ac

December 21, 2023

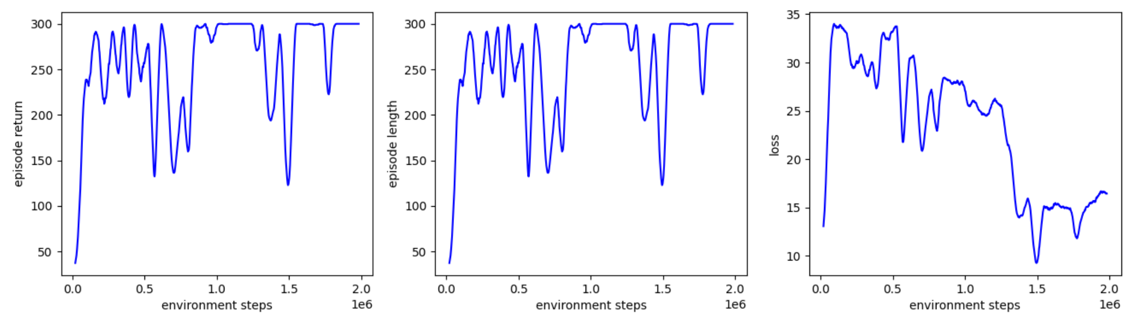
0.1 A3.1a) Run the given online REINFORCE algorithm

Read carefully through the provided code (only few classes have changed from exercise sheet 2) and run the given REINFORCE algorithm on the `Cartpole-v1` environment for 2 million (2M) steps. You can stop episodes after 200 steps. This can take 10-20 minutes.

[ ]:

```

```



0.2 A3.1b) Add a value bias to REINFORCE

Extend the `ReinforceLearner` class in the given Jupyter Notebook with a value function as bias (see slide 6 of Lecture 6). Implement two target-definitions for the value function, selected by the `'value_targets'` parameter: `'returns'` uses the returns `R_t` that are stored in the mini-batch, whereas `'td'` uses the TD-error. Make sure that the bias is ignored when the given parameter `'advantage_bias'` is `False`. Test your implementation as above on the evironment `Cartpole-v1` for 2M steps with the default parameters (using `'returns'` value targets).

Hint: The first heads of the model are interpreted as logits of a softmax policy, and the last head of the model is interpreted as the value function.

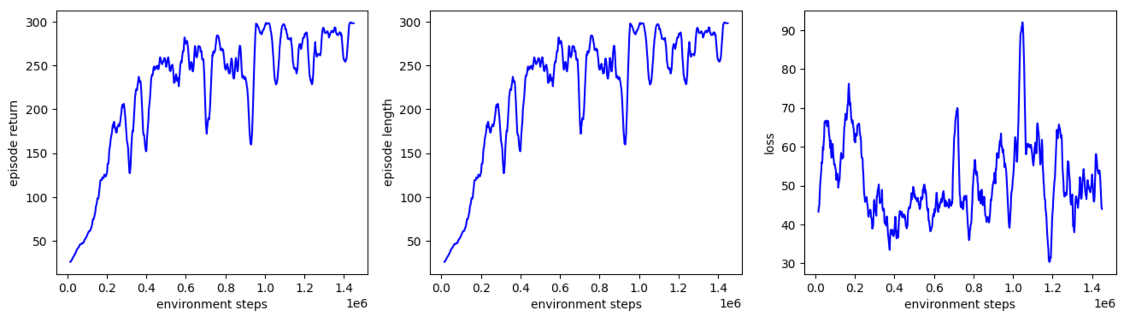
[ ]:

```python
class BiasedReinforceLearner (ReinforceLearner):
    def __init__(self, model, controller=None, params={}):
        super().__init__(model=model, controller=controller, params=params)
        self.advantage_bias = params.get('advantage_bias', True)
        self.value_targets = params.get('value_targets', 'returns')
        self.gamma = params.get('gamma')
        self.compute_next_val = (self.value_targets == 'td')
```

```
    # YOUR CODE HERE!!!
    def _advantages(self, batch, values=None, next_values=None):
        """ Computes the advantages, Q-values or returns for the policy loss.␣
↪"""
        return batch['returns'] - values

    def _value_loss(self, batch, values=None, next_values=None):
        """ Computes the value loss (if there is one). """
        if self.value_targets == "returns":
            target = batch["returns"]
        else:
            target = batch["rewards"] + self.gamma * next_values
        return (values - target).pow(2).mean()
```

