Attention-Over-Actions Option-Critic

Computer Science Extended Essay
Word Count:

Research Question

How are localized options trained in the Option-Critic architecture?

1 Introduction

As human we operate in high level actions. For example when driving a car, we make decisions about turning left or right instead of thinking about which muscle to contract. Human have the ability to group a chain of actions into one single high level action.

In Reinforcement Learning, we have a way to capture this idea of grouping action by using options [1]. When the options are defined, learning how to use them are very simple. However, if the options are not given and need to be learned, things get a lot harder since it requires knowing what makes an option good.

Many people argue that a good options should be diverse and localized, and many recent algorithms have followed this argument. Indeed, these algorithms have been able to discover options that are localized, so in this essay, I will try to derive a framework for localization in order to answer my research question, and I will develop an algorithm out of the framework to see whether it can actually train localized options.

This essay will be structured as follows: First, preliminary and related work will be presented to give a context to what I am trying to do. Second, previous work will be analyzed to figure out how localization is achieved. Third, a framework will be proposed based on the observations made in the analysis. Forth, an algorithm will be derived from the framework. Finally, the algorithm will be tested in the Four Rooms environment.

2 Preliminary

This section only acts as a summary. You are assumed to have basic knowledge about Reinforcement Learning and Options.

Markov Decision Process

Markov decision process (MDP) [4] is a mathematical framework for modeling decision making in a stochastic environment. It is defined as a tuple: $< \mathcal{S}, \mathcal{A}, r, \gamma, P >$ where:

$\mathcal{S}$ is the set of states.

$\mathcal{A}$ is the set of actions

$r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function

$\gamma \in [0, 1)$ is the discount factor that ensure the cumulative reward $\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$ converges

$P : \mathcal{S} \times A \times \mathcal{S} \to [0, 1]$ is the transition model which gives the probability for a particular transition to occur.

All MDPs must follow the Markov Property, which means that everything is stateless and does not depend on the history. In an MDP, A policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is responsible for choosing the action, after an action is chosen, the environment transitions to a new state according to $P(s, a, s')$, and a reward is given to the policy. This process repeats until the environment enters a terminal state.

Reinforcement Learning

Reinforcement Learning (RL) [4] is a machine learning paradigm which allows an agent to learn from interaction in an MDP. The agent's goal is to maximizes a certain objective function.

Actor-Critic [5] is one of the popular classes of algorithms that harvest the advantage of both Q-Learning and Policy Gradient. An actor network is trained to choose the best action, while a critic network is trained to evaluate the decision made by the actor network.

Option Framework

In the Option Framework[1], instead of only using 1 policy, we use a set of options, each option is defined as a tuple: $<\mathcal{I}_\omega, \pi_\omega, \beta_\omega>$, where:

$\mathcal{I}_\omega \subseteq \mathcal{S}$ is the initiation set that define which state the option can be selected

$\pi_\omega : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the internal policy

$\beta_\omega : \mathcal{S} \to [0, 1]$ is the termination probability.

These options take turns to choose the action. A policy-over-options $\pi_\Omega : \mathcal{S} \times \Omega \to [0, 1]$, where Ω is the set of options, decides which option to use. When an option is chosen, actions are chosen by its internal policy π_ω from then on, until it terminated according to its β_ω, then the policy-over-options π_Ω chooses an option again. An option has a chance to terminate every time environment transitions to a new state.

If the options are defined, the policy-over-options π_Ω can be learned by using SMDP Q-Learning [6] or Intra-Option Learning [7]. However, options are not always predefined and need to be discovered.

Four Rooms Environment

The Four Rooms environment is commonly used to evaluate algorithms that use options. The agent have 4 actions: up, down, left and right, which will move the agent in that corresponding direction. The choice of agent has 1/3 chance to fail and a random action will be chosen instead.

+50 reward will be given to the agent when it arrives to the goal state, and the episode will terminate after that.

Each cell position in the environment is mapped to a state number and is given to the agent in each step. The agent starts in a random state when an episode starts.

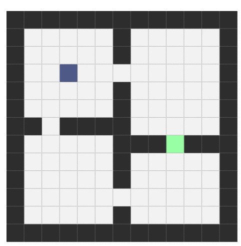

Figure 1: The Four Room environment. The black cells indicate the wall. The white cells indicate the area the agents can be in. The blue cell indicates the current position of the agent. The green cell indicates the goal.

3 Related Work

In this section, some option discovery algorithms will be summarized.

Option-Critic

Option-Critic [8] is an RL algorithm inspired by Action-Critic [5], where options are trained to maximize expected return, while an option-value function Q_Ω is trained to evaluate the decision the options.

Deliberation Cost

Since optimal policy can be achieved even without using options, if options are trained only to maximize expected return, they may degenerate and either terminate every steps or never terminate. Deliberation Cost [9] is a way to encourage longer option duration by punishing option switching.

Interest Option-Critic

The original Option-Critic assumes that options can be initiated everywhere, Interest Option-Critic [2] tries to remove this assumption by introducing interest functions $I : \mathcal{S} \times \Omega \to [0, 1]$ as a replacement for the initiation set. Experimental result shows that options learned by Interest Option-Critic is localized.

Termination-Critic

Termination-Critic [10] changes the objective of the termination function β from maximizing the expected return to minimize the entropy of the termination state. Since entropy can be interpreted as the information gain, this means minimizing the information gained from knowing the termination state, or in other words, making the termination state more predictable.

Attention Option-Critic

Attention Option-Critic [3] implements attention mechanism into Option-Critic. Different options are trained to attend to different features of the state. The attention units were trained to not only maximize the expected return, but also other things like maximizing difference between attention of different options.

4 Exploration

An analysis on localization will be conducted in this section.

What is Localization?

Localization is about options each responsible for a sub-task, or another way of looking at it is options each representing a skill. However, defining and measuring localization quantitatively is hard, which is why most work evaluate these option discovery algorithms qualitatively, by observing the agent acting for an episode in the environment. For example in the Four Rooms environment, only using one option in each room is considered as localization.

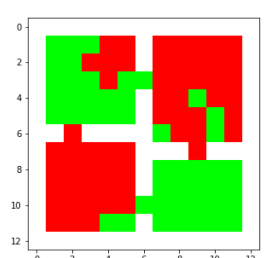

Figure 2: Red and green each represents an option, the options are pretty localized here.

Why Localization?

To understand why we want localization, first we need to answer a fundamental question: Why do we even use options in the first place? Is it to maximize expected return? However, optimal policy can be achieved using only primitive actions. If options cannot give us a higher return, why do we even need options? Some researchers suggest that options should speed up planning [9] [10] and also options should be transferable [2] [3].

If this is what a good option should be like, then a set of localized options would be beneficial. Localized options are easy to interpret, which made it easily reusable when transferred to a different environment. Also, easy-to-interpret options can speed up planning because each options have its clear purpose and usage.

How Localization is achieved?

Now I will analyze how some of the previous work achieve localization of options.

