lagom Documentation

Release 0.0.3

Xingdong Zuo

Nov 24, 2019

Installation

inte för mycket och inte för lite, enkelhet är bäst
not too much and not too little, simplicity is often the best

lagom is a light PyTorch infrastructure to quickly prototype reinforcement learning algorithms.

lagom balances between the flexibility and the userability when developing reinforcement learning (RL) algorithms. The library is built on top of PyTorch and provides modular tools to quickly prototype RL algorithms. However, we do not go overboard, because going too low level is rather time consuming and prone to potential bugs, while going too high level degrades the flexibility which makes it difficult to try out some crazy ideas.

We are continuously making lagom more 'self-contained' to run experiments quickly. Now, it internally supports base classes for multiprocessing (master-worker framework) to parallelize (e.g. experiments and evolution strategies). It also supports hyperparameter search by defining configurations either as grid search or random search.

One of the main pipelines to use lagom can be done as following:

1. Define environment and RL agent
2. User runner to collect data for agent
3. Define algorithm to train agent
4. Define experiment and configurations.

A graphical illustration is coming soon.

CHAPTER 1

Setup environment

CHAPTER 2

Installing lagom

CHAPTER 3

lagom

3.1 Agent

class lagom.**BaseAgent**(*config*, *env*, *device*, ***kwargs*)

Base class for all agents.

The agent could select an action from some data (e.g. observation) and update itself by defining a certain learning mechanism.

Any agent should subclass this class, e.g. policy-based or value-based.

Parameters

- **config** (*dict*) – a dictionary of configurations
- **env** (*Env*) – environment object.
- **device** (*Device*) – a PyTorch device
- ****kwargs** – keyword aguments used to specify the agent

choose_action(*x*, ***kwargs*)

Returns the selected action given the data.

Note: It's recommended to handle all dtype/device conversions between CPU/GPU or Tensor/Numpy here.

The output is a dictionary containing useful items,

Parameters

- **obs** (*object*) – batched observation returned from the environment. First dimension is treated as batch dimension.
- ****kwargs** – keyword arguments to specify action selection.

Returns

a dictionary of action selection output. It contains all useful information (e.g. action, action_logprob, state_value). This allows the API to be generic and compatible with different kinds of runner and agents.

Return type dict

learn(*D*, ***kwargs*)

Defines learning mechanism to update the agent from a batched data.

Parameters

- **D** (*list*) – a list of batched data to train the agent e.g. in policy gradient, this can be a list of *Trajectory*.
- ****kwargs** – keyword arguments to specify learning mechanism

Returns a dictionary of learning output. This could contain the loss and other useful metrics.

Return type dict

class lagom.**RandomAgent**(*config*, *env*, *device*, ***kwargs*)

A random agent samples action uniformly from action space.

choose_action(*x*, ***kwargs*)

Returns the selected action given the data.

Note: It's recommended to handle all dtype/device conversions between CPU/GPU or Tensor/Numpy here.

The output is a dictionary containing useful items,

Parameters

- **obs** (*object*) – batched observation returned from the environment. First dimension is treated as batch dimension.
- ****kwargs** – keyword arguments to specify action selection.

Returns

a dictionary of action selection output. It contains all useful information (e.g. action, action_logprob, state_value). This allows the API to be generic and compatible with different kinds of runner and agents.

Return type dict

learn(*D*, ***kwargs*)

Defines learning mechanism to update the agent from a batched data.

Parameters

- **D** (*list*) – a list of batched data to train the agent e.g. in policy gradient, this can be a list of *Trajectory*.
- ****kwargs** – keyword arguments to specify learning mechanism

Returns a dictionary of learning output. This could contain the loss and other useful metrics.

Return type dict

3.2 Data

class lagom.**StepType**
An enumeration.

class lagom.**TimeStep**(*step_type: lagom.data.StepType*, *observation: object*, *reward: float*, *done: bool*, *info: dict*)

class lagom.**Trajectory**

3.3 Logger

class lagom.**Logger**
Log the information in a dictionary.

If a key is logged more than once, then the new value will be appended to a list.

Note: It uses pickle to serialize the data. Empirically, `pickle` is 2x faster than `numpy.save` and other alternatives like `yaml` is too slow and `JSON` does not support numpy array.

Warning: It is discouraged to store hierarchical structure, e.g. list of dict of list of ndarray. Because pickling such complex and large data structure is extremely slow. Put dictionary only at the topmost level. Large numpy array should be saved separately.

Example:

- Default:

```
>>> logger = Logger()
>>> logger('iteration', 1)
>>> logger('train_loss', 0.12)
>>> logger('iteration', 2)
>>> logger('train_loss', 0.11)
>>> logger('iteration', 3)
>>> logger('train_loss', 0.09)

>>> logger
OrderedDict([('iteration', [1, 2, 3]), ('train_loss', [0.12, 0.11, 0.09])])

>>> logger.dump()
Iteration: [1, 2, 3]
Train Loss: [0.12, 0.11, 0.09]
```

- With indentation:

```
>>> logger.dump(indent=1)
    Iteration: [1, 2, 3]
    Train Loss: [0.12, 0.11, 0.09]
```

- With specific keys:

```
>>> logger.dump(keys=['iteration'])
Iteration: [1, 2, 3]
```

- With specific index:

```
>>> logger.dump(index=0)
Iteration: 1
Train Loss: 0.12
```

- With specific list of indices:

```
>>> logger.dump(index=[0, 2])
Iteration: [1, 3]
Train Loss: [0.12, 0.09]
```

__call__(*key*, *value*)
Log the information with given key and value.

Note: The key should be semantic and each word is separated by _.

Parameters

- **key** (*str*) – key of the information
- **value** (*object*) – value to be logged

clear()
Remove all loggings in the dictionary.

dump(*keys=None*, *index=None*, *indent=0*, *border=''*)
Dump the loggings to the screen.