[ ]:



0.3 A3.1c) Add an advantage to RENFORCE

Extend the `BiasedReinforceLearner` class in your implementation with an advantage function that uses bootstrapping (replaces `R_t` with `r_t + V(s_{t+1})`, see slide 6 of Lecture 6). Make sure the original behavior is maintained when the parameter `advantage_bootstrap` is `False`. Test your implementation as above on the evironment `Cartpole-v1` for 2M steps with the default parameters.

[ ]:

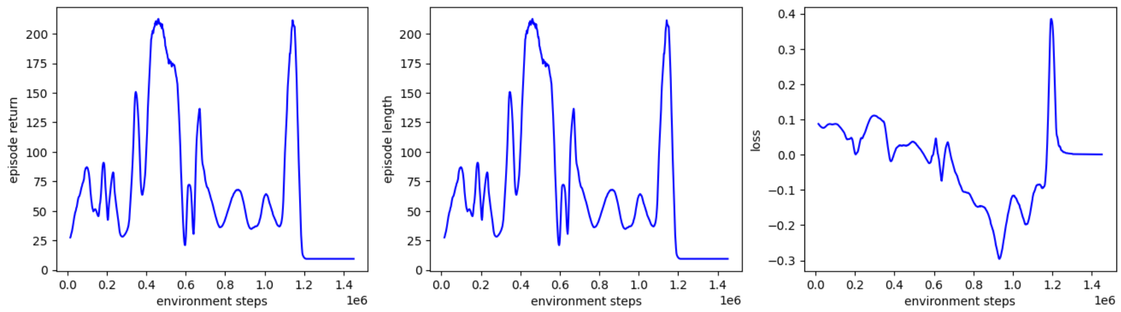
```
class ActorCriticLearner (BiasedReinforceLearner):
    def __init__(self, model, controller=None, params={}):
        super().__init__(model=model, controller=controller, params=params)
        self.advantage_bootstrap = params.get('advantage_bootstrap', True)
        self.compute_next_val = self.compute_next_val or self.
↪advantage_bootstrap

    # YOUR CODE HERE!!!

    def _advantages(self, batch, values=None, next_values=None):
```

```
        """ Computes the advantages, Q-values or returns for the policy loss.␣
↪"""
        if self.advantage_bootstrap:
            return batch["rewards"] * self.gamma * next_values.detach() -␣
↪values.detach()
        return batch['returns'] - values.detach()
```

[ ]:

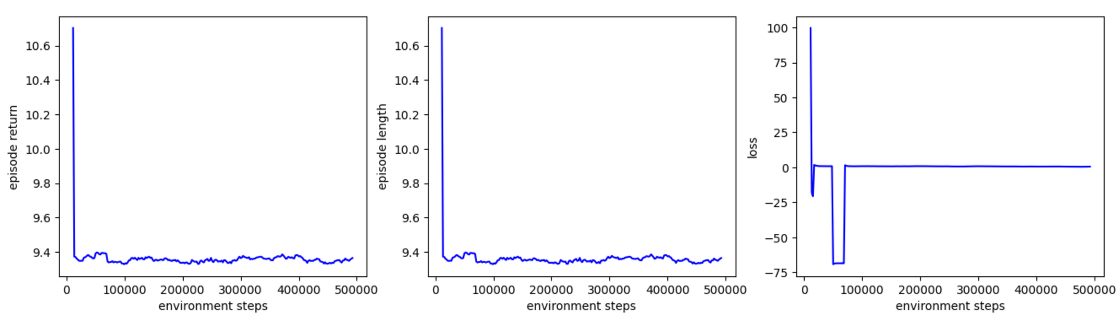


0.4 A3.1d) Extend the actor-critic algorithm with off-policy updates

Run your `ActorCriticLearner` with with 80 off-policy iterations (by setting the parameter `params['offpolicy_iterations'] = 80`). Extend your implementation to the class `OffpolicyActorCriticLearner`, that uses on-policy gradients in the first and off-policy gradients in all following iterations (`L_μ` on slide 15 of Lecture 6). Test your implementation as above on the evironment `Cartpole-v1` with the default parameters, but only for $500k$ steps.

Hint: `ReinforceLearner` has an attribute `old_pi` which is set to `None` at the beginning of `train()`. You can save the on-policy probabilities of the initial policy here to use them in the ratios of the off-policy loss.

[ ]:

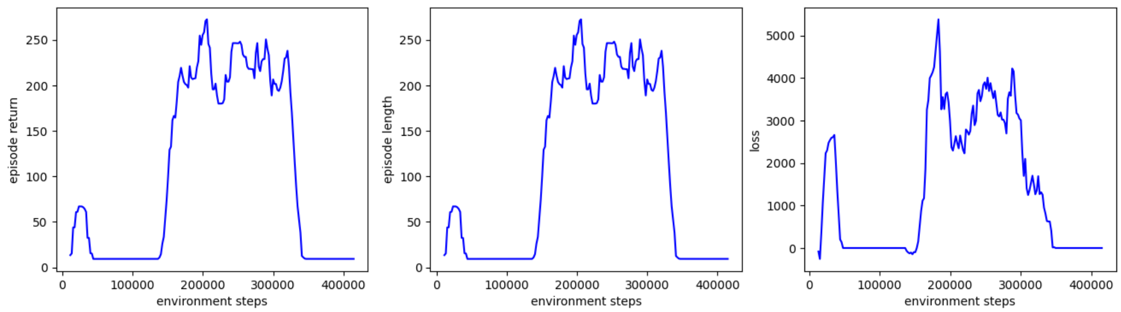


[ ]:

```
class OffpolicyActorCriticLearner (ActorCriticLearner):
    def __init__(self, model, controller=None, params={}):
        super().__init__(model=model, controller=controller, params=params)

    # YOUR CODE HERE!!!

    def _policy_loss(self, pi, advantages):
        """ Computes the policy loss. """
        if self.old_pi is None:
            self.old_pi = pi.clone().detach()
        else:
            pi /= self.old_pi
        return -(advantages.detach() * pi.log()).mean()
```



0.5 A3.1e) Add PPO clipping to the off-policy actor critic

Now extend `OffpolicyActorCriticLearner` by adding PPO clipping to the off-policy loss (`L_µ^{clip}` on slide 18 of Lecture 6). Test your implementation as above on the evironment `Cartpole-v1` with the default parameters, but only for 500k steps.

[ ]:

```
class PPOLearner (OffpolicyActorCriticLearner):
    def __init__(self, model, controller=None, params={}):
        super().__init__(model=model, controller=controller, params=params)
        self.ppo_clipping = params.get('ppo_clipping', False)
        self.ppo_clip_eps = params.get('ppo_clip_eps', 0.2)

    # YOUR CODE HERE!!!
    def _policy_loss(self, pi, advantages):
        """ Computes the policy loss. """
        if self.old_pi is None:
            self.old_pi = pi.clone().detach()
        else:
            pi /= self.old_pi
```

```
        pi_clipped = th.max(th.min(pi.log(), th.full(pi.shape, 1 - self.
↪ppo_clip_eps)), th.full(pi.shape, 1 + self.ppo_clip_eps))
        return -(th.min(advantages.detach() * pi.log(), advantages.detach() *␣
↪pi_clipped)).mean()
```