ESTorch

Release 1.0.0

Jun 11, 2020

Contents:

`estorch.estorch.rank_transformation(rewards)`

Applies rank transformation to the returns.

Examples

```
>>> rewards = [-123, -50, 3, -5, 20, 10, 100]
>>> estorch.rank_transformation(rewards)
array([-0.5       , -0.33333333,  0.        , -0.16666667,  0.33333333,
        0.16666667,  0.5       ])
```

class `estorch.VirtualBatchNorm(num_features, eps=1e-05)`

Applies Virtual Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in paper *Improved Techniques for Training GANs*: https://arxiv.org/abs/1606.03498

$$y = \frac{x - \mathrm{E}[x_{\mathrm{ref}}]}{\sqrt{\mathrm{Var}[x_{\mathrm{ref}}] + \epsilon}} * \gamma + \beta$$

VirtualBatchNorm requires two forward passes. First one is to calculate mean and variance over a reference batch and second is to calculate the actual output.

Parameters

- **num_features** – C from an expected input of size (N, C, H, W)
- **eps** – a value added to the denominator for numerical stability. Default: 1e-5

class `estorch.ES(policy, agent, optimizer, population_size, sigma=0.01, device=device(type='cpu'), policy_kwargs={}, agent_kwargs={}, optimizer_kwargs={})`

Classic Evolution Strategy Algorithm. It optimizes given policy for the max reward return. For example usage refer to https://github.com/goktug97/estorch/blob/master/examples/cartpole_es.py

$$\nabla_\theta \mathbb{E}_{\epsilon\sim N(0,I)} F(\theta + \sigma\epsilon) = \frac{1}{\sigma} \mathbb{E}_{\epsilon\sim N(0,I)} \{F(\theta + \sigma\epsilon)\epsilon\}$$

- Evolution Strategies as a Scalable Alternative to Reinforcement Learning: https://arxiv.org/abs/1703.03864

Parameters

- **policy** – PyTorch Module. Should be passed as a `class`.
- **agent** – Policy will be optimized to maximize the output of this class's rollout function. For an example agent class refer to; https://github.com/goktug97/estorch/blob/master/examples/cartpole_es.py Should be passed as a `class`.
- **optimizer** – Optimizer that will be used to update parameters of the policy. Any PyTorch optimizer can be used. Should be passed as a `class`.
- **population_size** – Population size of the evolution strategy.

> **Note:** if you are using multiprocessing make sure `population_size` is multiple of `n_proc`

- **sigma** – Standart Deviation to use while sampling the generation from the policy.
- **device** – Torch device

Note: For every process a target network will be created to use during rollout. That is why I don't recommend use of `torch.device('cuda')`.

- **policy_kwargs** – This dictionary of arguments will passed to the policy during initialization.
- **agent_kwargs** – This dictionary of arguments will passed to the agent during initialization.
- **optimizer_kwargs** – This dictionary of arguments will passed to the optimizer during initialization.

Variables

- **policy** – Each step this policy is optimized. Only in master process.
- **optimizer** – Optimizer that is used to optimize the `policy`. Only in master process.
- **agent** – Used for rollout in each processes.
- **n_parameters** – Number of trainable parameters of the `policy`.
- **best_reward** – Best reward achived during the training.
- **episode_reward** – Reward of the policy after the optimization.
- **best_policy_dict** – PyTorch `state_dict` of the policy with the highest reward.
- **population_returns** – Current population's rewards.
- **population_parameters** – Parameter vectors of the current population.

log()

`log` function is called after every optimization step. This function can be used to interract with the model during the training. By default its contents are:

```
print(f'Step: {self.step}')
print(f'Episode Reward: {self.episode_reward}')
print(f'Max Population Reward: {np.max(self.population_returns)}')
print(f'Max Reward: {self.best_reward}')
```

For example usage; https://github.com/goktug97/estorch/blob/master/examples/early_stopping.py

terminate()

Terminates the training and sends terminate signal to other processes.

train(*n_steps*, *n_proc=1*, *hwthread=False*, *hostfile=None*)

Train Evolution Strategy algorithm for n_steps in n_proc processes.

Note: This function can not be called more than once in the same script if `n_proc` is set to more than 1 because it executes the same script `n_proc` times which means it will start from the beginning of the script everytime.

Parameters

- **n_steps** – Number of training steps.
- **n_proc** – Number of processes. Processes are used for rollouts.