Parameters

- **keys** (*list, optional*) – a list of selected keys. If `None`, then use all keys. Default: `None`
- **index** (*int/list, optional*) – the index of logged values. It has following use cases:
  - `scalar`: a specific index. If `-1`, then use last element.
  - `list`: a list of indicies.
  - `None`: all indicies.
- **indent** (*int, optional*) – the number of tab indentation. Default: `0`
- **border** (*str, optional*) – the string to print as header and footer

save(*f*)
Save loggings to a file.

Parameters **f** (*str*) – file path

3.4 Engine

class lagom.**BaseEngine**(*config*, ***kwargs*)
Base class for all engines.

It defines the training and evaluation process.

eval(*n=None*, ***kwargs*)
Evaluation process for one iteration.

Note: It is recommended to use *Logger* to store loggings.

Note: All parameterized modules should be called *.eval()* to specify evaluation mode.

Parameters

- **n** (*int, optional*) – n-th iteration for evaluation.
- ****kwargs** – keyword aguments used for logging.

Returns a dictionary of evluation output

Return type dict

train(*n=None*, ***kwargs*)
Training process for one iteration.

Note: It is recommended to use *Logger* to store loggings.

Note: All parameterized modules should be called *.train()* to specify training mode.

Parameters

- **n** (*int, optional*) – n-th iteration for training.
- ****kwargs** – keyword aguments used for logging.

Returns a dictionary of training output

Return type dict

3.5 Runner

class lagom.**BaseRunner**
Base class for all runners.

A runner is a data collection interface between the agent and the environment.

__call__(*agent*, *env*, ***kwargs*)
Defines data collection via interactions between the agent and the environment.

Parameters

- **agent** (*BaseAgent*) – agent
- **env** (*Env*) – environment
- ****kwargs** – keyword arguments for more specifications.

class lagom.**EpisodeRunner**

class lagom.**StepRunner**(*reset_on_call=True*)

3.6 Evolution Strategies

class lagom.**BaseES**
Base class for all evolution strategies.

Note: The optimization is treated as minimization. e.g. maximize rewards is equivalent to minimize negative rewards.

Note: For painless parallelization, we highly recommend to use *concurrent.futures.ProcessPoolExecutor* with a few practical tips.

- Set *max_workers* argument to control the max parallelization capacity.
- When execution get stuck, try to use `CloudpickleWrapper` to wrap the objective function e.g. particularly for lambda, class methods
- Use *with ProcessPoolExecutor* once to wrap entire iterative ES generations. Because using this internally for each generation, it can slow down the parallelization dramatically due to overheads.
- **To reduce overheads further (e.g. PyTorch models, gym environments)**
  - Recreate such models for each generation will be very expensive.
  - Use initializer function for ProcessPoolExecutor
  - Within initializer function, define PyTorch models and gym environments as global variables Note that the global variables are defined to each worker independently
  - Don't forget to use *with torch.no_grad* to increase forward pass speed.

---

ask()
Sample a set of new candidate solutions.

Returns a list of sampled candidate solutions

Return type list

result
Return a namedtuple of all results for the optimization.

It contains: * xbest: best solution evaluated * fbest: objective function value of the best solution * evals_best: evaluation count when xbest was evaluated * evaluations: evaluations overall done * iterations: number of iterations * xfavorite: distribution mean in "phenotype" space, to be considered as current best estimate of the optimum * stds: effective standard deviations

tell(*solutions*, *function_values*)
Update the parameters of the population for a new generation based on the values of the objective function evaluated for sampled solutions.

Parameters

- **solutions** (*list/ndarray*) – candidate solutions returned from *ask()*
- **function_values** (*list*) – a list of objective function values evaluated for the sampled solutions.

class lagom.**CMAES**(*x0*, *sigma0*, *opts=None*)
Implements CMA-ES algorithm.

Note: It is a wrapper of the original CMA-ES implementation.

Parameters

- **x0** (*list*) – initial solution
- **sigma0** (*list*) – initial standard deviation
- **opts** (*dict*) – a dictionary of options, e.g. ['popsize', 'seed']

ask()
Sample a set of new candidate solutions.

Returns a list of sampled candidate solutions

Return type list

result
Return a namedtuple of all results for the optimization.

It contains: * xbest: best solution evaluated * fbest: objective function value of the best solution * evals_best: evaluation count when xbest was evaluated * evaluations: evaluations overall done * iterations: number of iterations * xfavorite: distribution mean in "phenotype" space, to be considered as current best estimate of the optimum * stds: effective standard deviations

tell(*solutions*, *function_values*)
Update the parameters of the population for a new generation based on the values of the objective function evaluated for sampled solutions.

Parameters

- **solutions** (*list/ndarray*) – candidate solutions returned from *ask()*
- **function_values** (*list*) – a list of objective function values evaluated for the sampled solutions.

class lagom.**CEM**(*x0*, *sigma0*, *opts=None*)

ask()
Sample a set of new candidate solutions.

Returns a list of sampled candidate solutions

Return type list

result
Return a namedtuple of all results for the optimization.

It contains: * xbest: best solution evaluated * fbest: objective function value of the best solution * evals_best: evaluation count when xbest was evaluated * evaluations: evaluations overall done * iterations: number of iterations * xfavorite: distribution mean in "phenotype" space, to be considered as current best estimate of the optimum * stds: effective standard deviations

tell(*solutions*, *function_values*)
Update the parameters of the population for a new generation based on the values of the objective function evaluated for sampled solutions.

Parameters

- **solutions** (*list/ndarray*) – candidate solutions returned from *ask()*
- **function_values** (*list*) – a list of objective function values evaluated for the sampled solutions.

CHAPTER 4

lagom.envs

class lagom.envs.**RecordEpisodeStatistics**(*env*, *deque_size=100*)

reset(***kwargs*)

Resets the state of the environment and returns an initial observation.

Returns the initial observation.