Attention Option-Critic

In Attention Option-Critic [3], each options are trained to attend to different features of the state. My hypothesis is that the attention mechanism can act as a constraint on what kind of policy each option can have. Each features of the state represents a piece of information about the state. When performing a sub-task, not all the features are necessary. Each sub-task requires different subset of features. Since the attention mechanism limits the subset of features given to an option, the option cannot learn sub-task that requires features outside of the subset of features it was given, or else the option will perform poorly. In the algorithm, each option is trained to have diverse attention, which force each option to learn to complete a different sub-task.

For example, there is 3 options and an RGB 2D image is the features of the state. Suppose the 3 options each attend to one of the RGB channels, and one of the sub-tasks is checking if there is a purple circle on the image. In this case, only the option with attention on the green channel can complete this sub-task.

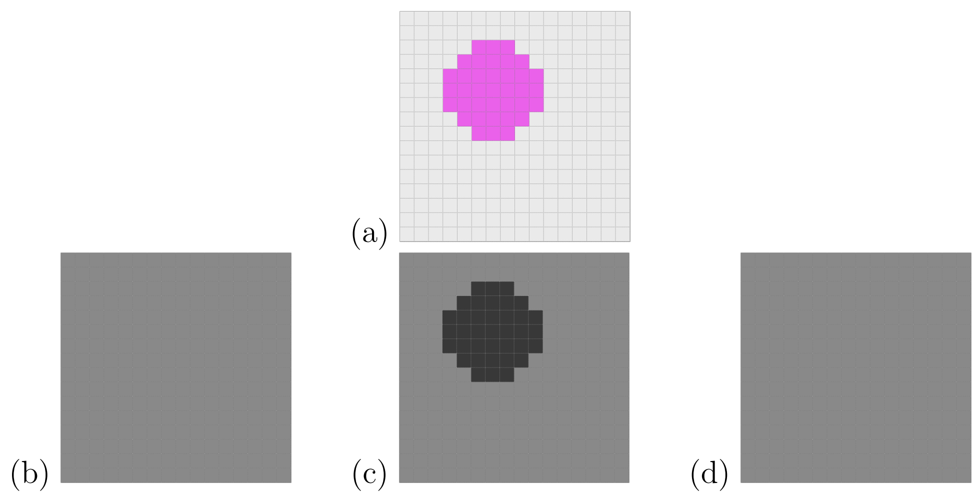

Figure 3: (a) is the RGB input, (b), (c) and (d) are the red, green and blue channel respectively. The circle can only be seen in the green channel.

Deliberation Cost

In Deliberation Cost [9], options are encouraged to be more temporally extended. Since options that terminates every step must be non-localized, Deliberation Cost can increase the chance of achieving localization.

Termination-Critic

In Termination-Critic [10], option termination states' entropy is being minimized, and experimental results show that option trained by this usually choose to terminate in bottleneck states (frequently visited states). My hypothesis is that bottleneck states are usually the start or end of a sub-task, having the option terminate at these states essentially chains termination with initiation.

I will illustrate this with a simple example: In the Four Rooms environment, assume that the sub-task is walking from one doorway to another. The two doorway are bottleneck states because the agent must go through them. Since the agent can take on many paths, all the other states are not bottleneck states. When the agent get to the next doorway, another option can be immediately initiated.

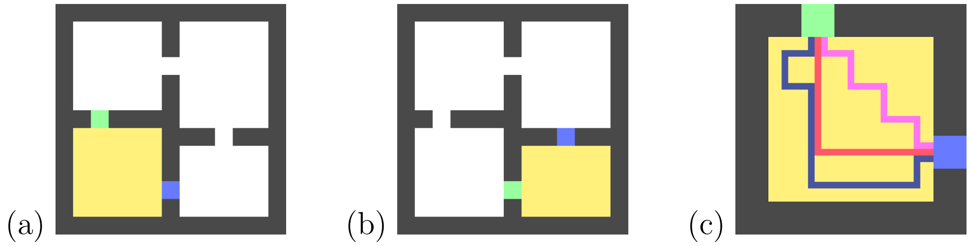

Figure 4: (a) and (b) are two options in the theoretic example. Green is the start of the option, blue is the end of the option, yellow is the intermediate states the option may encounter. (c) is a zoomed-in version of the bottom left room. Dark blue, red and pink lines are some of the paths the option can take.

Interest Option-Critic

In Interest Option-Critic. My hypothesis is that the interest function made the policy-over-options bias to choosing one of the options in a state, and since the paper uses a neural network as the interest function, the policy-over-options will also bias to choosing that option in a neighboring states.

5 The Localization Framework

Naturally, the next question that will be asked is: What do all of these algorithms have in common? Now I will propose a framework that can act as an abstraction for all of these algorithms.

The Localization Framework

1. **Grouping**
   Group states into meaningful sub-tasks based on a certain criterion
2. **Assignment**
   Assign the sub-tasks to different options
3. **Optimization**
   Train options to perform well in the sub-task it is given and also improve the initial grouping of sub-tasks
4. **Selection**
   Form the policy-over-options to select option in different state

---

This framework is inspired by Adaboost [11], which is an Ensemble Learning algorithm from Supervised Learning. There are a lot of similarities between Ensemble Learning and Option Learning, this has already been pointed out in previous work [12], the individual weak classifiers can be thought of as options.

Each weak classifier is responsible for classifying a small subset of the training data, just like how each option is responsible for a sub-task. In Adaboost, a bunch of weak classifiers are trained sequentially, each of them focuses on training data that is classified poorly by the previous weak classifiers.

Since this training process involves dividing training example into groups, then assign it to different weak classifiers, it inspires the Grouping and Assignment steps in the Localization Framework. Also, the

Optimization step in the framework is reminiscent of the weak classifiers learning to classify the training examples. After a lot of weak classifiers are trained, Adaboost combines them together to form a boosted classifier. The boosted classifier is the weighted sum of all the weak classifiers, the weighting is somewhat like a selection process, so it inspires the Selection step in the framework.

	Grouping	**Assignment**	**Optimization**	**Selection**
Attention Option-Critic	Group states that need the same sets of features	The algorithm assign an attention mechanism to each option	Options and attention mechanisms maximize return	Choose the option with maximum expected return
Termination-Critic	Group states between two bottleneck states	Each option terminates in a bottleneck state	Internal policy maximize return, termination function minimize entropy	Choose the option with maximum expected return
Interest Option-Critic	Group states that are close together	The algorithm assign an interest function to each option	Options and interest functions maximize return	Choose the option with high expected return and interest

Table 1: Previously mentioned algorithms can be fitted into the Localization Framework

This framework is the abstraction of algorithms that produce localized options, so it is very useful in deriving a new algorithm in the next section.

6 Attention-Over-Actions Option-Critic

Now that there is a framework, I can just follow the framework and derive a new algorithm. The following algorithm will be called Attention-Over-Actions Option-Critic because it perform abstraction on the action space, this algorithm is largely inspired by Attention Option-Critic.