- **hwthread** – A boolean value, if `True` use hardware threads as independent cpus. Some processors are hyperthreaded which means 1 CPU core is splitted into multiple threads. For example in Linux, *nproc* command returns number of cores and if that number doesn't work here set hwthread to `True` and try again.
- **hostfile** – If set, `n_proc` and `hwthread` will be ignored and the `hostfile` will be used to initialize multiprocessing. For more information visit https://github.com/open-mpi/ompi/blob/9c0a2bb2d675583934efd5e6e22ce8245dd5554c/README#L1904

Raises `RuntimeError` – train function can not be called more than once.

class `estorch.`**`NS_ES`**(*policy*, *agent*, *optimizer*, *population_size*, *sigma=0.01*, *meta_population_size=3*, *k=10*, *device=device(type='cpu')*, *policy_kwargs={}*, *agent_kwargs={}*, *optimizer_kwargs={}*)

Novelty Search Evolution Strategy Algorithm. It optimizes given policy for the max novelty return. For example usage refer to https://github.com/goktug97/estorch/blob/master/examples/nsra_es.py This class is inherited from the `ES` so every function that is described in the `ES` can be used in this class too.

$$\nabla_{\theta_t} \mathbb{E}_{e \sim N(0,I)} \left[N\left(\theta_t + \sigma\epsilon, A\right) | A \right] \approx \frac{1}{n\sigma} \sum_{i=1}^{n} N\left(\theta_t^i, A\right) \epsilon_i$$

Where $N\left(\theta_t^i, A\right)$ is calculated as;

$$N(\theta, A) = N\left(b\left(\pi_\theta\right), A\right) = \frac{1}{|S|} \sum_{j \in S} \left\| b\left(\pi_\theta\right) - b\left(\pi_j\right) \right\|_2$$

$$S = kNN\left(b\left(\pi_\theta\right), A\right)$$

$$= \left\{ b\left(\pi_1\right), b\left(\pi_2\right), \ldots, b\left(\pi_k\right) \right\}$$

- Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents http://papers.nips.cc/paper/7750-improving-exploration-in-evolution-strategies-for-deep-reinforcement-learning-via-a-population-of-novelty-seeking-pdf

Parameters

- **policy** – PyTorch Module. Should be passed as a `class`.
- **agent** – Policy will be optimized to maximize the output of this class's rollout function. For an example agent class refer to; https://github.com/goktug97/estorch/blob/master/examples/cartpole_es.py Should be passed as a `class`.
- **optimizer** – Optimizer that will be used to update parameters of the policy. Any PyTorch optimizer can be used. Should be passed as a `class`.
- **population_size** – Population size of the evolution strategy.

  Note: if you are using multiprocessing make sure `population_size` is multiple of `n_proc`

- **sigma** – Standart Deviation to use while sampling the generation from the policy.
- **meta_population_size** – Instead of one policy a meta population of policies are optimized during training. Each step a policy is chosen from the meta population. Probability of each policy is calculated as;

$$P\left(\theta^m\right) = \frac{N\left(\theta^m, A\right)}{\sum_{j=1}^{M} N\left(\theta^3, A\right)}$$

- **k** – Number of nearest neigbhours used in the calculation of the novelty.
- **device** – Torch device

Note: For every process a target network will be created to use during rollout. That is why I don't recommend use of `torch.device('cuda')`.

- **policy_kwargs** – This dictionary of arguments will passed to the policy during initialization.
- **agent_kwargs** – This dictionary of arguments will passed to the agent during initialization.
- **optimizer_kwargs** – This dictionary of arguments will passed to the optimizer during initialization.

Variables

- **meta_population** – List of (policy, optimizer) tuples.
- **idx** – Selected (policy, optimizer) tuple index in the current step.
- **agent** – Used for rollout in each processes.
- **n_parameters** – Number of trainable parameters.
- **best_reward** – Best reward achived during the training.
- **episode_reward** – Reward of the chosen policy after the optimization.
- **best_policy_dict** – PyTorch `state_dict` of the policy with the highest reward.
- **population_returns** – List of (novelty, reward) tuple of the current population.
- **population_parameters** – Parameter vectors of the current population that sampled from the chosen policy.

class estorch.**NSR_ES**(*policy*, *agent*, *optimizer*, *population_size*, *sigma=0.01*, *meta_population_size=3*, *k=10*, *device=device(type='cpu')*, *policy_kwargs={}*, *agent_kwargs={}*, *optimizer_kwargs={}*)

Quality Diversity Evolution Strategy Algorithm. It optimizes given policy for the max avarage of novelty and reward return. For example usage refer to https://github.com/goktug97/estorch/blob/master/examples/nsra_es.py This class is inherited from the `NS_ES` which inherits from `ES` so every function that is described in the `ES` can be used in this class too.

$$\theta_{t+1}^{m} \leftarrow \theta_{t}^{m} + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} \frac{f\left(\theta_{t}^{i,m}\right) + N\left(\theta_{t}^{i,m}, A\right)}{2} \epsilon_{i}$$

- Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents http://papers.nips.cc/paper/7750-improving-exploration-in-evolution-strategies-for-deep-reinforcement-learning-via-a-population-of-novelty-seeking-pdf

Parameters

- **policy** – PyTorch Module. Should be passed as a `class`.
- **agent** – Policy will be optimized to maximize the output of this class's rollout function. For an example agent class refer to; https://github.com/goktug97/estorch/blob/master/examples/cartpole_es.py Should be passed as a `class`.

- **optimizer** – Optimizer that will be used to update parameters of the policy. Any PyTorch optimizer can be used. Should be passed as a `class`.
- **population_size** – Population size of the evolution strategy.