Return type observation (object)

step(*action*)

Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

class lagom.envs.**NormalizeObservation**(*env*, *clip=5.0*, *constant_moments=None*)

class lagom.envs.**NormalizeReward**(*env*, *clip=10.0*, *gamma=0.99*, *constant_var=None*)

reset()

Resets the state of the environment and returns an initial observation.

Returns the initial observation.

Return type observation (object)

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

class lagom.envs.**TimeStepEnv**(*env*)

reset(***kwargs*)
Resets the state of the environment and returns an initial observation.

Returns the initial observation.

Return type observation (object)

step(*action*)
Run one timestep of the environment's dynamics. When end of episode is reached, you are responsible for calling *reset()* to reset this environment's state.

Accepts an action and returns a tuple (observation, reward, done, info).

Parameters **action** (*object*) – an action provided by the agent

Returns agent's observation of the current environment reward (float) : amount of reward returned after previous action done (bool): whether the episode has ended, in which case further step() calls will return undefined results info (dict): contains auxiliary diagnostic information (helpful for debugging, and sometimes learning)

Return type observation (object)

CHAPTER 5

lagom.experiment

5.1 Config

class lagom.experiment.**Grid**(*values*)
A grid search over a list of values.

class lagom.experiment.**Sample**(*f*)

class lagom.experiment.**Condition**(*f*)

class lagom.experiment.**Config**(*items*, *num_sample=1*, *keep_dict_order=False*)
Defines a set of configurations for the experiment.

The configuration includes the following possible items:

- Hyperparameters: learning rate, batch size etc.
- Experiment settings: training iterations, logging directory, environment name etc.

All items are stored in a dictionary. It is a good practice to semantically name each item e.g. *network.lr* indicates the learning rate of the neural network.

For hyperparameter search, we support both grid search (*Grid*) and random search (*Sample*).

Call *make_configs()* to generate a list of all configurations, each is assigned with a unique ID.

note:

```
For random search over small positive float e.g. learning rate, it is recommended␣
↪to
use log-uniform distribution, i.e.
.. math::
    \text{logU}(a, b) \sim \exp(U(\log(a), \log(b)))

An example: `np.exp(np.random.uniform(low=np.log(low), high=np.log(high)))`

Because direct uniform sampling is very `numerically unstable`_.
```

Warning: The random seeds should not be set here. Instead, it should be handled by `BaseExperimentMaster` and `BaseExperimentWorker`.

Example:

```
>>> config = Config({'log.dir': 'some path', 'network.lr': Grid([1e-3, 5e-3]),
↪'env.id': Grid(['CartPole-v1', 'Ant-v2'])}, num_sample=1, keep_dict_order=False)
>>> import pandas as pd
>>> print(pd.DataFrame(config.make_configs()))
   ID       env.id    log.dir  network.lr
0   0  CartPole-v1  some path       0.001
1   1       Ant-v2  some path       0.001
2   2  CartPole-v1  some path       0.005
3   3       Ant-v2  some path       0.005
```

Parameters

- **items** (*dict*) – a dictionary of all configuration items.
- **num_sample** (*int*) – number of samples for random configuration items. If grid search is also provided, then the grid will be repeated `num_sample` of times.
- **keep_dict_order** (*bool*) – if `True`, then each generated configuration has the same key ordering with `items`.

make_configs()

Generate a list of all possible combinations of configurations, including grid search and random search.

Returns a list of all possible configurations

Return type list

5.2 Run experiment

lagom.experiment.**run_experiment**(*run*, *config*, *seeds*, *log_dir*, *max_workers*, *chunksize=1*, *use_gpu=False*, *gpu_ids=None*)

A convenient function to parallelize the experiment (master-worker pipeline).

It is implemented by using *concurrent.futures.ProcessPoolExecutor*

It automatically creates all subfolders for each pair of configuration and random seed to store the loggings of the experiment. The root folder is given by the user. Then all subfolders for each configuration are created with the name of their job IDs. Under each configuration subfolder, a set subfolders are created for each random seed (the random seed as folder name). Intuitively, an experiment could have following directory structure:

```
- logs
    - 0  # ID number
        - 123  # random seed
        - 345
        - 567
    - 1
        - 123
        - 345
        - 567
    - 2
```

(continues on next page)

(continued from previous page)

```
      - 123
      - 345
      - 567
  - 3
      - 123
      - 345
      - 567
  - 4
      - 123
      - 345
      - 567
```

Parameters

- **run** (*function*) – a function that defines an algorithm, it must take the arguments *(config, seed, device, logdir)*
- **config** (Config) – a *Config* object defining all configuration settings
- **seeds** (*list*) – a list of random seeds
- **log_dir** (*str*) – a string to indicate the path to store loggings.
- **max_workers** (*int*) – argument for ProcessPoolExecutor. if *None*, then all experiments run serially.
- **chunksize** (*int*) – argument for Executor.map()
- **use_gpu** (*bool*) – if *True*, then use CUDA. Otherwise, use CPU.
- **gpu_ids** (*list*) – if *None*, then use all available GPUs. Otherwise, only use the GPU device defined in the list.

CHAPTER 6

lagom.metric: Metrics

lagom.metric.**returns**(*gamma*, *rewards*)

lagom.metric.**bootstrapped_returns**(*gamma*, *rewards*, *last_V*, *reach_terminal*)

Return (discounted) accumulated returns with bootstrapping for a batch of episodic transitions.

Formally, suppose we have all rewards $(r_1, \dots, r_T)$, it computes

$$Q_t = r_t + \gamma r_{t+1} + \cdots + \gamma^{T-t} r_T + \gamma^{T-t+1} V(s_{T+1})$$

Note: The state values for terminal states are masked out as zero !

lagom.metric.**td0_target**(*gamma*, *rewards*, *Vs*, *last_V*, *reach_terminal*)

Calculate TD(0) targets of a batch of episodic transitions.

