Multi-task Deep Reinforcement Learning via Policy Distillation

Author: Clement Wan[1]

Degree: MSc Machine Learning

Supervisor: John Shawne-Taylor

Submission date: 6th of September, 2019

[1]**Disclaimer:** This report is submitted as part requirement for the MSc Machine Learning at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Acknowledgements

thanks

Abstract

This report explores the problem of Multitask Reinforcement Learning through capturing commonalities across task-specific policies and condensing into a distilled policy. We examine algorithms introduced by Teh et al. [33] by first discussing the origins of these algorithms, and then conducted extended experiments to study the algorithms' behaviours. Through experimentation we also derived a variant of one of the original algorithms, called Distral 1 col beta, which significantly outperforms its predecessor under the soft Q-learning framework.

Contents

Chapter 1

Introduction and Key Results

1.1 Objectives and Contributions

There are 2 main variant of algorithms from the original work by Teh et al. The first algorithm is called Distral 1 column or Distral 1 col. The second algorithm is called Distral 2 column or Distral 2 col. Both algorithms can be implemented in either a soft Q-learning (SQL) or regularized Actor-Critic (AC) framework. The original work has tested Distral 2 col via SQL versus SQL baseline on 4 2D mazes. All 2D mazes have the same structure and only differ in the starting and goal states, **as shown in figure ...**. The experiment was conducted using a 10 step TD learning, where maximum number of steps per episode is 100 for 1000 episodes. We replicated this experiment with identical conditions and have achieved results consistent with the original work's reported performance, as demonstrated in **section ...**. The original work has also tested Distral 1 col and 2 col vis SQL on numerous DeepMind Lab environments, including 8 3D mazes, 4 Navigation tasks, and 8 laser tag environments. Note that each variant for each type of environment was trained on 10 billion frames.

We have decided to focus our efforts in testing studying the behaviour of the algorithms. This requires numerous deployments of algorithms, thus we have exclusively tested the algorithms on the 2D maze to significantly reduce training time. Specifically, our contributions include extended experiments on Distral 1 col and 2 col under both the SQL regularized AC frameworks. We also altered the settings such that all algorithms are doing a 1-step TD learning with 1000 steps as maximum number of steps per episode for 200 episode instead. The motivation for doing so is because testing the algorithms with 10

steps TD learning for 1000 episodes is substantially more time consuming than using a 1 step TD learning for 200 episodes. Furthermore, we decided that changing the maximum number of step per episode from 1000 to 200 is a reasonable compensation for lowering the number of episodes trained in total.

In addition to the extended testing of Distral 1 col and 2 col on both SQL and regularized AC frameworks, we have also discovered a new variant of Distral 1 col during the process of understanding why Distral 1 col SQL performs worse than Distral 2 col SQL. This algorithm significantly outperforms its predecessor on the 2D mazes. We call this algorithm Distral 1 col beta, because of the additional β coefficient multiplied to calculate the task-specific policies $\pi_i(a_t|s_t)$. We demonstrate this new algorithm's ability to rival its counterpart Distral 2 col, as shown in **section ...**.

Chapter 2

Literature Review

2.1 Prelude

Reinforcement Learning (RL) is a subject whose core goal is to learn to make "good" decisions through an `agent` interacting with the `environment`. An example of a reinforcement learning problem could be learning to play the game of chess, where the agent is a player and the environment is **both** the opponent and the board. Another example could be controlling a power station where the agent is the entity that controls the dials and buttons responsible for producing power, and the environment is the rest of the power plant. Note that key differentiation between agent and environment is that everything within the agent's control becomes part of the agent, and every external reaction to the agent's decisions is part of the environment.

The agent-environment interaction is typically described as follows. At each timestep, the agent receives an `observation` and `reward` from the environment (except on the first step). The agent then picks an `action` according to the given observation. In turn, the environment receives an action and reacts by returning a reward signal and observation to the agent.

The reward signal is a scalar that serves as an indication to quantify how well the agent is the doing at that time. Therefore, maximizing the amount of rewards in the long run aligns with the goal to make "good" decisions through interacting with the environment.

