DCRAC: Deep Conditioned Recurrent Actor-Critic for Multi-Objective Partially Observable Environments

Xiaodong Nian
University of Washington
Tacoma, Washington
xdnian@uw.edu

Athirai A. Irissappane
University of Washington
Tacoma, Washington
athirai@uw.edu

Diederik Roijers
HU University of Applied Sciences
Utrecht, Netherlands
diederik.yamamoto-roijers@hu.nl

ABSTRACT

In many decision-making problems, agents aim to balance multiple, possibly conflicting objectives. Existing research in deep reinforcement learning mainly focuses on fully-observable single-objective solutions. In this paper, we propose DCRAC, a deep reinforcement learning framework for solving partially-objective multi-objective problems. DCRAC follows a conditioned actor-critic approach in learning the optimal policy, where the network is conditioned on the weights, i.e, relative importance for the different objectives. To deal with longer action-observation histories, in the case of partially observable environments, we introduce DCRAC-M which uses memory networks to further enhance the reasoning ability of the agent. Experimental evaluation on benchmark problems show the superiority of our approach when compared to state-of-the-art.

CCS CONCEPTS

• **Computing methodologies** → **Reinforcement learning**;

KEYWORDS

Deep Reinforcement Learning; Multi-Objective POMDP; Actor-Critic;

ACM Reference Format:
Xiaodong Nian, Athirai A. Irissappane, and Diederik Roijers. 2020. DCRAC: Deep Conditioned Recurrent Actor-Critic for Multi-Objective Partially Observable Environments. In *Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), Auckland, New Zealand, May 9–13, 2020,* IFAAMAS, 8 pages.

1 INTRODUCTION

Combining advances in Reinforcement Learning (RL) and deep learning, deep reinforcement learning techniques [15] have helped solve a number of complex, high-dimensional decision-making problems [16]. Deep Q-Networks (DQNs) have shown to be capable of learning human-level control policies [9]. This breakthrough paved way for general-purpose agents that can learn a diverse range of tasks which are challenging for humans.

DQN and its recent extensions [7, 18, 19] generalize Q-learning to high-dimensional environments by using a neural network to estimate the Q-function.

However, the focus in these recent advances has been on designing deep RL algorithms for decision problems that can be modeled as single-objective fully observable environments, i.e., Markov decision processes (MDPs). While in most cases, the state is not fully observable, for example, an agent in a maze environment may have a limited view and can only observe the surrounding in front of itself. Thus, the agent may need to speculate the current state based on its memory of past observations and actions. This kind of problems can be modeled as partially observable MDPs (POMDPs). There are few research works on deep RL methods for partial observability. DRQN [4] and ADQRN [21] extend DQN by using recurrent neural networks for memorizing action-observation histories to solve POMDPs. The objective of these POMDP algorithms is to maximize the single scalar rewards across time. Given a single-objective scalar reward r_t, at time-step t, the agent has to find a policy π^* which maximizes the expected total reward $max_\pi E_\pi[\Sigma_t r_t]$.

Though single-objective scalar rewards are intuitive, many real-world problems have multiple and possibly conflicting objectives. For example, a strategy for manufacturing should focus on maximizing quality and efficiency while minimizing operation cost; an air traffic control system should ensure efficiency and flight safety simultaneously. This leads to a vector of rewards, with a reward for every objective at every time-step. Often, it is hard, or even unfeasible to do a priori scalarization of these vector-valued rewards to a single scalar mainly because: 1) the scalarization function can be unknown and complex; 2) the importance (weights) of the different objectives can be unknown and change dynamically. In this paper, we focus on this *dynamic weights* setting, where the priorities between the objectives change over time. This happens, for example when objectives represent the goods traded in an open market. The prices of the goods can fluctuate with time. For the dynamic weights setting, [1] proposed the use of conditioned network by extending DQN for multi-objective fully observable environments.

In this paper, we propose the Deep Conditioned Recurrent Actor Critic (DCRAC) method for handling environments that are both multi-objective and partially observable. DCRAC uses the advantage actor-critic method for learning the optimal policy and uses recurrent layers in the neural network architecture for remembering the action-observation histories. We further propose an extension, DCRAC-M, by replacing the recurrent layers in DCRAC with a memory network for learning longer action and observation histories. Diverse Experience Replay [1] is used for efficiently learning the optimal policy. The main contributions of the paper are listed below: 1) We propose a deep learning framework DCRAC to handle partially observable multi-objective problems; 2) We use memory networks (DCRAC-M) for learning long term dependencies in action-observation histories and empirically show their importance for partially observable environments; 3) We use diverse experience replay for effectively sampling transitions from the replay buffer tailored to partially observable environments; 4) We conduct experiments using the partially-observable multi-objective

Proc. of the 19th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2020), B. An, N. Yorke-Smith, A. El Fallah Seghrouchni, G. Sukthankar (eds.), May 9–13, 2020, Auckland, New Zealand.

version of Deep Sea Treasure [10]. Evaluation results comparing the cummulative average episodic discounted rewards and cummulative average episodic regret show that our approach performs better than condition networks [1].