Let $r_1, r_2, \dots, r_T$ be a list of rewards and let $V(s_0), V(s_1), \dots, V(s_{T-1}), V(s_T)$ be a list of state values including a last state value. Let γ be a discounted factor, the TD(0) targets are calculated as follows

$$r_t + \gamma V(s_t), \forall t = 1, 2, \dots, T$$

Note: The state values for terminal states are masked out as zero !

lagom.metric.**td0_error**(*gamma*, *rewards*, *Vs*, *last_V*, *reach_terminal*)

Calculate TD(0) errors of a batch of episodic transitions.

Let $r_1, r_2, \dots, r_T$ be a list of rewards and let $V(s_0), V(s_1), \dots, V(s_{T-1}), V(s_T)$ be a list of state values including a last state value. Let γ be a discounted factor, the TD(0) errors are calculated as follows

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Note: The state values for terminal states are masked out as zero !

lagom.metric.**gae**(*gamma*, *lam*, *rewards*, *Vs*, *last_V*, *reach_terminal*)

Calculate the Generalized Advantage Estimation (GAE) of a batch of episodic transitions.

Let δ_t be the TD(0) error at time step t, the GAE at time step t is calculated as follows

$$A_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k}$$

lagom.metric.**vtrace**(*behavior_logprobs*, *target_logprobs*, *gamma*, *Rs*, *Vs*, *last_V*, *reach_terminal*, *clip_rho=1.0*, *clip_pg_rho=1.0*)

CHAPTER 7

lagom.networks: Networks

class lagom.networks.**Module**(***kwargs*)

Wrap PyTorch nn.module to provide more helper functions.

from_vec(*x*)

Set the network parameters from a single flattened vector.

Parameters **x** (*Tensor*) – A single flattened vector of the network parameters with consistent size.

load(*f*)

Load the network parameters from a file.

It complies with the recommended approach for saving a model in PyTorch documentation.

Parameters **f** (*str*) – file path.

num_params

Returns the total number of parameters in the neural network.

num_trainable_params

Returns the total number of trainable parameters in the neural network.

num_untrainable_params

Returns the total number of untrainable parameters in the neural network.

save(*f*)

Save the network parameters to a file.

It complies with the recommended approach for saving a model in PyTorch documentation.

Note: It uses the highest pickle protocol to serialize the network parameters.

Parameters **f** (*str*) – file path.

to_vec()

Returns the network parameters as a single flattened vector.

lagom.networks.**ortho_init**(*module*, *nonlinearity=None*, *weight_scale=1.0*, *constant_bias=0.0*)
Applies orthogonal initialization for the parameters of a given module.

Parameters

- **module** (*nn.Module*) – A module to apply orthogonal initialization over its parameters.
- **nonlinearity** (*str, optional*) – Nonlinearity followed by forward pass of the module. When nonlinearity is not `None`, the gain will be calculated and `weight_scale` will be ignored. Default: `None`
- **weight_scale** (*float, optional*) – Scaling factor to initialize the weight. Ignored when `nonlinearity` is not `None`. Default: 1.0
- **constant_bias** (*float, optional*) – Constant value to initialize the bias. Default: 0.0

Note: Currently, the only supported `module` are elementary neural network layers, e.g. nn.Linear, nn.Conv2d, nn.LSTM. The submodules are not supported.

Example:

```
>>> a = nn.Linear(2, 3)
>>> ortho_init(a)
```

lagom.networks.**linear_lr_scheduler**(*optimizer*, *N*, *min_lr*)
Defines a linear learning rate scheduler.

Parameters

- **optimizer** (*Optimizer*) – optimizer
- **N** (*int*) – maximum bounds for the scheduling iteration e.g. total number of epochs, iterations or time steps.
- **min_lr** (*float*) – lower bound of learning rate

lagom.networks.**make_fc**(*input_dim*, *hidden_sizes*)
Returns a ModuleList of fully connected layers.

Note: All submodules can be automatically tracked because it uses nn.ModuleList. One can use this function to generate parameters in `BaseNetwork`.

Example:

```
>>> make_fc(3, [4, 5, 6])
ModuleList(
  (0): Linear(in_features=3, out_features=4, bias=True)
  (1): Linear(in_features=4, out_features=5, bias=True)
  (2): Linear(in_features=5, out_features=6, bias=True)
)
```

Parameters

- **input_dim** (*int*) – input dimension in the first fully connected layer.
- **hidden_sizes** (*list*) – a list of hidden sizes, each for one fully connected layer.

Returns A ModuleList of fully connected layers.

Return type nn.ModuleList

lagom.networks.**make_cnn**(*input_channel*, *channels*, *kernels*, *strides*, *paddings*)
Returns a ModuleList of 2D convolution layers.

Note: All submodules can be automatically tracked because it uses nn.ModuleList. One can use this function to generate parameters in `BaseNetwork`.

Example:

```
>>> make_cnn(input_channel=3, channels=[16, 32], kernels=[4, 3], strides=[2, 1],␣
↪paddings=[1, 0])
ModuleList(
  (0): Conv2d(3, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
  (1): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
)
```

Parameters

- **input_channel** (*int*) – input channel in the first convolution layer.
- **channels** (*list*) – a list of channels, each for one convolution layer.
- **kernels** (*list*) – a list of kernels, each for one convolution layer.
- **strides** (*list*) – a list of strides, each for one convolution layer.
- **paddings** (*list*) – a list of paddings, each for one convolution layer.

Returns A ModuleList of 2D convolution layers.

Return type nn.ModuleList

lagom.networks.**make_transposed_cnn**(*input_channel*, *channels*, *kernels*, *strides*, *paddings*, *output_paddings*)
Returns a ModuleList of 2D transposed convolution layers.

Note: All submodules can be automatically tracked because it uses nn.ModuleList. One can use this function to generate parameters in `BaseNetwork`.