That is, the core goal of an RL problem is to maximize the following

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots = \sum_{k=0}^{\infty} R_{t+k+1} \tag{2.1}$$

where G_t is called `return` and is the cumulative reward from timestep t. An agent's sequence of interaction up to some timestep t is

$$H_t = O_0, A_0, R_1, O_1, \cdots O_{t-1}, A_{t-1}, R_t, O_t$$

, where H_t is the history of interactions, O_t is an observation, R_t is a reward and A_t is an action. At each timestep, the agent picks an action depending on its internal state. We can define an agent's state as a function of its history $S_t^a = f(H_t)$. Specifically, the agent's state contains information used for action selection, where the information is a summary of the history thus far. In addition, the environment also maintains an internal state defined as S_t^e. The environment's internal state contains information needed to send an observation O_t and a reward R_t to the agent.

Crucially, note that both agent and environment simultaneously maintain their corresponding internal states, and the agent does not always have access to the environment's internal state. However, if the environment is **fully** observable, then the agent directly observes the environment and can make the environment's state as its internal state. That is to say $S_t^a = S_t^e = O_t$, and thus we only need to keep track of the agent's state. This implies that the agent is a Markov Decision Process[1] and its interactions with the environment can be illustrated [29] as follows

[1]If the environment's state is only partially observable, then a Partially-Observable Markov Decision Process is used instead to describe the dynamics between the agent and the environment.

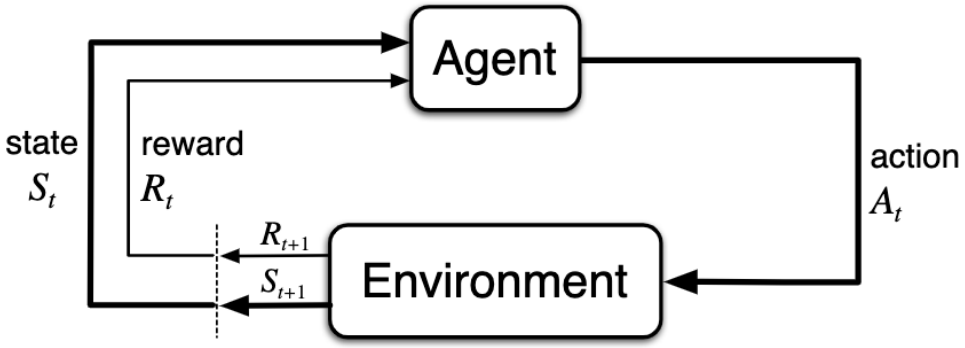


Figure 2.1: Agent-Environment Interface

resulting in a sequences of transitions

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \cdots \tag{2.2}$$

Naturally, we can see that the interaction between an agent and the environment is time dependent. Note that a complete sequence of transitions is called an `episode`. That is, an episode records all transitions from the start state until the agent reaches the terminal state or the maximum number of timesteps. In addition, we define the reward to arrive one time step later than its corresponding action[2].

Furthermore, we also see that the minimum necessary information for the agent to make a decision at each timestep t is S_t. This indicates the agent-environment interaction follows the Markovian property

$$P(A_t|S_0, S_1, \cdots, S_t) = P(A_t|S_t) \tag{2.3}$$

where the probability of selecting an action given current state is equivalent to the entire trajectory of states. That is, the action selection process is independent of the trajectory given the current state.

2.1.1 Markov Decision Process (MDP)

First, consider a Markov Reward Process (MRP), which is a formal way of describing an environment via a 4-tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ where **give ex of what S and A is**

[2]Note that this is the conventional definition. It is also valid to define actions and rewards such that they occur in the same timestep.

- $\mathcal{S}$ is a finite set of states
- $\mathcal{P}$ is the state transition probability matrix, and $P(S_{t+1}|S_t)$ is the probability of transition to the next state given the current state and action
- $\mathcal{R}$ is the reward function, and $\mathbb{E}[R_{t+1}|S_t]$ is the agent's expected reward given it is in some state
- $\gamma \in [0, 1]$ is a discount factor applied on return G_t

and the dynamics of agent-environment satisfies Markov property. A discount factor γ is introduced such that the return is modified from equation (2.1) to

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.4}$$

where the discount factor allows us to weight the importance of rewards proportionally according to how far in they are in the future. This is useful for several reasons. For example, the discount factor is mathematically convenient and avoids infinite sums in scenario where there isn't a terminal state. In addition, we may be motivated to weight future rewards less in situations where we favor immediate rewards.