2 BACKGROUND AND RELATED WORK

Partially Observable Markov Decision Process (POMDP). When complete information of the environment is not available, a RL problem is modeled as a Partially Observable Markov Decision Process (POMDP) [6]. POMDP is described by the tuple $\langle S, A, T, R, \Omega, O \rangle$, with S, the state space; A, the action space; T, the transition function; R, the reward function; Ω, a finite set of observations and O, the observation model. At every time step t, the environment has a state $s_t \in S$. The agent takes an action $a_t \in A$, which causes a state transition from s_t to a new state s_{t+1} using the transition function $T(s_{t+1}|s_t, a_t)$. The agent also receives observations based on the observation function $O(o_t|a_t, s_{t+1})$. For a transition, the agent receives a reward $R(s_t, a_t, s_{t+1})$. Since state s_t is not fully observable, the agent needs to remember the entire history in order to act optimally. Let $h_t = (a_0, o_1, a_1, o_2, \cdots, a_{t-1}, o_t)$ be the history of actions and observations until time step t. The goal of the agent is to learn a policy $\pi(h_t)$ which maps the history to an action such that it maximizes the expected discounted reward [5]. Assuming that π can sufficiently maintain a probability distribution over trajectories, we sample τ, a trajectory conditioned on h_t to obtain the remaining history. The expected discounted reward, i.e., value function $V_\pi(h_t)$ is then calculated as,

$$V_\pi(h_t) = E_{\tau|h_t}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right] \tag{1}$$

Here, $\gamma \in [0, 1]$ is the discount factor and r_t is the reward obtained at time step t based on h_t. $V_\pi(h_t)$ can be reconstructed from the Q-function $Q_\pi(h_t, a_t)$, which is the expected total reward of executing policy π, starting with h_t and taking action $a_t \in A$. The optimal policy can be computed from the optimal Q-function Q^*, where the agent executes, at every time-step, the action whose Q-value for the current history h_t is maximal, i.e., $\pi^* = argmax_{a_t \in A} Q^*_\pi(h_t, a_t)$.

Deep Networks for Partially Observable Environments. Deep Recurrent Q-Network (DRQN) [4] extends DQN for partially observable environments by using a recurrence layer to remember observation histories while approximating the Q-function. However, DRQN did not consider action histories. Action-based Deep Recurrent Q-Network (ADRQN)[21] includes both action and observation histories using an LSTM layer, demonstrating that it is possible to store only the sufficient statistics of the action observation histories using a recurrent layer instead of the entire preceding history [20]. The network takes as input a sequence of action, observation pairs and computes the Q-values for each action. ADRQN uses experience replay by adding the transition $(\{a_{t-1}, o_t\}, a_t, r_t, o_{t+1})$ to a replay buffer D at every time-step t. The network is trained by uniformly sampling mini-batches of experiences $U(D)$ using the loss function $L(\theta)$ given below,

$$L(\theta) = E_{(\{a_{t-1}, o_t\}, a_t, r_t, o_{t+1}) \sim U(D)}\left[y_t - Q(h_{t-1}, a_{t-1}, o_t, a_t|\theta)\right]$$
$$y_t = r_t + \gamma max_{a'} Q(h_t, a_t, o_{t+1}, a'|\theta^-)$$

Here, θ represents the parameters of the Q-network. The Q-network computes $Q(h_{t-1}, a_{t-1}, o_t, a_t|\theta_t)$, where a_{t-1}, o_t are inputs to the hidden layer of the unrolled LSTM which stores the sufficient statistics. y_t is computed from the target network with parameters θ^- which is cloned from θ, periodically.

Recurrent Deterministic Policy Gradient (RDPG) [5] uses an actor-critic technique for handling partial observability. Actor-critic algorithms are better in handling larger state and action spaces. Here, the actor and critic are neural networks. The actor is updated using policy gradient in the direction suggested by the critic (which determines the value function). Though the actor-critic method of our approach DCRAC is inspired from RDPG, DCRAC additionally handles multi-objectivity, uses memory networks and incorporates diverse experience replay for improving sample efficiency.

Deep Networks for Multi-Objective Environments. For a given n-objective decision problem, the reward $\boldsymbol{r}_t$ is vector valued, i.e., one reward per objective. Thus, for a policy π, the multi-objective value function $\boldsymbol{V}_\pi$ is also vector valued. Using a scalarization function f, we can map the multi-objective value $\boldsymbol{V}_\pi$ to a scalar when the importance of the objectives, weight $\boldsymbol{w} \in \mathbb{R}^n$ is known. When f is linear, the scalarization function becomes,

$$f(\boldsymbol{V}_\pi, \boldsymbol{w}) = \boldsymbol{w} \cdot \boldsymbol{V}_\pi \tag{2}$$

However, in most cases $\boldsymbol{w}$ is not known in advance. An optimal solution for the multi-objective MDP under a linear f is a convex coverage set (CCS), i.e., a set of undominated policies containing at least one optimal policy for any linear scalarization [14].

Scalarized deep Q-learning [10] extends DQN for a multi-objective fully observable setting by learning vector-valued Q-functions for a given $\boldsymbol{w}$. By sequentially training Q-networks until convergence on corner weights, they approximate the CCS with a set of Q-networks. However, this method is hard to generalize across all weights as the weights can vary over time and it can be time-consuming to learn the entire CCS.[1] introduced Conditioned Network for the dynamic weights setting. Instead of training a set of Q-networks, a single Q-network, conditioned on $\boldsymbol{w}$ is used. For a given transition (s_t, a_t, r_t, s_{t+1}), the loss $L(\theta)$, is computed as the sum of the loss on the active weight vector $\boldsymbol{w}_i$ and $\boldsymbol{w}_j$, which is randomly sampled from the set of previously encountered weight vectors.

$$L(\theta) = \frac{1}{2}|\boldsymbol{y}_t^{\boldsymbol{w}_i} - \boldsymbol{Q}_{CN}(s_t, a_t|\boldsymbol{w}_i, \theta)| + |\boldsymbol{y}_t^{\boldsymbol{w}_j} - \boldsymbol{Q}_{CN}(s_t, a_t|\boldsymbol{w}_j, \theta)|$$
$$\boldsymbol{y}_t^{\boldsymbol{w}} = \boldsymbol{r}_t + \gamma \boldsymbol{Q}_{CN}(s_{t+1}, argmax_{a'}[\boldsymbol{Q}_{CN}(s_{t+1}, a'|\boldsymbol{w}, \theta).\boldsymbol{w}]|\boldsymbol{w}, \theta^-)$$