Example:

```
make_transposed_cnn(input_channel=3,
                    channels=[16, 32],
                    kernels=[4, 3],
                    strides=[2, 1],
                    paddings=[1, 0],
                    output_paddings=[1, 0])
ModuleList(
  (0): ConvTranspose2d(3, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),␣
↪output_padding=(1, 1))
  (1): ConvTranspose2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
)
```

Parameters

- **input_channel** (*int*) – input channel in the first transposed convolution layer.

- **channels** (*list*) – a list of channels, each for one transposed convolution layer.
- **kernels** (*list*) – a list of kernels, each for one transposed convolution layer.
- **strides** (*list*) – a list of strides, each for one transposed convolution layer.
- **paddings** (*list*) – a list of paddings, each for one transposed convolution layer.
- **output_paddings** (*list*) – a list of output paddings, each for one transposed convolution layer.

Returns A ModuleList of 2D transposed convolution layers.

Return type nn.ModuleList

class lagom.networks.**MDNHead**(*in_features*, *out_features*, *num_density*, ***kwargs*)

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the *Module* instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

loss(*logit_pi*, *mean*, *std*, *target*)

Calculate the MDN loss function.

The loss function (negative log-likelihood) is defined by:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\ln\left(\sum_{k=1}^{K}\prod_{d=1}^{D}\pi_k(x_{n,d})\mathcal{N}\left(\mu_k(x_{n,d}),\sigma_k(x_{n,d})\right)\right)$$

For better numerical stability, we could use log-scale:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\ln\left(\sum_{k=1}^{K}\exp\left\{\sum_{d=1}^{D}\ln\pi_k(x_{n,d}) + \ln\mathcal{N}\left(\mu_k(x_{n,d}),\sigma_k(x_{n,d})\right)\right\}\right)$$

Note: One should always use the second formula via log-sum-exp trick. The first formula is numerically unstable resulting in +/- `Inf` and `NaN` error.

The log-sum-exp trick is defined by

$$\log\sum_{i=1}^{N}\exp(x_i) = a + \log\sum_{i=1}^{N}\exp(x_i - a)$$

where $a = \max_i(x_i)$

Parameters

- **logit_pi** (*Tensor*) – the logit of mixing coefficients, shape [N, K, D]
- **mean** (*Tensor*) – mean of Gaussian mixtures, shape [N, K, D]
- **std** (*Tensor*) – standard deviation of Gaussian mixtures, shape [N, K, D]

- **target** (*Tensor*) – target tensor, shape [N, D]

Returns calculated loss

Return type Tensor

sample(*logit_pi*, *mean*, *std*, *tau=1.0*)

Sample from Gaussian mixtures using reparameterization trick.

- Firstly sample categorically over mixing coefficients to determine a specific Gaussian
- Then sample from selected Gaussian distribution

Parameters

- **logit_pi** (*Tensor*) – the logit of mixing coefficients, shape [N, K, D]
- **mean** (*Tensor*) – mean of Gaussian mixtures, shape [N, K, D]
- **std** (*Tensor*) – standard deviation of Gaussian mixtures, shape [N, K, D]
- **tau** (*float*) – temperature during sampling, it controls uncertainty. * If $\tau > 1$: increase uncertainty * If $\tau < 1$: decrease uncertainty

Returns sampled data with shape [N, D]

Return type Tensor

7.1 Recurrent Neural Networks

class lagom.networks.**LayerNormLSTMCell**(*input_size*, *hidden_size*)

class lagom.networks.**LSTMLayer**(*cell*, **cell_args*)

forward(*input*, *state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the *Module* instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class lagom.networks.**StackedLSTM**(*num_layers*, *layer*, *first_layer_args*, *other_layer_args*)

forward(*input*, *states*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the *Module* instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

lagom.networks.**make_lnlstm**(*input_size*, *hidden_size*, *num_layers=1*)

7.2 RL components

class lagom.networks.**CategoricalHead**(*feature_dim*, *num_action*, *device*, ***kwargs*)

Defines a module for a Categorical (discrete) action distribution.

Example

```
>>> import torch
>>> action_head = CategoricalHead(30, 4, 'cpu')
>>> action_head(torch.randn(2, 30))
Categorical(probs: torch.Size([2, 4]))
```

Parameters

- **feature_dim** (*int*) – number of input features
- **num_action** (*int*) – number of discrete actions
- **device** (*torch.device*) – PyTorch device
- ****kwargs** – keyword arguments for more specifications.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the *Module* instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class lagom.networks.**DiagGaussianHead**(*feature_dim*, *action_dim*, *device*, *std0*, ***kwargs*)

Defines a module for a diagonal Gaussian (continuous) action distribution which the standard deviation is state independent.

The network outputs the mean $\mu(x)$ and the state independent logarithm of standard deviation $\log \sigma$ (allowing to optimize in log-space, i.e. both negative and positive).

The standard deviation is obtained by applying exponential function $\exp(x)$.

Example

```
>>> import torch
>>> action_head = DiagGaussianHead(10, 4, 'cpu', 0.45)
>>> action_dist = action_head(torch.randn(2, 10))
>>> action_dist.base_dist
Normal(loc: torch.Size([2, 4]), scale: torch.Size([2, 4]))
>>> action_dist.base_dist.stddev
tensor([[0.4500, 0.4500, 0.4500, 0.4500],
        [0.4500, 0.4500, 0.4500, 0.4500]], grad_fn=<ExpBackward>)
```

Parameters

- **feature_dim** (*int*) – number of input features

- **action_dim** (*int*) – flat dimension of actions
- **device** (*torch.device*) – PyTorch device
- **std0** (*float*) – initial standard deviation
- ****kwargs** – keyword arguments for more specifications.

forward(*x*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the *Module* instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

CHAPTER 8

lagom.transform: Transformations

class lagom.transform.**Describe**(*count: int*, *mean: float*, *std: float*, *min: float*, *max: float*, *repr_indent: int = 0*, *repr_prefix: str = None*)

lagom.transform.**describe**(*x*, *axis=-1*, *repr_indent=0*, *repr_prefix=None*)

lagom.transform.**interp_curves**(*x*, *y*)

Piecewise linear interpolation of a discrete set of data points and generate new $x-y$ values from the interpolated line.