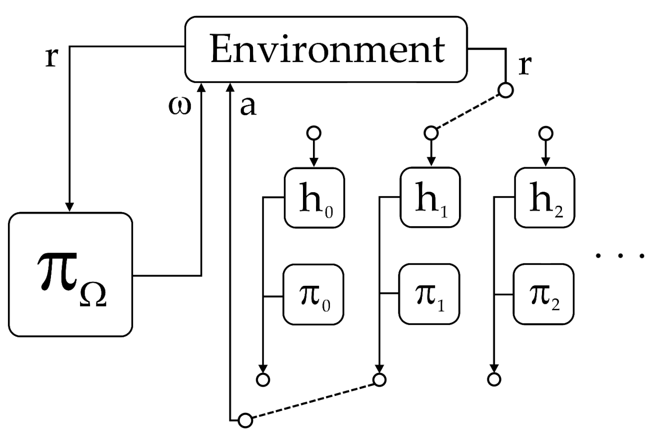

Figure 5: Visualization of the interaction between the environment and the Attention-Over-Actions Option-Critic algorithm.

1. Grouping

In Attention Option-Critic, the Grouping step divide sub-tasks based on features needed, this works because each sub-task requires a different subset of features. The following algorithm borrow the idea of attention over state features and apply it to actions, so each option will attend to a different sets of actions.

The intuition for this is that each sub-task will not need to use all the actions, for example, you would not consider performing a kicking action when you are driving a car. This is essentially performing actions abstraction, since the option can only attend to a subset of the actions.

2. Assignment

The following algorithm assigns the sub-task in a similar way as Attention Option-Critic, an attention mechanism $h_{\omega,\phi} : \mathcal{A} \to [0, 1]$ parameterized by ϕ will be given to each option. But instead of masking the state features, it masks the probability for choosing each actions. So the final probability π_{h_ω} for option ω to choose action a will be:

$$\pi_{h_\omega}(a|s) = \frac{\pi_{\omega,\theta}(a|s)h_{\omega,\phi}(a)}{\sum_{a'} \pi_{\omega,\theta}(a'|s)h_{\omega,\phi}(a')}$$

where $\pi_{\omega,\theta}$ is the internal policy of option ω and is parameterized by θ.

3. Optimization

The optimization of the attention mechanism and that of the option will be described separately.

Option Optimization

Let's first consider the optimization of the option. I will train the internal policy and termination function just like in Option-Critic. They will maximize the expected return by using gradient ascent:

$$\theta \leftarrow \theta + \alpha_\theta \nabla_\theta Q_\Omega(s, \omega)$$

$$\nu \leftarrow \nu + \alpha_\nu \nabla_\nu U_\Omega(s', \omega)$$

where θ and ν are the parameters for the internal policy and termination function respectively, $Q_\Omega(s, \omega)$ and $U_\Omega(s', \omega)$ are the expected return for choosing the option ω in state s and entering the state s when using option ω respectively.

I can directly reuse the result from the Option-Critic paper:

$$\nabla_\theta Q_\Omega(s, \omega) = E[\nabla_\theta \log \pi_{\omega,\theta}(a|s) Q_U(s, \omega, a)]$$

$$\nabla_\nu U_\Omega(s', \omega) = E[\nabla_\nu \beta_{\omega,\nu}(s') A(s', \omega)]$$

where $Q_U(s, \omega, a)$ is the expected return for choosing action a when using option ω in state s.

The reason why I ignored the attention mechanism here is that the internal policy should not know the about the attention mechanism, or else it may want to revert the effect of the attention mechanism, i.e. Assigning a high weight to an action with low attention.

Attention Mechanism Optimization

Now let's consider the optimization of $h_{\omega,\phi}$. I want the grouping to be different for all options because or else all options will just aim for the sub-task with highest return. I also want the options to focus on as little actions as possible while still having acceptable performance. Essentially what I need is the algorithm to consider the trade-off between these objectives and achieve a balance between them. A nice way to do this is to add all of these objective up and then perform gradient ascent on the sum:

$$\phi \leftarrow \phi + \alpha_\phi \nabla_\phi \sum_o (w_o O_o)$$

where o is the index of an objective, w_o is the weight of the objective, O_o is the objective function. This method has been used for Attention Option-Critic too. Now I will list out the objectives that I want the option to consider:

1. Perform well
2. Different from other options
3. The components of the attention mechanism is close to 0 or 1
4. Focus on small set of actions

For the first objective, I can just use $Q_\Omega(s, \omega)$ like in Attention Critic.

$$\max_h O_1 = \max_h Q_\Omega(s, \omega)$$

For the second objective, I will minimize cosine similarity just like in Attention Critic.

$$\min_h O_2 = \min_h \sum_{h' \neq h} C(h, h') = \min_h \sum_{h' \neq h} \frac{< h', h >}{||h'|| \times ||h||}$$

For the third objective, I will minimize entropy in the attention mechanism. Entropy measures the uncertainty in a probability distribution, so h should be normalized first.

$$\max_h O_3 = \max_h H(\frac{h}{||h||}) = \max_h < \frac{h}{||h||}, \log \frac{h}{||h||} >$$

For the forth objective, I will minimize the length of the attention mechanism, which discourage focusing on too many actions.

$$\min_h O_4 = \min_h ||h||$$

4. Selection

Any policy-over-options that favor higher Q-value options will work in this case, because the option will need the right set of actions in order to perform well, or else it will fail horribly. So the Q-value already encoded which option has the right set of actions. This means that policies like ϵ-greedy should work for this algorithm.

Algorithm 1 Pseudocode for Attention-Over-Actions Option-Critic (AOAOC)

$s \leftarrow s_0$
Choose ω according to the policy-over-options $\pi_\Omega(s)$
repeat
  Choose a according to $\pi_{h_\omega}(a|s)$
  Take action a in s, observe s', r

  1. Options evaluation:
  $\delta \leftarrow r - Q_U(s, \omega, a)$
  if s' is non-terminal **then**
    $\delta \leftarrow \delta + \gamma(1 - \beta_{\omega,\nu}(s'))Q_\Omega(s', \omega) + \gamma\beta_{\omega,\nu}(s') \max_{\omega'} Q_\Omega(s', \omega')$
  end if
  $Q_U(s, \omega, a) \leftarrow Q_U(s, \omega, a) + \alpha\delta$

  1. Options improvement:
  $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta \log \pi_{\omega,\theta}(a|s) Q_U(s, \omega, a)$
  $\nu \leftarrow \nu + \alpha_\nu \nabla_\nu \beta_{\omega,\nu}(s')(Q_\Omega(s', \omega) - V_\Omega(s'))$
  $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi \sum_o (w_o O_o)$
  if $\beta_{\omega,\nu}$ terminates in s' **then**
    choose new ω according to the policy-over-options $\pi_\Omega(s)$
  end if
  $s \leftarrow s'$
until s' is terminal

7 Experiment

In this section the algorithm AOAOC will be tested in the Four Rooms environment.

Initial Setup of the Experiment

Q_Ω and Q_U each is represented as a tensor. I do not need to use a neural network here since the state space is discrete in Four Rooms.

π_Ω is an ϵ-greedy policy for the Q_Ω, i.e. the policy-over-options has a probability of $1-\epsilon+\frac{\epsilon}{n_\Omega}$ to select the best option, where n_Ω is the number of options.

ν is a tensor, and $\beta_{\omega,\nu}$ is parameterized by ν. More precisely, $\beta_{\omega,\nu}=\sigma(\nu)$, where σ is the sigmoid function $\frac{1}{1+e^{-x}}$. This can ensure that $\beta_{\omega,\nu}$ is a probability.