The setup of MRP leads allows for a formal description to evaluate the quality of states. Specifically, the state-value function $V(s)$ is a function that evaluates the value of a state in the long run. That is, $V(s)$ is the expected value of the return given the agent starts at state s

$$V(s) = \mathbb{E}[G_t|S_t = s] \tag{2.5}$$

where the return can be expanded to derive a recursive relationship between pairs of

consecutive states

$$\begin{aligned}
V(s) &= \mathbb{E}[G_t|S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots\right) |S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma V(S_{t+1})|S_t = s] \\
&= R_{t+1} + \gamma \sum_{s' \in \mathcal{S}} P(S_{t+1} = s'|S_t)V(s')
\end{aligned}$$

as demonstrated in the second last expression. Also, note that last line of expression for $V(s)$ shows that the expectation is taken over transition probabilities from one state to the next. This implies a Markov Reward Process does not involve any decision making. That is, the agent does not have autonomy in the decision making process, and is merely transitioning between states as dictated by the environment's dynamics.

To complete the picture, we need to allow the agent to make its own decisions and quantify how the agent behaves when selecting an action. This leads to the policy of an agent $\pi(a|s)$, which is a distribution over all possible actions when in some state s

$$\pi(a|s) = P(A_t = a|S_t = s) \tag{2.6}$$

Now, we can add the policy π and some finite action space $\mathcal{A}$ to the original Markov *Reward* Process to obtain a Markov *Decision* Process (MDP). This results in a MDP 5-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}_\pi, \mathcal{R}_\pi, \gamma)$ and a policy π, where the transition probability of a pair of states is

$$P_\pi(S_{t+1}|S_t, A_t) = \mathbb{E}_\pi[S_{t+1}|S_t, A_t] = \sum_a \pi(a|s)P(S_{t+1}|S_t = s, A_t = a)$$

and expected reward given a state-action pair is

$$R_\pi(S_t, A_t) = \mathbb{E}_\pi[R_{t+1}|S_t, A_t] = \sum_a \pi(a|s)P(R_{t+1}|S_t = s, A_t = a)$$

In addition, we define a action value function $Q_\pi(s, a)$, which evaluates the value of the state-action pair in the long run. That is, $Q(s, a)$ is the expected value of the return given

the agent starts at state s and takes action a, and follows policy π from there onward

$$\begin{aligned} Q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_t + \gamma Q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \end{aligned}$$

which also decomposes into a recursive relationship as demonstrated. Furthermore, we can redefine the value function $V_\pi(s)$ as the value estimate of a state under policy π. This allows us to relate state value function and action value function in the following sense

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi\left[R_t + \gamma V_\pi(S_{t+1})|S_t = s\right] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) Q_\pi(s,a) \end{aligned} \tag{2.7}$$

$$Q_\pi(s,a) = R_t + \gamma \sum_{s'\in\mathcal{S}} P(S_{t+1} = s'|S_t = s, A_t = a) V_\pi(s') \tag{2.8}$$

Notice that unlike the value function $V(s)$, probability transitions $\mathcal{P}$ and reward function $\mathcal{R}$ defined for a Markov *Reward* Process, the corresponding functions for a Markov *Decision* Process are taking expectations over π.

2.1.2 A Note on Model-Based vs Model-Free RL

Before discussing any RL algorithms in detail, we must first make a distinction between model-based and model-free reinforcement learning. These two implementations are fundamentally different in that a model-based RL agent explicitly models the environment it is interacting with. An example would be Dina-Q, an algorithm that models the environment and samples transitions from that model in order to "plan" its moves in the real environment.

An example model-free RL agent would be the classical Q-learning agent, in which the agent has no knowledge regarding to the environment beyond the action value estimate which approximates the value of a certain state-action pair $Q_\pi(s,a)$.