It receives a batch of curves with $x - y$ values, a global min and max of the x-axis are calculated over the entire batch and new x-axis values are generated to be applied to the interpolation function. Each interpolated curve will share the same values in x-axis.

Note: This is useful for plotting a set of curves with uncertainty bands where each curve has data points at different x values. To generate such plot, we need the set of y values with consistent x values.

Warning: Piecewise linear interpolation often can lead to more realistic uncertainty bands. Do not use polynomial interpolation which the resulting curve can be extremely misleading.

Example:

```
>>> import matplotlib.pyplot as plt

>>> x1 = [4, 5, 7, 13, 20]
>>> y1 = [0.25, 0.22, 0.53, 0.37, 0.55]
>>> x2 = [2, 4, 6, 7, 9, 11, 15]
>>> y2 = [0.03, 0.12, 0.4, 0.2, 0.18, 0.32, 0.39]

>>> plt.scatter(x1, y1, c='blue')
>>> plt.scatter(x2, y2, c='red')
```

(continues on next page)

(continued from previous page)

```
>>> new_x, new_y = interp_curves([x1, x2], [y1, y2], num_point=100)
>>> plt.plot(new_x[0], new_y[0], 'blue')
>>> plt.plot(new_x[1], new_y[1], 'red')
```

Parameters

- **x** (*list*) – a batch of x values.
- **y** (*list*) – a batch of y values.
- **num_point** (*int*) – number of points to generate from the interpolated line.

Returns

a tuple of two lists. A list of interpolated x values (shared for the batch of curves) and followed by a list of interpolated y values.

Return type tuple

lagom.transform.**geometric_cumsum**(*alpha*, *x*)

Calculate future accumulated sums for each element in a list with an exponential factor.

Given input data $x_1, \dots, x_n$ and exponential factor $\alpha \in [0, 1]$, it returns an array y with the same length and each element is calculated as following

$$y_i = x_i + \alpha x_{i+1} + \alpha^2 x_{i+2} + \cdots + \alpha^{n-i-1} x_{n-1} + \alpha^{n-i} x_n$$

Note: To gain the optimal runtime speed, we use `scipy.signal.lfilter`

Example

```
>>> geometric_cumsum(0.1, [1, 2, 3, 4])
array([[1.234, 2.34 , 3.4  , 4.   ]])
```

Parameters

- **alpha** (*float*) – exponential factor between zero and one.
- **x** (*list*) – input data

Returns calculated data

Return type ndarray

lagom.transform.**explained_variance**(*y_true*, *y_pred*, ***kwargs*)

Computes the explained variance regression score.

It involves a fraction of variance that the prediction explains about the ground truth.

Let $\hat{y}$ be the predicted output and let y be the ground truth output. Then the explained variance is estimated as follows:

$$\mathrm{EV}(y, \hat{y}) = 1 - \frac{\mathrm{Var}(y - \hat{y})}{\mathrm{Var}(y)}$$

The best score is 1.0, and lower values are worse. A detailed interpretation is as following:

- EV $= 1$: perfect prediction
- EV $= 0$: might as well have predicted zero
- EV < 0: worse than just predicting zero

Note: It calls the function from `scikit-learn` which handles exceptions better e.g. zero division, batch size.

Example

```
>>> explained_variance(y_true=[3, -0.5, 2, 7], y_pred=[2.5, 0.0, 2, 8])
0.9571734475374732
```

```
>>> explained_variance(y_true=[[3, -0.5, 2, 7]], y_pred=[[2.5, 0.0, 2, 8]])
0.9571734475374732
```

```
>>> explained_variance(y_true=[[0.5, 1], [-1, 1], [7, -6]], y_pred=[[0, 2], [-1,␣
↪2], [8, -5]])
0.9838709677419355
```

```
>>> explained_variance(y_true=[[0.5, 1], [-1, 10], [7, -6]], y_pred=[[0, 2], [-1,␣
↪0.00005], [8, -5]])
0.6704023148857179
```

Parameters

- **y_true** (*list*) – ground truth output
- **y_pred** (*list*) – predicted output
- ****kwargs** – keyword arguments to specify the estimation of the explained variance.

Returns estimated explained variance

Return type float

class lagom.transform.**LinearSchedule**(*initial*, *final*, *N*, *start=0*)

A linear scheduling from an initial to a final value over a certain timesteps, then the final value is fixed constantly afterwards.

Note: This could be useful for following use cases:

- Decay of epsilon-greedy: initialized with 1.0 and keep with `start` time steps, then linearly decay to `final` over `N` time steps, and then fixed constantly as `final` afterwards.
- Beta parameter in prioritized experience replay.

Note that for learning rate decay, one should use PyTorch `optim.lr_scheduler` instead.

Example

```
>>> scheduler = LinearSchedule(initial=1.0, final=0.1, N=3, start=0)
>>> [scheduler(i) for i in range(6)]
[1.0, 0.7, 0.4, 0.1, 0.1, 0.1]
```

Parameters

- **initial** (*float*) – initial value
- **final** (*float*) – final value
- **N** (*int*) – number of scheduling timesteps
- **start** (*int, optional*) – the timestep to start the scheduling. Default: 0

__call__(*x*)

Returns the current value of the scheduling.

Parameters **x** (*int*) – the current timestep.

Returns current value of the scheduling.

Return type float

lagom.transform.**rank_transform**(*x*, *centered=True*)

Rank transformation of a vector of values. The rank has the same dimensionality as the vector. Each element in the rank indicates the index of the ascendingly sorted input. i.e. `ranks[i] = k`, it means i-th element in the input is k-th smallest value.