Here, $\boldsymbol{Q}_{CN}(a_t, s_t|\boldsymbol{w}, \theta)$ is the network's Q-value-vector for action a_t in state s_t using the weight vector $\boldsymbol{w}$. θ represents the network parameters. $\boldsymbol{y}_t^{\boldsymbol{w}}$ is computed from the target network, which is also a conditioned network with parameters θ^-. Diverse Experience Replay (DER), a diverse buffer from which relevant experiences can be sampled for weight vectors whose policies have not been executed recently is used for improving sample efficiency. However, [1] can be applied only to fully-observable scenario. [14] introduced OLSAR for partially observable multi-objective environments, which finds the CCS by solving a series of scalarized POMDP problems by selecting $\boldsymbol{w}$. The OLSAR-complaint Perseus solver was used to solve the POMDPs. However, these POMDP

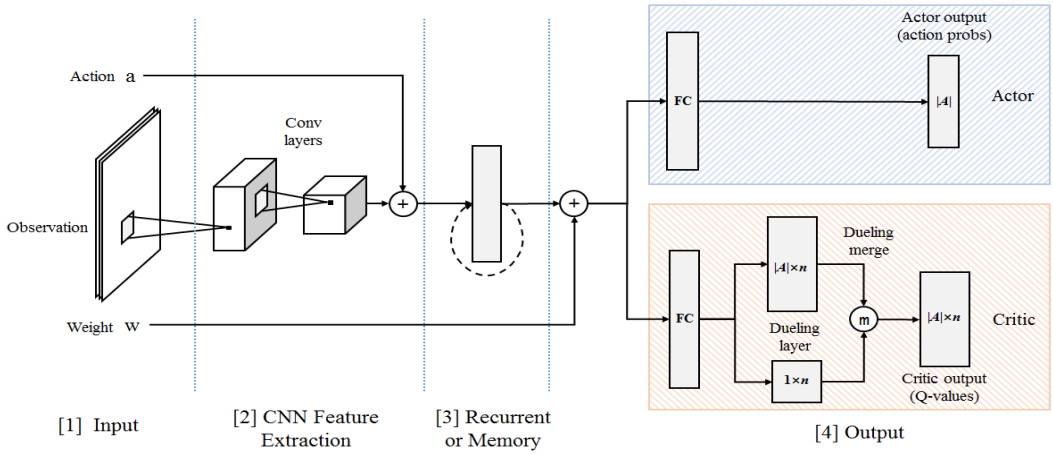

Figure 1: Architecture of DCRAC - Deep Conditioned Recurrent Actor-Critic

solutions are often intractable for large state, action spaces as finding the optimal policy is PSPACE complete [12]. In this paper, we will use deep networks as Q-function approximators for solving Multi-Objective POMDPs (MOPOMDPs) as they can scale to high dimensional spaces.

3 PROPOSED APPROACH

We propose DCRAC, a Deep Conditioned Recurrent Actor Critic model for decision making in partially-observable multi-objective environments. DCRAC uses the actor-critic technique for determining the optimal policy. Both the actor and critic are implemented as neural networks. The actor specifies the policy, i.e., the mapping from histories to actions. Since we deal with a partially observable environment, the history h_t represents a sequence of k action-observation pairs $\{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}$, until the current time-step t. Ideally, the entire sequence of action-observation pairs from 0 to t should be used, however due to memory constraints, at a given time, we train the network with k sequenctial action-observation pairs. The critic computes the value function based on which the actor is updated using policy gradient [8]. Both the actor and critic networks are conditioned on the weight vector $\boldsymbol{w}$, to generalize across different weights better. This helps to deal with the dynamic weights setting. The actor and critic networks use recurrent layers for handling sequences of action-observation pairs. We also propose to use memory networks (DCRAC-M) in Sec. 3.3 to account for long term dependencies among the action and observation histories. We follow an off-policy setting, where the agent stores its interactions in a replay buffer from which transitions are sampled to train the DCRAC network. We use Diverse Experience Replay (DER) [1] to improve sample efficiency and prevent replay buffer bias. In our work, the replay buffer stores transitions of the form $(\{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}, a_t, \mathbf{r}_t, o_{t+1})$, consisting of action-observation histories (see Sec. 3.4 for details).

3.1 Actor-Critic Network

Both the actor and critic networks take as input a transition of the form $(\{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}, a_t, \mathbf{r}_t, o_{t+1})$, containing a sequence of k action-observation pairs, along with the weight vector $\boldsymbol{w}$ as shown in Fig. 1. Here, $\mathbf{r}_t$ is vector valued, i.e., one reward per objective[1]. Instead of having two different networks with different parameters for processing the same input, the actor and critic networks share the weights for the layers upto Step. 3, as shown in Fig. 1, representing the DCRAC architecture. A LSTM layer is used to process the history of actions $\{a_{t-k-1} \cdots a_t\}$ and observations $\{o_{t-k} \cdots o_{t+1}\}$, as shown in Fig. 1[Step. 3]. However, since each observation (for example o_t) in our case, is an image, it is first passed though convolution layers, as shown in Fig. 1[Step. 2]. The extracted features are then concatenated with the corresponding action (a_{t-1}), which are encoded as one-hot vectors.

The concatenated vector then serves as input to the LSTM layer. Fig. 2 shows the unrolled LSTM for 2 time-steps $t-1, t$. At time step t, the LSTM takes as input the concatenated features obtained from o_t and a_{t-1}, as well as the hidden state $\hat{h}_{t-1}$, which summarizes the action and observation histories till time $t-1$. Depending on memory availability, the LSTM can be unrolled for k time-steps.