Note: if you are using multiprocessing make sure `population_size` is multiple of `n_proc`

- **sigma** – Standart Deviation to use while sampling the generation from the policy.
- **meta_population_size** – Instead of one policy a meta population of policies are optimized during training. Each step a policy is chosen from the meta population. Probability of each policy is calculated as;

$$P\left(\theta^{m}\right)=\frac{N\left(\theta^{m}, A\right)}{\sum_{j=1}^{M} N\left(\theta^{3}, A\right)}$$

- **k** – Number of nearest neigbhours used in the calculation of the novelty.
- **device** – Torch device

Note: For every process a target network will be created to use during rollout. That is why I don't recommend use of `torch.device('cuda')`.

- **policy_kwargs** – This dictionary of arguments will passed to the policy during initialization.
- **agent_kwargs** – This dictionary of arguments will passed to the agent during initialization.
- **optimizer_kwargs** – This dictionary of arguments will passed to the optimizer during initialization.

Variables

- **meta_population** – List of (policy, optimizer) tuples.
- **idx** – Selected (policy, optimizer) tuple index in the current step.
- **agent** – Used for rollout in each processes.
- **n_parameters** – Number of trainable parameters.
- **best_reward** – Best reward achived during the training.
- **episode_reward** – Reward of the chosen policy after the optimization.
- **best_policy_dict** – PyTorch `state_dict` of the policy with the highest reward.
- **population_returns** – List of (novelty, reward) tuple of the current population.
- **population_parameters** – Parameter vectors of the current population that sampled from the chosen policy.

class estorch.**NSRA_ES**(*policy*, *agent*, *optimizer*, *population_size*, *sigma=0.01*, *meta_population_size=3*, *k=10*, *min_weight=0.0*, *weight_t=50*, *weight_delta=0.05*, *device=device(type='cpu')*, *policy_kwargs={}*, *agent_kwargs={}*, *optimizer_kwargs={}*)

Quality Diversity Evolution Strategy Algorithm. It optimizes given policy for the max weighted avarage of novelty and reward return. For example usage refer to https://github.com/goktug97/estorch/blob/master/examples/

nsra_es.py This class is inherited from the *NS_ES* which inherits from *ES* so every function that is described in the *ES* can be used in this class too.

$$\theta_{t+1}^{m} \leftarrow \theta_{t}^{m}+\alpha \frac{1}{n \sigma} \sum_{i=1}^{n} w f\left(\theta_{t}^{i, m}\right) \epsilon_{i}+(1-w) N\left(\theta_{t}^{i, m}, A\right) \epsilon_{i}$$

- Improving Exploration in Evolution Strategies for Deep Reinforcement Learning via a Population of Novelty-Seeking Agents http://papers.nips.cc/paper/7750-improving-exploration-in-evolution-strategies-for-deep-reinforcement-learning-via-a-population-of-novelty-seeking-pdf

Parameters

- **policy** – PyTorch Module. Should be passed as a `class`.
- **agent** – Policy will be optimized to maximize the output of this class's rollout function. For an example agent class refer to; https://github.com/goktug97/estorch/blob/master/examples/cartpole_es.py Should be passed as a `class`.
- **optimizer** – Optimizer that will be used to update parameters of the policy. Any PyTorch optimizer can be used. Should be passed as a `class`.
- **population_size** – Population size of the evolution strategy.

Note: if you are using multiprocessing make sure `population_size` is multiple of `n_proc`

- **sigma** – Standart Deviation to use while sampling the generation from the policy.
- **meta_population_size** – Instead of one policy a meta population of policies are optimized during training. Each step a policy is chosen from the meta population. Probability of each policy is calculated as;

$$P\left(\theta^{m}\right)=\frac{N\left(\theta^{m}, A\right)}{\sum_{j=1}^{M} N\left(\theta^{3}, A\right)}$$

- **k** – Number of nearest neigbhours used in the calculation of the novelty.
- **min_weight,weight_t,weight_delta** – If the max reward doesn't improve for `weight_t` the `weight` is lowered by `weight_delta` amount. It can't get lower than `min_weight`.
- **device** – Torch device

Note: For every process a target network will be created to use during rollout. That is why I don't recommend use of `torch.device('cuda')`.

- **policy_kwargs** – This dictionary of arguments will passed to the policy during initialization.
- **agent_kwargs** – This dictionary of arguments will passed to the agent during initialization.
- **optimizer_kwargs** – This dictionary of arguments will passed to the optimizer during initialization.

Variables

- **meta_population** – List of (policy, optimizer) tuples.
- **idx** – Selected (policy, optimizer) tuple index in the current step.
- **agent** – Used for rollout in each processes.
- **n_parameters** – Number of trainable parameters.
- **best_reward** – Best reward achived during the training.
- **episode_reward** – Reward of the chosen policy after the optimization.
- **best_policy_dict** – PyTorch `state_dict` of the policy with the highest reward.
- **population_returns** – List of (novelty, reward) tuple of the current population.
- **population_parameters** – Parameter vectors of the current population that sampled from the chosen policy.

Python Module Index

e

Index

E

L

N

R

T

V