In this report, we will focus on model-free RL agents, because the main school of thoughts regarding to multitask reinforcement learning deems explicit estimation for each task's environment too expensive and thus not scalable [27].

2.1.3 Policy Iteration

start with a clearer definition of policy iteration Policy iteration is one of the core ideas that guarantees the success of reinforcement learning agents. Policy iteration consists of two mechanisms, policy improvement and policy evaluation.

These two mechanisms are represented by four Bellman Equations [29] [32]

$$V_\pi(s) = \mathbb{E}[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t \sim \pi(S_t)] \tag{2.9}$$

$$Q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \tag{2.10}$$

$$V_*(s) = \arg\max_a \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \tag{2.11}$$

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma \max_{a'} Q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{2.12}$$

, where V_* and Q_* represent the true values of states and state-action pairs. That is, they are the maximum state and action value functions over all policies

$$V_*(s) = \max_\pi V_\pi(s) \tag{2.13}$$

$$Q_*(s,a) = \max_\pi Q_\pi(s,a) \tag{2.14}$$

Equations (2.9) and (2.10) are referred as predictions as they evaluate the value of a state or a state-action pair. For example, states closer to the goal state will have higher values than otherwise. Equations (2.11) and (2.12) are classically referred to as control or optimal policies. This is because given the true values V_* and Q_*, the maximization over actions must yield the most amount of rewards.

Consider the goal to estimate the state-value function V_π for an arbitrary policy π. We can start with an approximate value function V_0 and iteratively improve our estimation of V_π via the following update rule

$$V_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma V_k(S_{t+1}) | S_t = s] \tag{2.15}$$

$$= \sum_a \pi(a|s) \sum_{s',r} P(s', r | s, a)[r + \gamma V_k(s')] \tag{2.16}$$

and arrive at the following algorithm [29] for estimating V_π

Iterative policy evaluation

Input π, the policy to be evaluated
Initialize an array $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx v_\pi$

Now, if we consider the control equation (2.11) maximizing over iterative states value estimates V_{k+1} instead of the true values V_*, then the control equation becomes policy improvement. Let V'_{k+1} be the new policy derived from maximizing V_{k+1}. Then we are guaranteed to have $V'_{k+1} \geq V_{k+1}$ according to the Policy Improvement Theorem [29].

We can then estimate V_{k+2} based on V'_{k+1} instead of V_{k+1} to obtain the full cycle of policy iteration, as illustrated in the following diagram from David Silver's lecture notes [28]

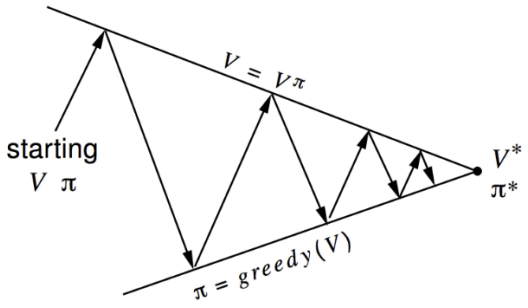


Figure 2.2: Visualization of Policy Iteration

As shown above, the iterative policy evaluation and improvement eventually converges to the optimal policy and state value function, where each iteration is a strict improvement over the last until convergence. A demonstration [29] of the policy iteration algorithm is shown in detail below

Policy iteration (using iterative policy evaluation)

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      old-action $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

It should also be noted that all algorithms discussed in the following sections will be operating under the policy iteration framework.

2.1.4 Table of definitions

Here is a list of definitions and terminologies first introduced in the previous sections, and reiterated in a list for clarity. Note that we are following conventional statistical notations, in that all capital notations are **random variables** and all small letter notations are **constants**. **change the phrasing of arbitrary timestep of at time t**

Terminology	Definition
t	Timestep of an episode
T	Number of timesteps reached until episode terminates
a / A_t	An agent's action at an arbitrary timestep or at time t
$R(s,a)$ / R_t	Reward obtained during or after taking one step in the Markov Decision Process
s / S_t	State of the agent at an arbitrary timestep or at time t
γ	Discount value
$\pi(a\|s)$	Policy of an agent
G_t	Cumulative rewards obtained from timestep t until episode terminates
$Q_\pi(s,a)$	Value of a state-action pair under policy π
$V_\pi(s)$	Value of a state under policy π
Q_*	True/optimal action value function
V_*	True/optimal state value function