θ is also a tensor, and $\pi_{\omega,\theta}$ is parameterized by θ. More precisely, $\pi_{\omega,\theta}$ is the softmax over the components of different actions in θ. This can ensure that $\pi_{\omega,\theta}$ is a probability distribution over actions.

ϕ is also a tensor, and $h_{\omega,\phi}$ is parameterized by ϕ. Just like as $\beta_{\omega,\nu}$, $h_{\omega,\phi}=\sigma(\phi)$. This can ensure that $h_{\omega,\phi}$ is between 0 and 1.

Also, I would like the agent to have a larger incentive to go to the goal in a shorter duration, so I added a punishment of -2 for each step taken by the agent.

Problematic Attention Mechanism

When running the experiment, two problem is encountered:

1. The attention mechanism h_ω often becomes all one.
2. The attention objectives conflict with one another.

Problem 1 leads to degenerate attention mechanism because the option is paying the same amount of attention to all actions. Suppose each of the component in the attention mechanism is a constant k, i.e. $h_\omega=[k,k,k,...]$. The final policy will be:

$$\pi_{h_\omega}(a|s)=\frac{k\times\pi_\omega(a|s)}{\sum_{a'}k\times\pi_\omega(a'|s)}=\frac{\pi_\omega(a|s)}{\sum_{a'}\pi_\omega(a'|s)}$$

which is the same as not using attention at all.

This problem is probably caused by the objective 1, which is to maximizing $Q_\Omega(s,\omega)$. Having a uniform attention always yield a higher return because there will not be a constraint to which action it can use.

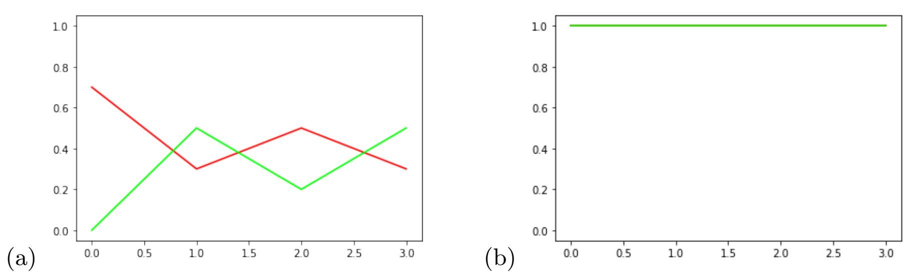

Figure 6: x-axis is the actions. y-axis is the attention on that action. Red and green lines each represents an attention mechanism. (a) is what the attention mechanism should look like. (b) is when it becomes all ones.

Problem 2 is follows directly from problem 1. Objective 1 will try to increase all the components to 1. Objective 3 will try to make only one of the components to 1, while the others all 0. Objective 4 will try to decrease all the components to 0.

My original intention is that the algorithm will automatically balance each objective and find the optimal attention mechanism. However, it turns out that the attention mechanism almost always result in 1 of 3 situations:

1. All components are 1

2. All components are 0
3. Only 1 of the component is one, while the others are all zero

For most of the time, 1 of the objectives completely dominates and results in 1 of the 3 situations above.

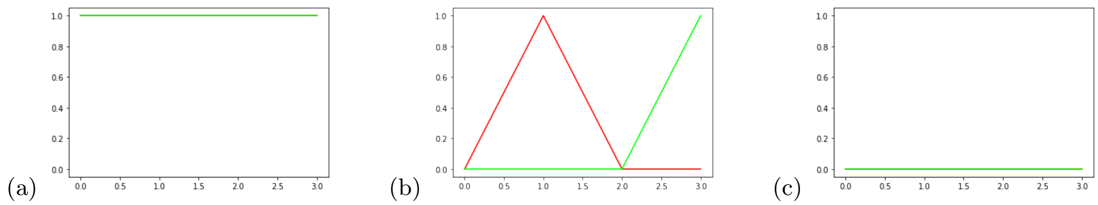

Figure 7: (a), (b) and (c) is the perfect end result for objective 1,3 and 4 respectively. All of which are degenerated attention mechanism.

8 Conclusion

9 Future Direction

This algorithm currently only focuses on discrete action space, a possible future direction might be extending this to continuous space. For vector actions, it might be good to attend to only some of the components of the vector, while ignoring other components by either randomly selecting values for them or keeping them the same as in the last state of the previous option. For scalar action, one direction is to use a one dimensional gaussian distribution as the attention mechanism, and multiply it with the original action distribution. However, some sort of trick may need to be deployed to speed up the process of normalization.

Bibliography

[1] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1, pp. 181 – 211, 1999.

[2] K. Khetarpal, M. Klissarov, M. Chevalier-Boisvert, P.-L. Bacon, and D. Precup, "Options of interest: Temporal abstraction with interest functions," 2020.

[3] R. Chunduru and D. Precup, "Attention option-critic," 2020.

[4] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning series, MIT Press, 2018.

[5] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," 2000.

[6] S. Bradtke and M. Duff, "Reinforcement learning methods for continuous-time markov decision problems," 12 1994.

[7] R. Sutton, D. Precup, and S. Singh, "Intra-option learning about temporally abstract actions.," pp. 556–564, 01 1998.

[8] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," 2016.

[9] J. Harb, P.-L. Bacon, M. Klissarov, and D. Precup, "When waiting is not an option : Learning options with a deliberation cost," 2017.

[10] A. Harutyunyan, W. Dabney, D. Borsa, N. Heess, R. Munos, and D. Precup, "The termination critic," 2019.

[11] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning Theory* (P. Vitányi, ed.), (Berlin, Heidelberg), pp. 23–37, Springer Berlin Heidelberg, 1995.

[12] S. Zhang, H. Chen, and H. Yao, "Ace: An actor ensemble algorithm for continuous control with tree search," 2018.

[13] K. Khetarpal, "ioc repository." `https://github.com/kkhetarpal/ioc`, 2020.

Appendix

A Notations

Markov Decision Process
$\mathcal{S}$ — Set of states
$\mathcal{A}$ — Set of actions
r — Reward function or reward
γ — Discount factor
P — Transition model
π — Policy
s — State
a — Action
s_T — Terminal state

Option Framework
Ω — Set of options
$\pi_{\omega,\theta}$ — Internal policy of option ω
β_ω — Termination probability of option ω
$\mathcal{I}_\omega$ — Initiation set of option ω
π_Ω — Policy-over-options
Q_U — Expected return for choosing an action
Q_Ω — Expected return for choosing an option
A_Ω — Expected advantage for choosing an option
V_Ω — Expected return in a state
U_Ω — Expected return for arriving in a state
θ — Parameter for internal policy
ν — Parameter for termination probability
ω — option

Attention-Over-Actions Option-Critic
$h_{\omega,\phi}$ — Attention Mechanism
π_{h_ω} — Final probability
ϕ — Parameter for attention mechanism
α — Learning rate for Q_U
α_θ — Learning rate for internal policy
α_ν — Learning rate for termination probability
α_ϕ — Learning rate for attention mechanism
O_o — Objective function
w_o — Weight of objective function
o — Objective
δ — One step Q-value error