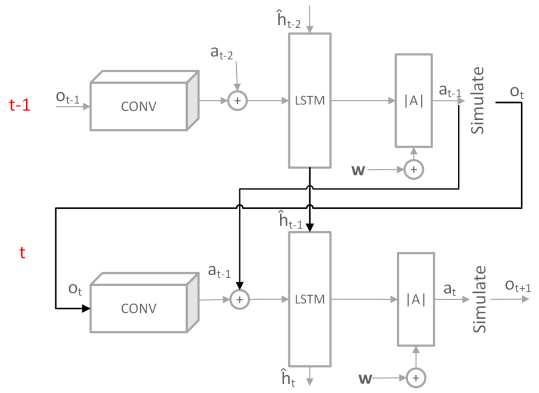

Figure 2: Unrolled LSTM Layer

The output of the LSTM layer is concatenated with the weight vector $\boldsymbol{w}$, as shown in Fig. 1. The weight vector $\boldsymbol{w}$ is introduced after the LSTM layer in order to minimize interference while processing

[1]The vector-valued variables are mentioned in bold throughout the paper.

action-observation histories. The actor $\pi(h_t, \boldsymbol{w}|\theta)$ with parameters θ is a policy network which maps history h_t to an action $\in A$. The concatenated result of the LSTM output and weight vector $\boldsymbol{w}$ is then passed through a fully connected layer to determine the probability of selecting each action. The actor network is updated using policy gradient $\nabla_\theta J_\pi$,

$$\nabla_\theta J_\pi = \frac{1}{2}\left[\nabla_\theta \pi(h_t, \boldsymbol{w}_e|\theta) \cdot \mathcal{A}^{\boldsymbol{w}_e} + \nabla_\theta \pi(h_t, \boldsymbol{w}_r|\theta) \cdot \mathcal{A}^{\boldsymbol{w}_r}\right] \quad (3)$$

$$\mathcal{A}^{\boldsymbol{w}} = \left[\mathbf{y}_t^{\boldsymbol{w}} - E_{a\in A}\mathbf{Q}(h_t, a, \boldsymbol{w}|\omega)\right] \cdot \boldsymbol{w} \quad (4)$$

The gradient is calculated based two different weight vectors, the current weight $\boldsymbol{w}_e$ as well as a random previously encountered weight $\boldsymbol{w}_r$. Training the actor on two different weights at the same time avoids the problem of overfitting [1]. $\mathcal{A}^{\boldsymbol{w}}$ is the advantage-based objective function, $\mathbf{y}_t^{\boldsymbol{w}}$ is the target value and $\mathbf{Q}(h_t, a, \boldsymbol{w}|\omega)$ is the Q-value computed using the critic network as described below.

The critic $\mathbf{Q}(h_t, a_t, \boldsymbol{w}|\omega)$ is a Q-network (with parameters ω) for estimating the Q-values for the actions $a_t \in A$, given the history h_t and the weight vector $\boldsymbol{w}$. The critic has a different structure than the actor as shown in Fig. 1[Step. 4]. A dueling architecture [19] is followed for the critic, where the first fully connected layer is connected to two separate heads, one to estimate the state value $\mathbf{V}(h_t, \boldsymbol{w}|\omega, \beta)$ and the other to estimate the action advantage $\mathbf{A}(h_t, a_t, \boldsymbol{w}|\omega, \alpha)$, respectively. Here, ω denotes the parameters until the first fully connected layer, while α and β are the parameters of the two separate dueling heads. The advantage function $\mathbf{A}(h_t, a_t, \boldsymbol{w}|\omega, \alpha)$ gives the relative measure of importance of each action. $\mathbf{Q}(h_t, a_t, \boldsymbol{w}|\omega)$ is then be obtained using Eqn. 5.

$$\begin{aligned}\mathbf{Q}(h_t, a_t, \boldsymbol{w}|\omega, \alpha, \beta)) &= \mathbf{V}(h_t, \boldsymbol{w}|\omega, \beta) \\ &+ [\mathbf{A}(h_t, a_t, \boldsymbol{w}|\omega, \alpha) - E_{a\in A}\mathbf{A}(h_t, a, \boldsymbol{w}|\omega, \alpha)]\end{aligned} \quad (5)$$

The critic network is updated by minimizing the loss L_ω based on TD error, given by Eqn. 6, where $\boldsymbol{w}_e$ is the current sampled weight, $\boldsymbol{w}_r$ is randomly chosen previous weight.

$$L_\omega = \frac{1}{2}\left(\left|\mathbf{y}_t^{\boldsymbol{w}_e} - \mathbf{Q}(h_t, a_t, \boldsymbol{w}_e|\omega)\right| + \left|\mathbf{y}_t^{\boldsymbol{w}_r} - \mathbf{Q}(h_t, a_t, \boldsymbol{w}_r|\omega)\right|\right) \quad (6)$$

To improve stability and convergence, we use the target actor π' and critic $\mathbf{Q}'$ networks with parameters θ' and ω', respectively. The parameters θ' and ω' of the target networks are updated using soft updates [8]. The target value $\mathbf{y}_t^{\boldsymbol{w}}$ is computed as,

$$\mathbf{y}_t^{\boldsymbol{w}} = \boldsymbol{r}_t + \gamma \mathbf{Q}'(h_{t+1}, \pi'(h_{t+1}, \boldsymbol{w}|\theta'), \boldsymbol{w}|\omega') \quad (7)$$

3.2 Algorithm

Algorithm. 1 describes DCRAC in detail. In lines 1-4, we initialize: the actor π and critic $\mathbf{Q}$ networks; the target networks π', $\mathbf{Q}'$; the replay buffer D; the unique weight history W; and k, the number of time-steps for which the LSTM can be unrolled. At beginning of each training episode e, the agent receives a weight vector w_e. If w_e is new and not previously encountered, it is added to W (Lines 6-7). The complete history of action observation pairs H for episode e, is initialized to an empty set. $h_{t=0}$ which represents the previous k action-observation pairs is also initialized to an empty set (Line 8). Based on h_0, the initial observation o_0 is obtained (Line 9).