2.2 Q-learning

First introduced by Watkins et al. [35], Q-Learning is an off-policy model-free algorithm. Under the policy iteration framework, the representation of action values are updated according to the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2.17}$$

This is an off-policy algorithm because of the maximization of Q values over actions as supposed to using the agent's behaviour policy[3]. Below is Q-learning's pseudo-code extracted from Sutton and Barto [29]

[3]Behaviour policy means the policy used by the agent to obtain a transition with the environment. In the case of the pseudocode above, it is the first line in the loop for each step of an episode.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

It is also known that this algorithm performs worse during training but does better than its counterparts in the testing phase.

2.2.1 Deep Q-Learning (DQN)

The concept of Q-learning is further developed into an architecture called Deep Q-learning (DQN) [14], which have demonstrated several advantages in replacing consecutive transitions with experience replay in the training process.

The idea of experience replay is to store the agent's interaction with the environment in a replay buffer and update state-action estimates by sampling from it. Mnih et al. [14] suggests that using a replay memory improves data efficiency as each transition can be sampled multiple times. Furthermore, they argue that sampling decoupled transitions instead of learning from consecutive ones lowers the variance of action value estimates, and thus making the agent more robust to oscillating behaviour in performance. Below is the detailed psuedocode for DQN from Mnih et al. [14]

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

2.3 Policy Gradient

An alternative to value-based methods is policy-based methods, that is to directly optimize an agent's policy $\pi(a_t|s_t)$ directly rather than indirectly through, say ϵ-greedy of action value estimates $Q(s_t, a_t)$.

One of the most compelling motivation for us to optimize against π instead of Q or V is that in some circumstances, it is much easier for the agent to learn an optimal policy π_* compared to learning an optimal value function Q_* or V_*. For example, an agent may learn that the optimal policy is to always move right in a 2D platformer such as Sonic Mania (2.3), because the end of each level/stage is always to the right of its current position. Whereas, the accurate action values of each frame may be very difficult to learn due to the complex nature of the environment.

Figure 2.3: Sonic Mania, a 2D platformer

In addition, policy-based methods also has some advantages over value-based methods. Specifically, policy-based methods have better convergence properties, are effective in high dimensional or continuous spaces, and can learn stochastic policies. Policy-based methods are better than the conventional ϵ-greedy action selection simply because parameterization of action selection probabilities change smoothly with respect to its learned parameters [29]. In contrast, ϵ-greedy policy can change drastically due to a small perturbation because of the `argmax` mechanism over action value estimates.

Moreover, policy-based methods are also convenient for learning the action selection probabilities in a very large or continuous action space. For example, a continuous action space can be parameterized as a Gaussian policy such that

$$\pi(a|s,\theta) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left(-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right) \tag{2.18}$$

, where the π in the denominator is the irrational number $3.14159\cdots$ and the mean μ and standard deviation σ are parameterized by θ.

Furthermore, an example where stochastic policies are required to solve a problem can be demonstrated in a simple maze (figure 2.4) from [25]

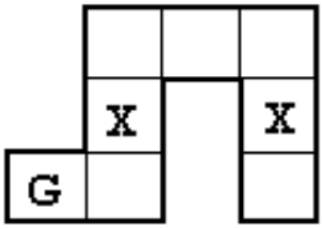


Figure 2.4: Simple maze with alias states

As seen above, the states with "x" are identical to an agent as both states represent a corridor where the agent can only move up or down. However, the agent has to move up in the right alias state and down in the left alias state in order to reach the goal state. A value-based method would have difficulty evaluating the state-action estimates of those states as they are aliases of each other. Whereas, a policy-based method may learn that the optimal policy is to remain stochastic at the alias states to guarantee convergence to the goal state.