Operations
∇ — Gradient
$E[\,]$ — Expected value

$\leftarrow$ — Assignment
$<,>$ — Dot product
$||.||$ — Length of a vector

B Proof

B.1 Derivative of Objective 1

Let h_ω^a be the element in the attention vector h_ω that corresponds to the action a, i.e. $h_\omega = [h_\omega^{a_0}, h_\omega^{a_1}, ...]$

$$\frac{\partial Q_\Omega(s,\omega)}{\partial h_\omega^a} = \sum_{s,\omega} \mu_\Omega(s,\omega|s_0,\omega_0) \sum_a \frac{\partial \pi_{h_\omega}(a|s)}{\partial h_\omega^a} Q_U(s,\omega,a)$$

$$= E_{s,\omega,a\sim\pi_{h_\omega}}[\frac{\partial \log \pi_{h_\omega}(a|s)}{\partial h_\omega^a} Q_U(s,\omega,a)]$$

The steps above follows directly from the Option-Critic appendix. The next step is to express $\frac{\partial \log \pi_{h_\omega}(a|s)}{\partial h_\omega^a}$ in a simpler form.

$$\frac{\partial \log \pi_{h_\omega}(a|s)}{\partial h_\omega^a} = \frac{\partial}{\partial h_\omega^a}[\log \frac{\pi_\omega(a|s) h_\omega^a}{\sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}}]$$

$$= \frac{\partial}{\partial h_\omega^a}[\log \pi_\omega(a|s) + \log h_\omega^a - \log \sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}]$$

$$= \frac{\partial}{\partial h_\omega^a}[\log \pi_\omega(a|s)] + \frac{\partial}{\partial h_\omega^a}[\log h_\omega^a] - \frac{\partial}{\partial h_\omega^a}[\log \sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}]$$

$\pi_\omega(a|s)$ is independent of h_ω^a, hence $\frac{\partial}{\partial h_\omega^a}[\log \pi_\omega(a|s)] = 0$

$$= \frac{\partial}{\partial h_\omega^a}[\log h_\omega^a] - \frac{\partial}{\partial h_\omega^a}[\log \sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}]$$

$$= \frac{1}{h_\omega^a} - \frac{\pi_\omega(a|s)}{\sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}}$$

$$= \frac{1}{h_\omega^a} - \frac{\pi_\omega(a|s)}{\sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}} \times \frac{h_\omega^a}{h_\omega^a}$$

By definition $\pi_{h_\omega}(a|s) = \frac{\pi_\omega(a|s) h_\omega^a}{\sum_{a'} \pi_\omega(a'|s) h_\omega^{a'}}$

$$= \frac{1}{h_\omega^a} - \frac{\pi_{h_\omega}(a|s)}{h_\omega^a}$$

$$= \frac{1 - \pi_{h_\omega}(a|s)}{h_\omega^a}$$

Substitute back to the original calculation,

$$\frac{\partial Q_\Omega(s,\omega)}{\partial h_\omega^a} = E_{s,\omega,a\sim\pi_{h_\omega}}[\frac{1 - \pi_{h_\omega}(a|s)}{h_\omega^a} Q_U(s,\omega,a)]$$

B.2 Derivative of Objective 2

Let h_ω^a be the element in the attention vector h_ω that corresponds to the action a, i.e. $h_\omega = [h_\omega^{a_0}, h_\omega^{a_1}, ...]$ Since we would like to minimize the cosine similarity, the negative cosine similarity will be used instead.

$$\sum_{h_{\omega'} \neq h_\omega} \frac{\partial C(h_\omega, h_{\omega'})}{\partial h_\omega^a} = \sum_{h_{\omega'} \neq h_\omega} \frac{\partial}{\partial h_\omega^a} \frac{< h_\omega, h_{\omega'} >}{||h_\omega|| \times ||h_{\omega'}||}$$

$$= \sum_{h_{\omega'} \neq h_\omega} \frac{||h_\omega|| \times ||h_{\omega'}|| \frac{\partial}{\partial h_\omega^a} < h_\omega, h_{\omega'} > - < h_\omega, h_{\omega'} > \frac{\partial}{\partial h_\omega^a} ||h_\omega|| \times ||h_{\omega'}||}{(||h_\omega|| \times ||h_{\omega'}||)^2}$$

I will calculate each of the derivative separately:

$$\frac{\partial}{\partial h_\omega^a} < h_\omega, h_{\omega'} > = \frac{\partial}{\partial h_\omega^a} \sum_{a'} h_\omega^{a'} h_{\omega'}^{a'} = h_{\omega'}^a$$

$$\frac{\partial}{\partial h_\omega^a} ||h_\omega|| \times ||h_{\omega'}|| = ||h_{\omega'}|| \frac{\partial}{\partial h_\omega^a} \sqrt{\sum_{a'} (h_\omega^{a'})^2} = ||h_{\omega'}|| \frac{h_\omega^a}{\sqrt{\sum_{a'} (h_\omega^{a'})^2}}$$

Substitute back to the original calculation,

$$\sum_{h_{\omega'} \neq h_\omega} \frac{\partial}{\partial h_\omega^a} \frac{< h_\omega, h_{\omega'} >}{||h_\omega|| \times ||h_{\omega'}||} = \sum_{h_{\omega'} \neq h_\omega} \frac{||h_\omega|| \times ||h_{\omega'}|| h_{\omega'}^a - < h_\omega, h_{\omega'} > ||h_{\omega'}|| \frac{h_\omega^a}{\sqrt{\sum_{a'} (h_\omega^{a'})^2}}}{(||h_\omega|| \times ||h_{\omega'}||)^2}$$

$$= \sum_{h_{\omega'} \neq h_\omega} \frac{h_{\omega'}^a}{||h_\omega|| \times ||h_{\omega'}||} - \frac{< h_\omega, h_{\omega'} > h_\omega^a}{(||h_\omega|| \times ||h_{\omega'}||) \times ||h_\omega||^2}$$

B.3 Derivative of Objective 3

Let h_ω^a be the element in the attention vector h_ω that corresponds to the action a, i.e. $h_\omega = [h_\omega^{a_0}, h_\omega^{a_1}, ...]$ Since entropy usually applies to probabilities, I will normalize the attention unit into $\overline{h_\omega} = \frac{h_\omega}{\sum_{a'} h_\omega^{a'}}$

$$\frac{\partial H(\overline{h_\omega})}{\partial h_\omega^a} = \frac{\partial \sum_{a'} \overline{h_\omega^{a'}} \log \overline{h_\omega^{a'}}}{\partial h_\omega^a}$$

$$= \frac{\partial \overline{h_\omega^a} \log \overline{h_\omega^a}}{\partial h_\omega^a} + \sum_{\overline{h_\omega^{a'}} \neq \overline{h_\omega^a}} \frac{\partial \overline{h_\omega^{a'}} \log \overline{h_\omega^{a'}}}{\partial h_\omega^{a'}}$$

$$= \frac{\partial \overline{h_\omega^a} \log \overline{h_\omega^a}}{\partial \overline{h_\omega^a}} \times \frac{\partial \overline{h_\omega^a}}{h_\omega^a} + \sum_{\overline{h_\omega^{a'}} \neq \overline{h_\omega^a}} \frac{\partial \overline{h_\omega^{a'}} \log \overline{h_\omega^{a'}}}{\partial \overline{h_\omega^{a'}}} \times \frac{\partial \overline{h_\omega^{a'}}}{h_\omega^{a'}}$$

