sample a batch of experiences.

**Return type** Dict[str, ndarray]

**store**(*obs*, *act*, *rew*, *next_obs*)
Store experience and priority.

**update_priorities**(*indices*, *priorities*)
Update priorities of sampled transitions.

**class** flexs.utils.replay_buffers.**ReplayBuffer**(*obs_dim*, *size*, *batch_size=128*)
Bases: object

A simple numpy replay buffer.

**sample_batch**()
Sample batch of timesteps from replay buffer.

**Return type** Dict[str, ndarray]

**store**(*obs*, *act*, *rew*, *next_obs*)
Store timestep in replay buffer.

**class** flexs.utils.replay_buffers.**SegmentTree**(*capacity*, *operation*, *init_value*)
Bases: object

Create SegmentTree. Taken from OpenAI baselines Github repository: https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py

**capacity**

**Type** int

**tree**

**Type** list

**operation**

**Type** function

**operate**(*start=0*, *end=0*)
Return result of applying *self.operation*.

**Return type** float

**class** flexs.utils.replay_buffers.**SumSegmentTree**(*capacity*)
Bases: *flexs.utils.replay_buffers.SegmentTree*

Create SumSegmentTree. Taken from OpenAI baselines github repository: https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py

**retrieve**(*upperbound*)
Find the highest index *i* about upper bound in the tree

**Return type** int

**sum**(*start=0*, *end=0*)
Return arr[start] + ... + arr[end].

**Return type** float

#### 5.1.9.3 flexs.utils.sequence_utils

Utility functions for manipulating sequences.

flexs.utils.sequence_utils.**AAS = 'ILVAGMFYWEDQNHCRKSTP'**
Amino acid alphabet for proteins (length 20 - no stop codon).

**Type** str

flexs.utils.sequence_utils.**BA = '01'**
Binary alphabet '01'.

**Type** str

flexs.utils.sequence_utils.**DNAA = 'TGCA'**
DNA alphabet (4 base pairs).

**Type** str

flexs.utils.sequence_utils.**RNAA = 'UGCA'**
RNA alphabet (4 base pairs).

**Type** str

flexs.utils.sequence_utils.**construct_mutant_from_sample**(*pwm_sample*, *one_hot_base*)

Return one hot mutant, a utility function for some explorers.

**Return type** ndarray

flexs.utils.sequence_utils.**generate_random_mutant**(*sequence*, *mu*, *alphabet*)

Generate a mutant of *sequence* where each residue mutates with probability *mu*.

So the expected value of the total number of mutations is *len(sequence) * mu*.

**Parameters**

- **sequence** (str) – Sequence that will be mutated from.
- **mu** (float) – Probability of mutation per residue.
- **alphabet** (str) – Alphabet string.

**Return type** str

**Returns** Mutant sequence string.

flexs.utils.sequence_utils.**generate_random_sequences**(*length*, *number*, *alphabet*)

Generate random sequences of particular length.

**Return type** List[str]

flexs.utils.sequence_utils.**generate_single_mutants**(*wt*, *alphabet*)

Generate all single mutants of *wt*.

**Return type** List[str]

flexs.utils.sequence_utils.**one_hot_to_string**(*one_hot*, *alphabet*)

Return the sequence string representing a one-hot vector according to an alphabet.

**Parameters**

- **one_hot** (Union[List[List[int]], ndarray]) – One-hot of shape *(len(sequence), len(alphabet)* representing a sequence.
- **alphabet** (str) – Alphabet string (assigns each character an index).

**Return type** str

**Returns** Sequence string representation of *one_hot*.

flexs.utils.sequence_utils.**string_to_one_hot**(*sequence*, *alphabet*)

Return the one-hot representation of a sequence string according to an alphabet.

**Parameters**

- **sequence** (str) – Sequence string to convert to one_hot representation.
- **alphabet** (str) – Alphabet string (assigns each character an index).

**Return type** ndarray

**Returns** One-hot numpy array of shape *(len(sequence), len(alphabet))*.

CHAPTER

SIX

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## f

# INDEX

## F



## G



## I



## K

## L



## M



## N



## O



## P



## Q



## R

## S



## T



## U

## V



## W