2.3.1 Problem setting

Under the standard MDP framework, consider the goal to find the best θ given some policy $\pi_\theta(a_t|s_t)$. There are several common approaches to evaluate the quality of π_θ [29], [30]:

- In an episodic environment, we can define performance as $J(\theta) = V_{\pi_\theta}(S_0)$
- For continuous problems without a terminal state, we can define performance as the average rate of reward per timestep $J(\theta) = \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)r$, where $\mu(s)$ is the stationary distribution[4] of the MDP

Consider an episodic environment, then we can compute the gradient of $J(\theta)$ analytically, assuming π_θ is differentiable everywhere when $\pi_\theta \neq 0$. Furthermore, assume that we are dealing with a one-step episodic bandit scenario such that

$$J(\theta) = \mathbb{E}_{\mu,\pi}[R(S, A)] \tag{2.19}$$

, where the expectation is taken of μ and π, and μ is the stationary distribution of states and π is the policy over actions.

[4]A stationary distribution of a MDP is defined as the expected number of visits for each state as the number of transitions approaches infinity.

Let $\pi_\theta = \pi$ to simplify notations, then the gradient w.r.t. the objective is

$$\begin{aligned}
\nabla_\theta \mathbb{E}[R(S,A)] &= \nabla_\theta \sum_s \mu(s) \sum_a \pi(a|s) R(s,a) \\
&= \sum_s \mu(s) \sum_a \nabla_\theta \pi(a|s) R(s,a) \\
&= \sum_s \mu(s) \sum_a \pi(a|s) \frac{\nabla_\theta \pi(a|s)}{\pi(a|s)} R(s,a) \qquad (2.20) \\
&= \mathbb{E}[\nabla_\theta \log \pi(a|s) R(s,a)]
\end{aligned}$$

, where the transition from the second last (2.20) to the last line is commonly referred to as the score-function trick or log likelihood ratio identity. This expression can the be sampled and updated incrementally via stochastic gradient descent.

2.3.2 Policy Gradient on Trajectories

Now consider instead of a one-step bandit scenario, we have an episodic environment that lasts over multiple steps. That is, each episode consists of a trajectory

$$\zeta = S_0, A_0, R_1, S_1, A_1, R_2, S_2 \cdots$$

with return as $G(\zeta)$. Assume the discount factor $\gamma = 1$ for simplicity, then the policy gradient becomes

$$\begin{aligned}
\nabla_\theta J_\theta(\pi) &= \nabla_\theta \mathbb{E}[G(\zeta)] \\
&= \mathbb{E}[G(\zeta) \nabla_\theta \log P(\zeta)] \qquad (2.21)
\end{aligned}$$

via the score function trick as mentioned in (2.20). Expanding the derivative w.r.t. to $\log P(\zeta)$ gives

$$\begin{aligned}
\nabla_\theta \log P(\zeta) &= \nabla_\theta \log \left[P(S_0)\pi(A_0|S_0) \prod_{t\geq 1} P(S_t|S_{t-1}, A_{t-1})\pi(A_t|S_t) \right] \\
&= \nabla_\theta \left[\log P(S_0) + \log \pi(A_0|S_0) + \sum_{t\geq 1} \log P(S_t|S_{t-1}, A_{t-1}) + \sum_{t\geq 1} \log \pi(A_t|S_t) \right] \\
&= \nabla_\theta \left[\sum_{t\geq 0} \log \pi(A_t|S_t) \right]
\end{aligned}$$

Substituting the expression above back into the partial derivative w.r.t. the performance of the policy

$$\begin{aligned}
\nabla_\theta J_\theta(\pi) &= \mathbb{E}\left[G(\zeta) \nabla_\theta \sum_{t\geq 0} \log \pi(A_t|S_t) \right] \\
&= \mathbb{E}\left[\sum_{t\geq 0} R_{t+1} \nabla_\theta \sum_{t\geq 0} \log \pi(A_t|S_t) \right]
\end{aligned}$$