We use ϵ-greedy exploration which is annealed over time (Lines 11, 29), increasing the probabilty of choosing the action suggested by the actor $a_t = \pi(h_t, \boldsymbol{w}_e|\theta)$ (Line 12). a_t is executed to receive reward $\mathbf{r}_t$ and observation o_{t+1} (Line 13). H is updated with the new action-observation pair (Line 14). For every timestep, the transition $(e, \{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}, a_t, \mathbf{r}_t, o_{t+1})$, which includes the episode number e, previous k-1 action-observation pairs, the current action a_t, the current reward r_t and the observation obtained o_{t+1} is added to the replay buffer D (Line 15). Then, a mini-batch of N transitions is randomly sampled using diverse experience replay (Line 16) to update the actor and critic networks (See Sec. 3.4 for more details). The mini-batch can contain transitions from different episodes. Such random sampling reduces correlation among samples and is found to improve computational efficiency [4].

We train the networks using two different weight vectors, the active weight $\mathbf{w}_e$ and $\mathbf{w}_r$, which is randomly sampled from a uniform distribution of previously observed weights $U(\mathrm{W})$. Doing so, prevents over-fitting the recent weights. The target value $\mathbf{y}_t^{\boldsymbol{w}}$ is computed by Eqn. 7 (Lines 19), and used to update critic (Line 21) by minimize Eqn.6. The actor is updated using policy gradient based on advantage given by Eqn. 3 & 4 (Line 22-24). A soft update [5] of the parameters of the target network is performed in each round (Lines 25 - 27). The whole process is repeated until termination (Line 10) and for E episodes (Line 5).

3.3 DCRAC with Memory Networks

We also propose DCRAC-M, in which DCRAC uses memory networks (instead of LSTM) for learning long term dependencies among the action-observation histories, especially when the history is lengthy [17]. The memory of recurrent layers is small and they encode the entire history as dense vectors $\hat{h}_t$, thereby, failing to explicitly focus attention on different parts of the history. Memory networks use an external memory space (other than the neural network) to store action and observation pairs for each timestep independently rather than summarizing the entire history as a

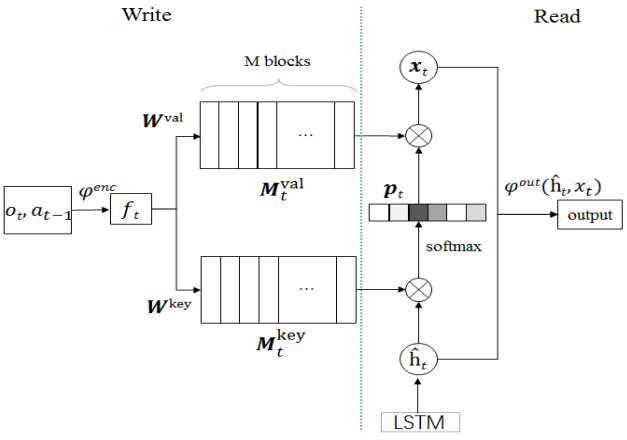

Figure 3: Memory Network

Algorithm 1 DCRAC: Deep Conditioned Recurrent Actor-Critic

Randomly initialize actor network $\pi(h_t, \boldsymbol{w}|\theta)$, critic network $\mathbf{Q}(h_t, a_t, \boldsymbol{w}|\omega)$ with weights θ, ω, respectively.
Initialize target networks π', $\mathbf{Q}'$ with weights $\theta' \leftarrow \theta$, $\omega' \leftarrow \omega$
Initialize (diverse) replay buffer D and unique weight history W
Initialize k, the number of time-steps of the unrolled LSTM
for episode e = 1 to E **do**
  $\boldsymbol{w}_e$ = getWeightVector(e)
  add $\boldsymbol{w}_e$ to W
  Initialize full history $H = \{\}$, current empty history $h_0 = \{\}$
  Receive initial observation o_0
  while $o_t \neq$ terminal **do**
    With probability ϵ select a random action a_t
    Otherwise select action $a_t = \pi(h_t, \boldsymbol{w}_e|\theta)$, where $h_t = \{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}$ contains sequence of k action-observation pairs.
    Execute action a_t and obtain reward $\boldsymbol{r}_t$ and new observation o_{t+1}
    Update full history $H \leftarrow H \cup \{a_t, o_{t+1}\}$
    Store transaction $(e, \{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}, a_t, \mathbf{r}_t, o_{t+1})$ in replay buffer D
    Sample a mini-batch of N transactions from D using diverse experience replay (see Sec. 3.4)
    Randomly sample $\boldsymbol{w}_r$ from $U(\mathrm{W})$
    Compute target values for $\boldsymbol{w}_e$ and $\boldsymbol{w}_r$:
      Target value $y_t^{\boldsymbol{w}} = \boldsymbol{r}_t + \gamma \mathbf{Q}'(h_{t+1}, \pi'(h_{t+1}, \boldsymbol{w}|\theta'), \boldsymbol{w}|\omega')$
    Update the critic network by minimizing the loss:
      $L_\omega = \frac{1}{2}\left(\left|\mathbf{y}_t^{\boldsymbol{w}_e} - \mathbf{Q}(h_t, a_t, \boldsymbol{w}_e|\omega)\right| + \left|\mathbf{y}_t^{\boldsymbol{w}_r} - \mathbf{Q}(h_t, a_t, \boldsymbol{w}_r|\omega)\right|\right)$
    Update the actor network using policy gradient:
      $\nabla_\theta J_\pi = \frac{1}{2}\left[\nabla_\theta \pi(h_t, \boldsymbol{w}_e|\theta) \cdot \mathcal{A}^{\boldsymbol{w}_e} + \nabla_\theta \pi(h_t, \boldsymbol{w}_r|\theta) \cdot \mathcal{A}^{\boldsymbol{w}_r}\right]$
      where $\mathcal{A}^{\boldsymbol{w}} = \left[\mathbf{y}_t^{\boldsymbol{w}} - E_{a \in A}\mathbf{Q}(h_t, a, \boldsymbol{w}|\omega)\right] \cdot \boldsymbol{w}$
    Update the target networks with learning rate λ:
      $\omega' = \lambda\omega + (1-\lambda)\omega'$
      $\theta' = \lambda\theta + (1-\lambda)\theta'$
  end while
  Anneal(ϵ)
end for

dense vector. Such a design helps to store accurate histories and can easily focus attention to specific portions of the stored memory [13]. We replace the recurrent layer (Fig. 1[Step. 3]) of DCRAC with an adapted version of Feedback Recurrent Memory Q-Network [11].