I will calculate each of the derivative separately:

$$\frac{\partial \overline{h_\omega^a} \log \overline{h_\omega^a}}{\partial \overline{h_\omega^a}} = \log \overline{h_\omega^a} \frac{\partial \overline{h_\omega^a}}{\partial \overline{h_\omega^a}} + \overline{h_\omega^a} \frac{\partial \log \overline{h_\omega^a}}{\partial \overline{h_\omega^a}} = \log \overline{h_\omega^a} + 1$$

$$\frac{\partial \overline{h_\omega^a}}{\partial h_\omega^a} = \frac{\partial}{\partial h_\omega^a} \frac{h_\omega^a}{\sum_b h_\omega^b} = \frac{\sum_b h_\omega^b - h_\omega^a}{(\sum_b h_\omega^b)^2}$$

$$\frac{\partial \overline{h_\omega^{a'}}}{\partial h_\omega^a} = \frac{\partial}{\partial h_\omega^a} \frac{h_\omega^{a'}}{\sum_b h_\omega^b} = \frac{-h_\omega^{a'}}{(\sum_b h_\omega^b)^2}$$

Substitute back to the original calculation,

$$\frac{\partial \overline{h_\omega^a} \log \overline{h_\omega^a}}{\partial \overline{h_\omega^a}} = (\log \overline{h_\omega^a} + 1)\frac{\sum_b h_\omega^b - h_\omega^a}{(\sum_b h_\omega^b)^2} + \sum_{\overline{h_\omega^{a'}} \neq \overline{h_\omega^a}} (\log \overline{h_\omega^{a'}} + 1)\frac{-h_\omega^{a'}}{(\sum_b h_\omega^b)^2}$$

$$= \frac{(\log \overline{h_\omega^a} + 1)}{\sum_b h_\omega^b} + \sum_{h_\omega^{a'}} (\log \overline{h_\omega^{a'}} + 1)\frac{-h_\omega^{a'}}{(\sum_b h_\omega^b)^2}$$

B.4 Derivative of Objective 4

Let h_ω^a be the element in the attention vector h_ω that corresponds to the action a, i.e. $h_\omega = [h_\omega^{a_0}, h_\omega^{a_1}, ...]$ Since we would like to minimize the length, the negative length will be used instead.

$$\frac{\partial ||h_\omega||}{\partial h_\omega^a} = \frac{\partial}{\partial h_\omega^a}\sqrt{\sum_{a'} (h_\omega^{a'})^2}$$

$$= \frac{h_\omega^a}{\sqrt{\sum_{a'} (h_\omega^{a'})^2}} = \frac{h_\omega^a}{||h_\omega||}$$

C Experimental Details

D Code

The code below is based on codes in the ioc repository.[13]

fourrooms.py

```
import numpy as np
from random import uniform

class Fourrooms:
    def __init__(self, initstate_seed, punishEachStep, deterministic, modified, easier):
        self.punishEachStep = punishEachStep
        self.deterministic = deterministic
        self.modified = modified
        if easier:
            self.layout = """\
wwwwwwwwwwwww
w           w
w     w     w
w     w     w
w     w     w
w     w     w
w wwwww     w
w     wwwww w
w     w     w
w     w     w
w     w     w
w           w
wwwwwwwwwwwww
"""
        else:
            self.layout = """\
wwwwwwwwwwwww
w     w     w
w     w     w
w           w
w     w     w
```

```
w     w     w
ww wwww     w
w     www www
w     w     w
w     w     w
w           w
w     w     w
wwwwwwwwwwwww
"""


        self.occupancy = np.array([list(map(lambda c: 1 if c=='w' else 0, line)) for line in self.layout.splitlines()])

        self.action_space = 4

        self.observation_space = int(np.sum(self.occupancy == 0))

        # 0 - Up
        # 1 - Down
        # 2 - Left
        # 3 - Right

        self.directions = [np.array((-1,0)), np.array((1,0)), np.array((0,-1)), np.array((0,1))]

        self.rng = np.random.RandomState(1234)

        self.initstate_seed = initstate_seed
        self.rng_init_state = np.random.RandomState(self.initstate_seed)

        self.tostate = {}

        self.occ_dict = dict(zip(range(self.observation_space),
                                 np.argwhere(self.occupancy.flatten() == 0).squeeze()))

        statenum = 0
        for i in range(13):
            for j in range(13):
                if self.occupancy[i, j] == 0:
                    self.tostate[(i, j)] = statenum
                    statenum += 1

        self.tocell = {v:k for k,v in self.tostate.items()}

        self.goal = 62
        self.init_states = list(range(self.observation_space))
        self.init_states.remove(self.goal)


    def empty_around(self, cell):
        avail = []
        for action in range(self.action_space):
            nextcell = tuple(cell + np.multiply(self.directions[action], self.inQuad24(self.currentcell)))
            if not self.occupancy[nextcell]:
                avail.append(nextcell)
        return avail

    def reset(self, test=None):
        if test:
            state = test
        else:
            state = self.rng_init_state.choice(self.init_states)
        self.currentcell = self.tocell[state]
        return state

    def step(self, action):
```

```
        reward = -2 * int(self.punishEachStep)
        if self.rng.uniform() < 1/3 and not(self.deterministic):
            empty_cells = self.empty_around(self.currentcell)
            nextcell = empty_cells[self.rng.randint(len(empty_cells))]
        else:
            nextcell = tuple(self.currentcell + np.multiply(self.directions[action], self.
                inQuad24(self.currentcell)))


        if not self.occupancy[nextcell]:
            self.currentcell = nextcell

        state = self.tostate[self.currentcell]

        if state == self.goal:
            reward = 50

        done = state == self.goal
        return state, reward, float(done), None

    def inQuad24(self, cell):
        if not(self.modified):
            return np.array([1,1])
        if cell[1] > 6:
            if cell[0] < 7:
                return np.array([1, -1])
        else:
            if cell[0] > 6:
                return np.array([1, -1])
        return np.array([1, 1])
```