Also, since $\sum_{t=0}^{k} R_{t+1}$ doesn't depend on actions $A_{k+1}, A_{k+2}, \cdots$, we can rewrite and simplify (2.21)

$$\begin{aligned}
\nabla_\theta J_\theta(\pi) &= \mathbb{E}\left[\sum_{t\geq 0} R_{t+1} \nabla_\theta \sum_{t\geq 0} \log \pi(A_t|S_t) \right] \\
&= \mathbb{E}\left[\sum_{t\geq 0} \nabla_\theta \log \pi(A_t|S_t) \left(\sum_{i=t} R_{i+1} \right) \right] \qquad (2.22)
\end{aligned}$$

Note that if $\gamma \neq 1$, then

$$\nabla_\theta J_\theta(\pi) = \mathbb{E}\left[\sum_{t\geq 0} \nabla_\theta \log \pi(A_t|S_t) \left(\sum_{i=t} \gamma^{i-t} R_{i+1} \right) \right] \qquad (2.23)$$

2.3.3 Policy Gradient Theorem

The method of rewriting the expression of policy gradient stated in the context of a one-step and n-step bandit in the previous sections can be generalized into a theorem. Specifically, policy gradient theorem states that [29] given any differentiable policy $\pi(s|a)$ and any

policy objective function $J(\theta)$ mentioned above, the policy gradient is

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_s \mu(s) \sum_a \pi(a|s) R(s,a) \\ &= \mathbb{E}_\pi[\nabla_\theta \log \pi(a|s) Q_\pi(s,a)] \end{aligned} \tag{2.24}$$

Note that rewriting the gradient of the expected value into expected value of a gradient is important as the second expression circumvents the problem of depending on the action selection and the stationary distribution of states, which are both determined by the policy $\pi(a|s)$.

2.3.4 REINFORCE

One of the earliest policy gradient algorithms is the REINFORCE algorithm [36], as state in equation 2.22 in the previous sections

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)] \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) G_t] \end{aligned} \tag{2.25}$$

, where $G_t = \sum_{t'>t} R_{t'}$ is the return or cumulative reward from time timestep t onward. This is because $\mathbb{E}_{\pi_\theta}[G_t|S_t, A_t] = Q_{\pi_\theta}(S_t, A_t)$ by definition.

Note that REINFORCE is a Monte-Carlo algorithm, in that it uses the entire trajectory of returns up until the final transition of the episode. Here is the pseudocode from Sutton and Barto [29]

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$

2.4 Advantage Actor Critic Model (A2C)

Another well known variant of policy gradient is the Advantage Actor-Critic (A2C) algorithm, which is a union between policy gradient and state value function estimate. That is, the policy gradient plays the role of an actor and the state value function estimate plays the role of the critic by evaluating the value of states.

Consider using the same policy gradient as REINFORCE, and then construct a critic which models $Q_\pi(s,a)$ explicitly with another function approximation, such as a neural network[5] such that $Q_w(s,a) \approx Q_\pi(s,a)$ for some parameters w. Then

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q_{\pi_\theta}(s,a)] \\ &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)] \end{aligned} \tag{2.26}$$

, where $Q_w(s,a)$ is analogous to policy evaluation as mentioned in section 2.1.3. Note that the transition from the first to the second line is only equivalent when the state-value function satisfies the Compatible Function Approximation Theorem [34] [28]. Specifically, the theorem is only satisfied if and only if the value function is compatible with the policy and the value function parameters w minimizes a least squares loss function.

We can then update the parameters w by performing 1-step TD-learning via least squares

$$\begin{aligned} L(Q_w) &= \frac{1}{2}\mathbb{E}\left[(\hat{Q}_w(S_t,a) - Q_w(S_t,a))^2\right] \\ \hat{Q}_w(S_t,a) &= R_{t+1} + \gamma Q_w(S_{t+1}, A_{t+1}) \\ \text{where } A_{t+1} &\sim \pi \end{aligned}$$

Furthermore, subtracting $Q_w(s,a)$ by a baseline can reduce the variance of the policy gradient, which is allowed as the expectation of the gradient remains the same. For example,

[5]Note that actor and critic networks can either be represented by the same or separate neural network [5].

let $B(s)$ be the baseline we wish to subtract $Q_\pi(s,a)$ from, then

$$\begin{aligned}\mathbb{E}[\nabla_\theta \log \pi(a|s)B(s)] &= \sum_s \mu(s) \sum_a \nabla_\theta \pi(a|s)B(s) \\ &= \sum_s \mu(s)B(s)\nabla_\theta \sum_a \pi(a|s) \\ &= 0\end{aligned} \tag{2.27}$$