The memory network implementation is shown in Fig 3. The left-side shows how information is written to the memory. The concatenated output of action a_{t-1} and feature vectors from the convolutional layer for o_t (output of Step. 2 in Fig. 1) is considered to be the embedded feature $f_t \in \mathbb{R}^f$ that is saved in the memory at time-step t. This operation is given by φ^{enc} as shown below,

$$f_t = \varphi^{enc}(o_t, a_{t-1}) \tag{8}$$

Two memory blocks, key block $\mathbf{M}_t^{key}$, value block $\mathbf{M}_t^{val} \in \mathbb{R}^{m \times M}$, with m dimensional embeddings are used. The length of each memory block is M. Though every embedded feature f_t is added to the memory separately, we can consolidate that memory network linearly transforms the embedded feature of last M action-observation pairs as key and value memory blocks using Eqn. 9, 10.

$$\mathbf{M}_t^{key} = \boldsymbol{U}^{key}\mathbf{F}_t \tag{9}$$

$$\mathbf{M}_t^{val} = \boldsymbol{U}^{val}\mathbf{F}_t \tag{10}$$

where, $\boldsymbol{U}^{key}, \boldsymbol{U}^{val} \in \mathbb{R}^{m \times f}$ are parameters of the linear transformations for the key block and value block, respectively. Here, $\mathbf{F}_t = [f_{t-1}, f_{t-2}, \ldots, f_{t-M}] \in \mathbb{R}^{f \times M}$ is the concatenation of embedded features from the last M action-observation pairs.

The right side of Fig. 3 shows how to read information from the memory using soft attention [2]. To retrieve the memory, the attention weights $p_t(i)$ for the i^{th} memory block at time-step t is calculated. Specifically, we compute the softmax attention σ, given the hidden state $\hat{h}_t$ and key memory $\mathbf{M}_t^{key}$ using Eqn. 11. Then value x_t is retrieved from the memory using Eqn. 12.

$$p_t(i) = \sigma(\hat{h}_t^\top \mathbf{M}_t^{key}[i]) \tag{11}$$

$$x_t = \mathbf{M}_t^{val}\boldsymbol{p}_t \tag{12}$$

$\hat{h}_t$ is obtained from an LSTM layer which takes as input the concatenated vector $[f_t, x_{t-1}]$ and the previous hidden state $\hat{h}_{t-1}$.

$$\hat{h}_t = LSTM([f_t, x_{t-1}], \hat{h}_{t-1}) \tag{13}$$

The final output of the memory network is computed using the retrieved memory x_t and the hidden state $\hat{h}_t$, using φ^{out} with

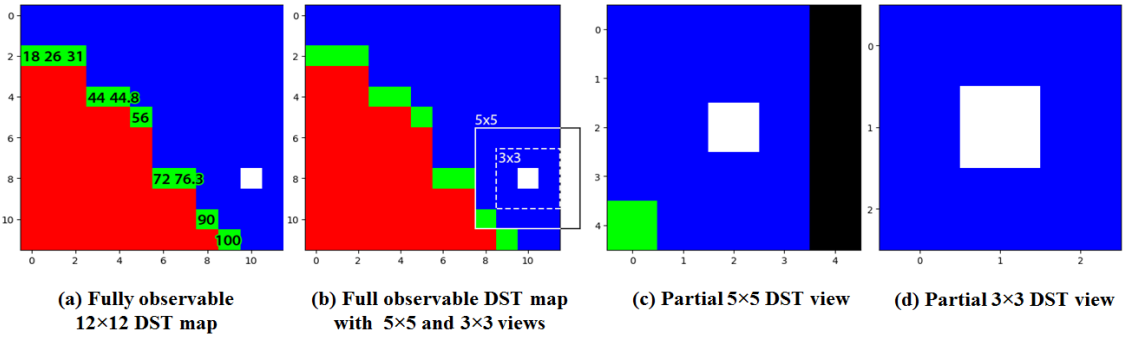

Figure 4: Evaluation Environments: Partially observable Deep Sea Treasure with different size field of view

parameter u as shown below.

$$\varphi^{out}(\hat{h}_t, x_t) = ReLU(u\hat{h}_t + x_t) \tag{14}$$

3.4 Experience Replay

We follow an off-policy setting where the transitions are stored in a replay buffer and are randomly sampled to train the network. Transitions are of the form $(e, \{a_{t-k-1}, o_{t-k}, \cdots, a_{t-1}, o_t\}, a_t, \mathbf{r}_t, o_{t+1})$, including the episode number e, previous $k-1$ action-observation pairs, the current action a_t, the current reward r_t and the observation obtained o_{t+1}. In our paper, we use $k = 10$, based on the memory availability. The transition always contains action-observation histories for previous k time-steps. During bootstrapping when previous k time-steps do not exist, they are filled with zero vectors which are masked during neural network training.

Additionally, since different weights should be trained at the same time to avoid the models from being over-fitted using the same weight spaces or the early trained weights being gradually ignored in the training progress, we use Diverse Experience Replay (DER) [1]. Standard experience replay keeps a replay buffer that drops a transition in the first-in-first-out order when the buffer becomes full. However, in the multi-objective environment with dynamic weights, it is important to always keep a diverse replay buffer with transitions relevant to all possible objective preference weights' optimal policies. This ensures that the agent learns appropriately without ignoring other possible policies.