Rank transformation reduce sensitivity to outliers, e.g. in OpenAI ES, gradient computation involves fitness values in the population, if there are outliers (too large fitness), it affects the gradient too much.

Note that a centered rank transformation to the range [-0.5, 0.5] is supported by an option.

Example

```
>>> rank_transform([3, 14, 1], centered=True)
array([ 0. ,  0.5, -0.5])
```

```
>>> rank_transform([3, 14, 1], centered=False)
array([1, 2, 0])
```

Parameters

- **x** (*list/ndarray*) – a vector of values.
- **centered** (*bool, optional*) – if `True`, then centered the rank transformation to $[-0.5, 0.5]$. Defualt: `True`

Returns an numpy array of ranks of input data

Return type ndarray

class lagom.transform.**PolyakAverage**(*alpha*)

Keep a running average of a quantity via Polyak averaging.

Compared with estimating mean, it is more sentitive to recent changes.

Parameters **alpha** (*float*) – factor to control the sensitivity to recent changes, in the range [0, 1]. Zero is most sensitive to recent change.

__call__(*x*)
Update the estimate.

Parameters **x** (*object*) – additional data to update the estimation of running average.

get_current()
Return the current running average.

class lagom.transform.**RunningMeanVar**(*shape*)
Estimates sample mean and variance by using Chan's method.

It supports for both scalar and multi-dimensional data, however, the input is expected to be batched. The first dimension is always treated as batch dimension.

Note: For better precision, we handle the data with *np.float64*.

Warning: To use estimated moments for standardization, remember to keep the precision *np.float64* and calculated as ..math:*frac{x - mu}{sqrt{sigma^2 + 10^{-8}}}*.

Example

```
>>> f = RunningMeanVar(shape=())
>>> f([1, 2])
>>> f([3])
>>> f([4])
>>> f.mean
2.499937501562461
>>> f.var
1.2501499923440393
```

__call__(*x*)
Update the mean and variance given an additional batched data.

Parameters **x** (*object*) – additional batched data.

n
Returns the total number of samples so far.

lagom.transform.**smooth_filter**(*x*, *window_length*, *polyorder*, ***kwargs*)
Smooth a sequence of noisy data points by applying Savitzky–Golay filter. It uses least squares to fit a polynomial with a small sliding window and use this polynomial to estimate the point in the center of the sliding window.

This is useful when a curve is highly noisy, smoothing it out leads to better visualization quality.

Example

```
>>> import matplotlib.pyplot as plt
```

```
>>> x = np.linspace(0, 4*2*np.pi, num=100)
>>> y = x*(np.sin(x) + np.random.random(100)*4)
>>> y2 = smooth_filter(y, window_length=31, polyorder=10)
```

```
>>> plt.plot(x, y)
>>> plt.plot(x, y2, 'red')
```

Parameters

- **x** (*list*) – one-dimensional vector of scalar data points of a curve.
- **window_length** (*int*) – the length of the filter window
- **polyorder** (*int*) – the order of the polynomial used to fit the samples

Returns an numpy array of smoothed curve data

Return type ndarray

class lagom.transform.**SegmentTree**(*capacity*, *operation*, *identity_element*)

Defines a segment tree data structure.

It can be regarded as regular array, but with two major differences

- Value modification is slower: O(ln(capacity)) instead of O(1)
- Efficient reduce operation over contiguous subarray: O(ln(segment size))

Parameters

- **capacity** (*int*) – total number of elements, it must be a power of two.
- **operation** (*lambda*) – binary operation forming a group, e.g. sum, min
- **identity_element** (*object*) – identity element in the group, e.g. 0 for sum

reduce(*start=0*, *end=None*)

Returns result of operation(A[start], operation(A[start+1], operation(... A[end - 1]))).

Parameters

- **start** (*int*) – start of segment
- **end** (*int*) – end of segment

Returns result of reduce operation

Return type object

class lagom.transform.**SumTree**(*capacity*)

Defines the sum tree for storing replay priorities.

Each leaf node contains priority value. Internal nodes maintain the sum of the priorities of all leaf nodes in their subtrees.

find_prefixsum_index(*prefixsum*)

Find the highest index *i* in the array such that sum(A[0] + A[1] + ... + A[i - 1]) <= prefixsum

if array values are probabilities, this function efficiently sample indices according to the discrete probability.

Parameters **prefixsum** (*float*) – prefix sum.

Returns highest index satisfying the prefixsum constraint

Return type int

sum(*start=0*, *end=None*)
Return A[start] + … + A[end - 1]

class lagom.transform.**MinTree**(*capacity*)

min(*start=0*, *end=None*)
Returns min(A[start], …, A[end])

CHAPTER 9

lagom.utils: Utils

lagom.utils.**set_global_seeds**(*seed*)

Set the seed for generating random numbers.

It sets the following dependencies with the given random seed:

1. PyTorch
2. Numpy
3. Python random

Parameters **seed** (*int*) – a given seed.

class lagom.utils.**Seeder**(*init_seed=0*)

A random seed generator.

Given an initial seed, the seeder can be called continuously to sample a single or a batch of random seeds.

Note: The seeder creates an independent RandomState to generate random numbers. It does not affect the RandomState in `np.random`.

Example:

```
>>> seeder = Seeder(init_seed=0)
>>> seeder(size=5)
[209652396, 398764591, 924231285, 1478610112, 441365315]
```

__call__(*size=1*)

Return the sampled random seeds according to the given size.

Parameters **size** (*int or list*) – The size of random seeds to sample.

Returns a list of sampled random seeds.

Return type list

9.1 Smart data type converter

lagom.utils.**tensorify**(*x*, *device*)

lagom.utils.**numpify**(*x*, *dtype=None*)

9.2 Use Python multiprocessing library

class lagom.utils.**ProcessWorker**(*master_conn*, *worker_conn*)
Base class for all workers implemented with Python multiprocessing.Process.