aoaoc_tabular.py

```
import numpy as np
from fourrooms import Fourrooms
from scipy.special import logsumexp, expit, softmax
'''
=======CLASS MAP=======
Option
    - FinalPolicy pi_h
        - Internal Policy (SoftmaxPolicy) pi_omega
        - Attention Unit (LearnableAttention/PredefinedAttention) h_omega
            - Value Objective (ValueObj) o1
            - Cosine Similarity Objective (CoSimObj) o2
            - Entropy Objective (EntropyObj) o3
            - Length Objective (LengthObj) o4
    - Termination Function (SigmoidTermination) beta_omega
    - Q_omega (Q_O)
Policy Over Options (POO) pi_Omega
    - Policy (EgreedyPolicy)
    - Q_Omega (Q_U)
'''
#=======Option=======
class Option:
    def __init__(self, rng, nfeatures, nactions, args, policy_over_options, index):
        self.weights = np.zeros((nfeatures, nactions))
        self.policy = FinalPolicy(rng, nfeatures, nactions, args, self.weights, index)
        self.termination = SigmoidTermination(rng, nfeatures, args)
        self.Qval = Q_U(nfeatures, nactions, args, self.weights, policy_over_options)

    def sample(self, phi):
        return self.policy.sample(phi)

    def terminate(self, phi, value=False):
        if value:
            return self.termination.pmf(phi)
```

```python
        else:
            return self.termination.sample(phi)

    def _Q_update(self, traject, reward, done, termination):
        self.Qval.update(traject, reward, done, termination)

    def _H_update(self, traject):
        qVal = self.Qval.value(traject[0][0], traject[2])
        self.policy.H_update(traject, qVal)

    def _B_update(self, phi, option, advantage):
        self.termination.update(phi, option, advantage)

    def _P_update(self, traject, baseline):
        self.policy.P_update(traject, baseline)

    def update(self, traject, reward, done, phi, option, termination, baseline, advantage):
        self._Q_update(traject, reward, done, termination)
        self._H_update(traject)
        self._P_update(traject, baseline)
        self._B_update(phi, option, advantage)


#=======Final Policy=======
class FinalPolicy:
    def __init__(self, rng, nfeatures, nactions, args, qWeight, index):
        self.rng = rng
        self.nactions = nactions
        self.internalPI = SoftmaxPolicy(rng, nfeatures, nactions, args, qWeight)
        if (args.h_learn):
            self.attention = LearnableAttention(nactions, args, index)
        else:
            self.attention = PredefinedAttention(args, index)

    def pmf(self, phi):
        pi = self.internalPI.pmf(phi)
        h = self.attention.pmf()
        normalizer = np.dot(pi, h)
        return (pi*h)/normalizer

    def sample(self, phi):
        return int(self.rng.choice(self.nactions, p=self.pmf(phi)))

    def H_update(self, traject, qVal):
        self.attention.update(traject, self.pmf(traject[0][0]), qVal)

    def P_update(self, traject, baseline):
        self.internalPI.update(traject, baseline)


#=======Internal Policy=======
class SoftmaxPolicy:
    def __init__(self, rng, nfeatures, nactions, args, qWeight):
        self.rng = rng
        self.nactions = nactions
        self.temp = args.temp
        self.weights = np.zeros((nfeatures, nactions))
        self.qWeight = qWeight
        self.lr = args.lr_intra

    def _value(self, phi, action=None):
        if action is None:
            return np.sum(self.weights[phi, :], axis=0)
        return np.sum(self.weights[phi, action], axis=0)

    def pmf(self, phi):
        v = self._value(phi)/self.temp
        return np.exp(v - logsumexp(v))
```

```python

    def sample(self, phi):
        return int(self.rng.choice(self.nactions, p=self.pmf(phi)))

    def update(self, traject, baseline):
        actions_pmf = self.pmf(traject[0][0])
        critic = self.qWeight[traject[0][0], traject[2]]
        if baseline:
            critic -= baseline
        self.weights[traject[0][0], :] -= self.lr*critic*actions_pmf
        self.weights[traject[0][0], traject[2]] += self.lr*critic


#=======Attention=======
class LearnableAttention():
    def __init__(self, nactions, args, index):
        self.weights = np.random.uniform(low=-1, high=1, size=(nactions,))
        self.lr = args.lr_attend
        self.o1 = ValueObj(args)
        self.o2 = CoSimObj(args, index)
        self.o3 = EntropyObj(args)
        self.o4 = LengthObj(args)
        CoSimObj.add2list(self)
        self.normalize = args.normalize

    def pmf(self):
        if self.normalize:
            return np.clip(softmax(self.weights), 0.05, None)
        return expit(self.weights)

    def _grad(self):
        attend = self.pmf()
        return attend*(1. - attend)

    def attention(self, a):
        return self.pmf()[a]

    def update(self, traject, finalPmf, qVal):
        hPmf = self.pmf()
        gradList = [self.o1.grad(traject[0][0], traject[2], hPmf, finalPmf, qVal), self.o2.
            grad(hPmf), self.o3.grad(hPmf), self.o4.grad(hPmf)]
        self.weights += self.lr * np.sum(gradList, axis=0) * self._grad()
        if self.normalize:
            self.normalizing()

    def normalizing(self):
        self.weights -= np.mean(self.weights)


class PredefinedAttention():
    def __init__(self, args, index):
        if (index==0):
            self.weights = np.array([1, 1, 1, 1])
        if (index==1):
            self.weights = np.array([1, 1, 1, 1])

    def pmf(self):
        return self.weights

    def attention(self, a):
        return self.pmf()[a]

    def update(self, traject, finalPmf, qVal):
        pass


#=======Objectives========
class Objective:
```

```python
    def __init__(self, weight):
        self.weight = weight

    def grad(self):
        return None

    def loss(self):
        return None


class ValueObj(Objective):
    def __init__(self, args):
        super().__init__(args.wo1)

    def grad(self, phi, a, hPmf, finalPmf, qVal):
        return self.weight * ((finalPmf + 1)/hPmf[a]) * qVal

    def loss(self):
        pass


class CoSimObj(Objective):
    hList = []

    def __init__(self, args, index):
        super().__init__(args.wo2)
        self.index = index

    def grad(self, hPmf):
        gradient = []
        for i in range(len(hPmf)):
            derivative = 0.
            exclude = 0
            for a in self.hList:
                if exclude == self.index:
                    continue
                exclude +=1

                normalizer = np.linalg.norm(hPmf)*np.linalg.norm(a.pmf())
                term1 = a.pmf()[i]/normalizer
                term2 = hPmf[i]*np.dot(hPmf,a.pmf()) / (normalizer*np.power(np.linalg.norm(
                    hPmf),2))
                derivative += -1*(term1 - term2)
            gradient.append(derivative)
        return self.weight * np.array(gradient)

    def loss(self):
        return np.sum([np.dot(hPmf,a.pmf())/(np.linalg.norm(hPmf)*np.linalg.norm(a.pmf()))
            for a in self.hList])

    @classmethod
    def add2list(cls, attention):
        cls.hList.append(attention)

    @classmethod
    def reset(cls):
        cls.hList = []


class EntropyObj(Objective):
    def __init__(self, args):
        super().__init__(args.wo3)

    def grad(self, hPmf):
        gradient = []
        normalizer = np.sum(hPmf)
        normh = hPmf/normalizer
        for i in range(len(hPmf)):
```

```
            term1 = (1.+np.log(normh[i]))/normalizer
            term2 = np.sum([(1.+np.log(normh[index]))*hPmf[index]/(normalizer**2) for index
                in range(len(hPmf))])
            gradient.append((term1-term2)*(self.loss(hPmf)-0.69))
        return self.weight * np.array(gradient)

    def loss(self, hPmf):
        normalizer = np.sum(hPmf)
        normh = hPmf/normalizer
        return -1*np.sum(normh * np.log(normh))


class LengthObj(Objective):
    def __init__(self, args):
        super().__init__(args.wo4)

    def grad(self, hPmf):
        return -1 * hPmf / self.loss(hPmf)


    def loss(self, hPmf):
        return np.linalg.norm(hPmf)


#=======Termination Function=======
class SigmoidTermination:
    def __init__(self, rng, nfeatures, args):
        self.rng = rng
        self.weights = np.zeros((nfeatures,))
        self.lr = args.lr_term
        self.dc = args.dc

    def pmf(self, phi):
        return expit(np.sum(self.weights[phi]))

    def sample(self, phi):
        return int(self.rng.uniform() < self.pmf(phi))

    def _grad(self, phi):
        terminate = self.pmf(phi)
        return terminate*(1. - terminate), phi

    def update(self, phi, option, advantage):
        magnitude, direction = self._grad(phi)
        self.weights[direction] -= self.lr*magnitude*(advantage+self.dc)


#=======Q-Value Individual Option=======
class Q_U:
    def __init__(self, nfeatures, nactions, args, weights, policy_over_options):
        self.weights = weights
        self.lr = args.lr_criticA
        self.discount = args.discount
        self.policy_over_options = policy_over_options

    def value(self, phi, action):
        return np.sum(self.weights[phi, action], axis=0)

    def update(self, traject, reward, done, termination):
        update_target = reward
        if not done:
            current_values = self.policy_over_options.value(traject[1][0])
            update_target += self.discount*((1. - termination)*current_values[traject[0][1]]
                 + termination*np.max(current_values))

        tderror = update_target - self.value(traject[0][0], traject[2])
        self.weights[traject[0][0], traject[2]] += self.lr*tderror