DER separates the existing replay buffer D into two parts: a standard first-in-first-out replay buffer D_s and a diverse replay buffer D_d. The transitions are added and removed to the standard buffer D_s in a first-in-first-out manner. To add a new transition to the diverse replay buffer D_d, the diversity between the new transition and the existing transitions in D_d is computed and the new transition is included only if it improves the diversity of the buffer D_d. When D_d is full, the least diverse transition is dropped. Diversity among transitions is computed using the crowing distance [3] of their discounted cumulative rewards. Half of the existing experience replay buffer is used as diverse replay buffer, while the sampling is made across the whole buffer D without any distinction. During training, mini-batches of transitions are randomly sampled from D. These transitions can belong to different episodes, thereby, avoiding correlation among the samples. We can also sample a list of contiguous transitions belonging to the same episode. But, random sampling is shown to achieve similar performance as sequential transitions with much lower training cost [4, 21].

4 EXPERIMENTS

4.1 Setup

We perform experiments on Deep Sea Treasure (DST) [10], a multi-objective environment. DST is originally fully observable. The states are represented as images in DST. We introduce partial observability by limiting the field of vision of the agent to its surroundings, rather than the entire image. We evaluate different versions of our approach: (1) DCRAC with standard experience replay; (2) DCRAC-M which uses memory networks instead of the recurrent layer along with standard experience replay; (3) DCRAC+DER, in which DCRAC uses diverse experience replay; and (4) DCRAC-M+DER in which DCRAC-M uses diverse experience replay. We compare our approach with Conditioned Networks (CN) [1] and use CN+DER, i.e., CN with diverse experience replay as the baseline. Here, CN is given the partially observable view of the environment instead of the full image as the input. We use the cumulative average episodic discounted reward as an evaluation metric. Ideally, as the number of training steps increases, the cumulative episodic discounted reward should increase, demonstrating the learnability of the approach. We also compare the cumulative average episodic regret for the approaches. Regret [1] is calculated as the difference between optimal value (for each weight vector) and actual return. The optimal value is computed using guidelines given in [1].

To compute the actual return, the rewards for the different objectives are scalarized using weight vector $\boldsymbol{w}_e$ for that episode. $\boldsymbol{w}_e$ is randomly sampled from a Dirichlet distribution (α=1) every $5,000$ steps to introduce the dynamic weights setting. The same set of weight vectors are used across all the approaches for fair comparison. We trained using Nvidia RTX 2070 GPU and it took an average of 1 hour to run $100,000$ training steps for DCRAC approach.

For DCRAC and DCRAC-M, the observation input is scaled to $45 \times 45 \times 3$. The first convolution layer contains 32, 6×6 filters, with stride 2. The second convolution layer contains 48, 5×5 filters with stride 2. Each convolution layer is followed by a max-pooling layer. The features extracted from the convolution layer are then fed into another dense layer of 512 units and then concatenated with the action input, encoded as a one-hot vector. For DCRAC,

Algorithm	Overall				Last 25k steps			
	Standard ER		DER		Standerd ER		DER	
	Mean reward	vs. baseline	Mean reward	vs. baseline	Mean reward	vs. baseline	Mean reward	vs. baseline
CN*	–	–	10.373	–	–	–	7.199	–
DCRAC	10.377	+0.038%	10.491	+1.138%	7.513	+4.362%	8.173	+13.530%
DCRAC-M	10.354	-0.183%	10.248	-1.205%	7.642	+6.154%	7.251	+0.722%
	Mean regret	vs. baseline	Mean regret	vs. baseline	Mean regret	vs. baseline	Mean regret	vs. baseline
CN*	–	–	3.755	–	–	–	0.459	–
DCRAC	3.773	+0.479%	3.324	-11.478%	0.590	+28.540%	0.306	**-33.333%**
DCRAC-M	5.585	+22.104%	3.845	+2.397%	0.846	+84.314%	0.352	-23.312%

*CN+DER is the baseline used for comparison.

Table 1: Cumulative Episodic Discounted Reward and Regret for 5×5 partial view in DST

we use a 64-unit LSTM. The LSTM is unrolled for 10 time-steps. For DCRAC-M, which uses a memory network, the length of the external memory is 32. The memory network also uses a 64-unit LSTM, unrolled for 10 time-steps. For the separate actor and critic streams each of the fully connected (FC) layers are of size 256. The remaining are dense layers with size as mentioned in Fig. 1.

4.2 Deep Sea Treasure (DST)

In DST, a submarine should dive to collect treasures on the sea floor. The environment is a 12×12 RGB map as shown in Fig. 4(a) with treasures encoded in green (0,255,0), water encoded in blue (0,0,255) and sea-floors encoded in red (255,0,0). The submarine agent is encoded in white (255,255,255). The agent only has a partial view of the environment, which is a 5×5 or 3×3 view of the surrounding, with the agent in the center as shown in Fig. 4(c),(d). The regions in the 5×5 or 3×3 view which fall outside the original RGB map are filled in black (0,0,0). The agent can move *up*, *down*, *left* and *right* anywhere within the map except the seafloor. The reward is 2-dimensional: first dimension representing the value of the treasure collected, second dimension representing the penalty for power consumption when moving. The penalty is always set to -1 for each movement. The reward for each treasure is shown in Fig. 4(a).

4.3 Results

Table. 1 shows the cumulative average episodic discounted rewards and cumulative average episodic regret for the DST problem with 5×5 view. The values are computed every 100 steps and the experiments are run for 100, 000 steps. The results for the last 25, 000 steps is also reported. We can see that DCRAC+DER performs better than the baseline CN+DER by 13.53% in terms of rewards and 33.33% in terms of regret for last 25, 000 steps. DCRAC-M+DER obtains a regret of 0.352, also better than the baseline.