It communicates with master via a Pipe connection. The worker is stand-by infinitely waiting for task from master, working and sending back result. When it receives a `close` command, it breaks the infinite loop and close the connection.

work(*task_id*, *task*)
Work on the given task and return the result.

Parameters

- **task_id** (*int*) – the task ID.
- **task** (*object*) – a given task.

Returns working result.

Return type object

class lagom.utils.**ProcessMaster**(*worker_class*, *num_worker*)
Base class for all masters implemented with Python multiprocessing.Process.

It creates a number of workers each with an individual Process. The communication between master and each worker is via independent Pipe connection. The master assigns tasks to workers. When all tasks are done, it stops all workers and terminate all processes.

Note: If there are more tasks than workers, then tasks will be splitted into chunks. If there are less tasks than workers, then we reduce the number of workers to the number of tasks.

---

assign_tasks(*tasks*)
Assign a given list of tasks to the workers and return the received results.

Parameters **tasks** (*list*) – a list of tasks

Returns received results

Return type object

close()
Defines everything required after finishing all the works, e.g. stop all workers, clean up.

make_tasks()
Returns a list of tasks.

Returns a list of tasks

Return type list

9.3 Serialization

lagom.utils.**pickle_dump**(*obj*, *f*, *ext='.pkl'*)
Serialize an object using pickling and save in a file.

Note: It uses cloudpickle instead of pickle to support lambda function and multiprocessing. By default, the highest protocol is used.

Note: Except for pure array object, it is not recommended to use `np.save` because it is often much slower.

Parameters

- **obj** (*object*) – a serializable object
- **f** (*str/Path*) – file path
- **ext** (*str, optional*) – file extension. Default: .pkl

lagom.utils.**pickle_load**(*f*)
Read a pickled data from a file.

Parameters **f** (*str/Path*) – file path

lagom.utils.**yaml_dump**(*obj*, *f*, *ext='.yml'*)
Serialize a Python object using YAML and save in a file.

Note: YAML is recommended to use for a small dictionary and it is super human-readable. e.g. configuration settings. For saving experiment metrics, it is better to use *pickle_dump()*.

Note: Except for pure array object, it is not recommended to use `np.load` because it is often much slower.

Parameters

- **obj** (*object*) – a serializable object
- **f** (*str/Path*) – file path
- **ext** (*str, optional*) – file extension. Default: .yml

lagom.utils.**yaml_load**(*f*)
Read the data from a YAML file.

Parameters **f** (*str/Path*) – file path

class lagom.utils.**CloudpickleWrapper**(*x*)
Uses cloudpickle to serialize contents (multiprocessing uses pickle by default)

This is useful when passing lambda definition through Process arguments.

9.4 Misc

lagom.utils.**color_str**(*string*, *color*, *bold=False*)
Returns stylized string with coloring and bolding for printing.

Example:

```
>>> print(color_str('lagom', 'green', bold=True))
```

Parameters

- **string** (*str*) – input string
- **color** (*str*) – color name
- **bold** (*bool, optional*) – if `True`, then the string is bolded. Default: `False`

Returns stylized string

Return type out

lagom.utils.**timed**(*color='green'*, *bold=False*)
A decorator to print the total time of executing a body function.

Parameters

- **color** (*str, optional*) – color name. Default: 'green'
- **bold** (*bool, optional*) – if `True`, then the verbose is bolded. Default: `False`

lagom.utils.**timeit**(*_func=None*, ***, *color='green'*, *bold=False*)

lagom.utils.**ask_yes_or_no**(*msg*)
Ask user to enter yes or no to a given message.

Parameters **msg** (*str*) – a message

class lagom.utils.**IntervalConditioner**(*interval*, *mode*)

class lagom.utils.**NConditioner**(*max_n*, *num_conditions*, *mode*)

CHAPTER 10

lagom.vis: Visualization

class lagom.vis.**ImageViewer**(*max_width=500*)

Display an image from an RGB array in an OpenGL window.

Example:

```
imageviewer = ImageViewer(max_width=500)
image = np.asarray(Image.open('x.jpg'))
imageviewer(image)
```

__call__(*x*)

Create an image from the given RGB array and display to the window.

Parameters **x** (*ndarray*) – RGB array

close()

Close the Window.

class lagom.vis.**GridImage**(*ncol=8*, *padding=2*, *pad_value=0*)

Generate a grid of images. The images can be iteratively added.

Example:

```
grid = GridImage(ncol=8, padding=5, pad_value=0)

a = np.random.randint(0, 255+1, size=[10, 3, 64, 64])
grid.add(a)
grid()
```

Reference:

- https://github.com/pytorch/vision/blob/master/torchvision/utils.py
- https://github.com/facebookresearch/visdom/blob/master/py/visdom/__init__.py

Parameters

- **ncol** (*int, optional*) – Number of images to show in each row of the grid. Final grid size is [N/ncol, ncol]. Default: 8.
- **padding** (*int, optional*) – Number of paddings. Default: 2.
- **pad_value** (*float, optional*) – Padding value in the range [0, 255]. Black is 0 and white 255. Default: 0

__call__(***kwargs*)
Make grid of images.

Parameters ****kwargs** – keyword aguments used to specify the grid of images.

Returns a grid of image with shape [H, W, C] and dtype `np.uint8`

Return type Image

add(*x*)
Add a new data for making grid images.

Parameters **x** (*list/ndarray*) – a list or ndarray of images, with shape either [H, W], [C, H, W] or [N, C, H, W]

lagom.vis.**set_ticker**(*ax*, *axis='x'*, *num=None*, *KM_format=False*, *integer=False*)

lagom.vis.**read_xy**(*log_folder*, *file_name*, *get_x*, *get_y*, *smooth_out=False*, *smooth_kws=None*, *point_step=1*)

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

V

W

Y