```

```python

#=======Policy Over Option=======
class POO:
    def __init__(self, rng, nfeatures, args):
        self.weights = np.zeros((nfeatures, args.noptions))
        self.policy = EgreedyPolicy(rng, args, self.weights)
        self.Q_Omega = Q_O(args, self.weights)

    def update(self, traject, reward, done, termination):
        self.Q_Omega.update(traject, reward, done, termination)

    def sample(self, phi):
        return self.policy.sample(phi)

    def advantage(self, phi, option=None):
        values = np.sum(self.weights[phi],axis=0)
        advantages = values - np.max(values)
        if option is None:
            return advantages
        return advantages[option]

    def value(self, phi, option=None):
        if option is None:
            return np.sum(self.weights[phi, :], axis=0)
        return np.sum(self.weights[phi, option], axis=0)


class EgreedyPolicy:
    def __init__(self, rng, args, weights):
        self.rng = rng
        self.epsilon = args.epsilon
        self.noptions = args.noptions
        self.weights = weights

    def _value(self, phi, action=None):
        if action is None:
            return np.sum(self.weights[phi, :], axis=0)
        return np.sum(self.weights[phi, action], axis=0)

    def sample(self, phi):
        if self.rng.uniform() < self.epsilon:
            return int(self.rng.randint(self.weights.shape[1]))
        return int(np.argmax(self._value(phi)))


#=======Q-Value All Option=======
class Q_O:
    def __init__(self, args, weights):
        self.weights = weights
        self.lr = args.lr_critic
        self.discount = args.discount

    def _value(self, phi, option=None):
        if option is None:
            return np.sum(self.weights[phi, :], axis=0)
        return np.sum(self.weights[phi, option], axis=0)

    def update(self, traject, reward, done, termination):
        update_target = reward
        if not done:
            current_values = self._value(traject[1][0])
            update_target += self.discount*((1. - termination)*current_values[traject[0][1]]
                 + termination*np.max(current_values))

        tderror = update_target - self._value(traject[0][0], traject[0][1])
        self.weights[traject[0][0],traject[0][1]] += self.lr*tderror


```

```python
#=======Standard=======
# Follow the code standard of the ioc repository
class Tabular:
    def __init__(self, nstates):
        self.nstates = nstates

    def __call__(self, state):
        return np.array([state,])

    def __len__(self):
        return self.nstates
```

visualize.py

```python
import numpy as np
import matplotlib.pyplot as plt
from fourrooms import Fourrooms
from time import sleep

# 0 - Red
# 1 - Green
# 2 - Blue
# 3 - Black
class Visualization:
    def __init__(self, fRoom, args, nactions, colorList=[[255,0,0],[0,255,0],[0,0,255],[0,0,0]]):
        assert args.noptions <= len(colorList), "Length of color list must match number of options"
        self.colorList = colorList
        self.layout = fRoom.layout
        self.occupancy = fRoom.occupancy
        self.tostate = fRoom.tostate
        self.tocell = fRoom.tocell
        self.screen = np.array([list(map(lambda c: [0,0,0] if c=='w' else [255,255,255], line)) for line in self.layout.splitlines()])
        self.lastphi = None
        self.noptions = args.noptions
        self.nactions = nactions

    def showMap(self, phi, option):
        color = self.colorList[option]
        self._draw(self.lastphi, [255,255,255])
        self._draw(phi, color)
        self.lastphi = phi
        plt.figure(figsize=(5,5))
        plt.subplot(111)
        plt.imshow(self.screen, vmax=255, vmin=0)
        plt.show()
        sleep(0.05)

    def showAttention(self, options):
        x = np.array([i for i in range(self.nactions)])
        plt.plot(x, np.array([int(i != 0) for i in range(self.nactions)]), color=[1,1,1])
        for i in range(self.noptions):
            plt.plot(x, options[i].policy.attention.pmf(), color=np.array(self.colorList[i])/255.)
        plt.show()

    def showPref(self, weight): # policy_over_options.weightsP or options[index].weightsP for weight
        pref = np.zeros((13,13,3), dtype="int")
        for i in range(13):
            for j in range(13):
                if self.occupancy[i,j] == 0:
                    choice = np.argmax(weight[self.tostate[(i,j)],:])
                    pref[i,j] = np.array(self.colorList[choice])
```

```
                else:
                    pref[i,j] = np.array([255,255,255])
        plt.figure(figsize=(5,5))
        plt.subplot(111)
        plt.imshow(pref, vmax=255, vmin=0)
        plt.show()

    def savePref(self, weight, algo=None, wo=None, dc=None, run=None):
        pref = np.zeros((13,13,3), dtype="int")
        for i in range(13):
            for j in range(13):
                if self.occupancy[i,j] == 0:
                    choice = np.argmax(weight[self.tostate[(i,j)],:])
                    pref[i,j] = np.array(self.colorList[choice])
                else:
                    pref[i,j] = np.array([255,255,255])
        plt.figure(figsize=(5,5))
        plt.subplot(111)
        plt.imshow(pref, vmax=255, vmin=0)
        if wo != None:
            plt.savefig("../result/{0}/wo_{1}/dc_{2}/run_{3}.png".format(algo, str(wo), str(
                dc), str(run)))
        else:
            plt.savefig("../result/{0}/dc_{1}/run_{2}.png".format(algo, str(dc), str(run)))

    def resetMap(self, phi):
        self.screen = np.array([list(map(lambda c: [0,0,0] if c=='w' else [255,255,255],
            line)) for line in self.layout.splitlines()])
        self.lastphi = phi
        self._draw([62],[200,200,200])

    def _draw(self, phi, rgb):
        self.screen[self.tocell[phi[0]]] = np.array(rgb)
```