Fig. 5 shows how the cumulative average episodic discounted reward increases with time steps for the last 25, 000 steps to show the results after the convergence of the model. We can see that, with time, the approaches are able to learn better policies and obtain better rewards. However, DCRAC+DER learns faster and is able to obtain better rewards at final stage when compared to the baseline. Fig. 6 shows the cumulative average episodic regret. We can see that DCRAC+DER stabilizes effectively with the number of training steps, obtaining the best performance, i.e., higher reward and lower

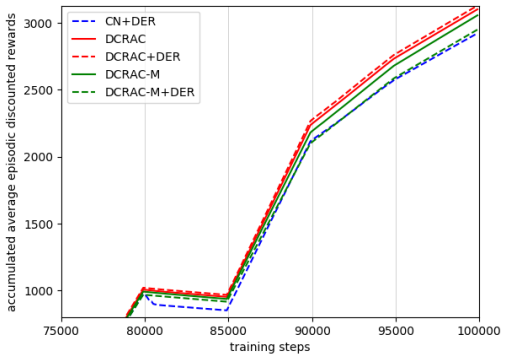

Figure 5: Cumulative Avg. Episodic Rewards with 5×5 view

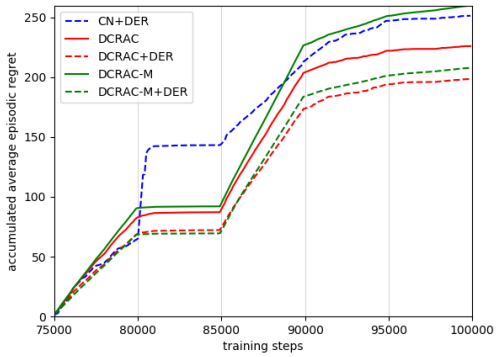

Figure 6: Cumulative Avg. episodic regret with 5×5 view

regret. The vertical lines in the figures represent the weight changes, which happens every 5, 000 steps.

Table. 2 shows the average episodic cumulative rewards and regret for the 3×3 partial view. This environment is more partially observable than the 5×5 view. DCRAC+DER outperforms all approaches with a mean regret of 2.591 outperforming the baseline by 11.78% for last 25, 000 steps. From the experiment, we can see that using DER always produces better results than using standard ER for both 5×5 and 3×3 view for the dynamic weights setting.

Algorithm	Overall				Last 25k steps			
	Standard ER		DER		Standerd ER		DER	
	Mean reward	vs. baseline	Mean reward	vs. baseline	Mean reward	vs. baseline	Mean reward	vs. baseline
CN*	–	–	8.883	–	–	–	7.085	–
DCRAC	8.923	+0.450%	9.433	+6.192%	7.642	+7.862%	8.525	+20.325%
DCRAC-M	8.437	-5.021%	9.131	+2.792%	6.653	-6.097%	8.843	+24.813%
	Mean regret	vs. baseline	Mean regret	vs. baseline	Mean regret	vs. baseline	Mean regret	vs. baseline
CN*	–	–	4.689	–	–	–	2.937	–
DCRAC	4.856	+3.562%	4.053	-13.564%	3.360	+14.402%	2.591	**-11.781%**
DCRAC-M	4.385	-6.483%	4.321	-7.848%	3.164	+7.729%	2.676	-8.887%

*CN+DER is the baseline used for comparison.

Table 2: Cumulative Average Episodic Reward and Regret for 3×3 partial view in DST

5 CONCLUSION

We propose DCRAC, a Deep Recurrent Actor-Critic approach, for decision making in partially-observable multi-objective environments. The actor-critic network is conditioned on the weights, i.e., the preferences of different objectives. Hence, DCRAC can generalize to different weights and can handle scenarios where the weights change dynamically over time. We also propose DCRAC-M, which uses memory networks for remembering long-term dependencies in the action-observation histories. We use diverse experience replay to sample transitions while training the network to prevent overfitting on recently trained weights. Experiments on the partially-observable version of DST shows that the DCRAC outperforms Conditioned Networks. As future-work, we plan to conduct experiments on more complex scenarios such as minecart.

REFERENCES

[1] Axel Abels, Diederik M Roijers, Tom Lenaerts, Ann Nowé, and Denis Steckelmacher. 2018. Dynamic Weights in Multi-Objective Deep Reinforcement Learning. *arXiv preprint arXiv:1809.07803* (2018).

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[3] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[4] Matthew Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*.

[5] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. 2015. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455* (2015).

[6] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101, 1-2 (1998), 99–134.

[7] Seungchan Kim, Kavosh Asadi, Michael Littman, and George Konidaris. 2019. DeepMellow: Removing the Need for a Target Network in Deep Q-Learning. (2019).

[8] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[10] Hossam Mossalam, Yannis M Assael, Diederik M Roijers, and Shimon Whiteson. 2016. Multi-objective deep reinforcement learning. *arXiv preprint arXiv:1610.02707* (2016).

[11] Junhyuk Oh, Valliappa Chockalingam, Satinder Singh, and Honglak Lee. 2016. Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128* (2016).

[12] Christos H Papadimitriou and John N Tsitsiklis. 1987. The complexity of Markov decision processes. *Mathematics of operations research* 12, 3 (1987), 441–450.

[13] Julien Perez and Tomi Silander. 2017. Non-markovian control with gated end-to-end memory policy networks. *arXiv preprint arXiv:1705.10993* (2017).

[14] Diederik Marijn Roijers, Shimon Whiteson, and Frans A Oliehoek. 2015. Point-based planning for multi-objective POMDPs. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

[15] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.

[16] Bradly C Stadie, Sergey Levine, and Pieter Abbeel. 2015. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814* (2015).

[17] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Advances in neural information processing systems*. 2440–2448.

[18] Hado Van Hasselt, Arthur Guez, and David Silver. 2016. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*.

[19] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581* (2015).

[20] Daan Wierstra, Alexander Foerster, Jan Peters, and Juergen Schmidhuber. 2007. Solving deep memory POMDPs with recurrent policy gradients. In *International Conference on Artificial Neural Networks*. Springer, 697–706.

[21] Pengfei Zhu, Xin Li, Pascal Poupart, and Guanghui Miao. 2017. On improving deep reinforcement learning for pomdps. *arXiv preprint arXiv:1704.07978* (2017).