, where the baseline is moved outside of the partial derivative because it is independent of θ. A typical baseline to use is the state-value estimate $V_v(s)$, which can also be estimated using another explicit function approximation with parameters v. Rewriting the policy gradient obtains the following

$$\nabla_\theta J_\theta(\pi) = \mathbb{E}\left[\sum_{t\geq 0} \nabla_\theta \log \pi(A_t|S_t)(Q_w(S_t,A_t) - V_v(S_t))\right] \tag{2.28}$$

$$= \mathbb{E}\left[\sum_{t\geq 0} \nabla_\theta \log \pi(A_t|S_t)A(S_t,A_t)\right] \tag{2.29}$$

, where $A(S_t,A_t)$ is called the advantage function as it measures the difference between the value of a particular state-action pair versus state, which can be interpreted as the average value across all actions of the state. Again, both Q_w and V_v can be updated separately via 1-step or n-step TD learning.

However, there is an alternative way of estimating the advantage function which is more efficient. Specifically, we know that the TD error δ_{π_θ} is an unbiased estimate [28] of the advantage function given the true value function $V_{\pi_\theta}(s)$

$$\begin{aligned}\mathbb{E}[\delta_{\pi_\theta}|s,a] &= \mathbb{E}[r + \gamma V_{\pi_\theta}(s') - V_{\pi_\theta}(s)|s,a] \\ &= \mathbb{E}[r + \gamma V_{\pi_\theta}(s')|s,a] - V_{\pi_\theta}(s) \\ &= Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \\ &= A_{\pi_\theta}(s,a)\end{aligned}$$

Thus the advantage function can be replaced by the TD error

$$\nabla_\theta J_\theta(\pi) = \mathbb{E}\left[\sum_{t\geq 0} \nabla_\theta \log \pi(A_t|S_t)\delta_v\right] \tag{2.30}$$

where $\delta_v = r + \gamma V_v(s') - V_v(s)$ is the estimate of δ_{π_θ}. Note that this is more efficient as only one set of parameters v need to be maintained instead of two. Similarly, an n-step TD estimate can be used instead, defining the loss on V_v as

$$L(V_v) = \frac{1}{2}\mathbb{E}\left[\left(\delta_{v,t+n}\right)^2\right] \tag{2.31}$$

$$= \frac{1}{2}\mathbb{E}\left[\left(R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} V_v(S_{t+n}) - V_v(s_t)\right)^2\right] \tag{2.32}$$

with an identical policy gradient as mentioned above except replacing δ_v with $\delta_{v,t+n}$. Finally, here is the detail pseudocode for a one-step A2C obtained from Sutton and Barto [29]

One-step Actor–Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

2.4.1 Asynchronous Advantage Actor-Critic (A3C)

Asynchronous Advantage Actor-Critic (A3C) is a variant of A2C in that there are multiple workers each learning on a different trajectory, and what is learned is shared to the master node. The policies of master and worker nodes are updated in an asynchronous manner, making the learning more stable. Mnih et al. [13] argues this variant of actor-critic learns faster than the conventional A2C framework at a lower computational cost. Below is a pseudocode for A3C from the original work [13]

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

2.5 Maximum Entropy Reinforcement Learning

2.5.1 Bandit setting

Consider the Bandit setting where each episode is 1 timestep, and at each time t, we choose an action a and receive a reward r sampled from $P(r|a)$ and rewards distribution is unknown.

Define $\bar{r}(a) = \mathbb{E}(r|a) = \sum_r rP(r|a)$ and let π be the policy representing probabilities over

actions. Then the expected per-timestep reward of π is

$$\mathbb{E}[r] = \sum_a \pi(a)\bar{r}(a)$$

Conventionally, we would want to maximize the expected rewards as stated above. However, under entropy regularized setting, we want to maximize $\eta(\pi)$, and entropy version of the original objective instead:

$$\eta(\pi) = \mathbb{E}_\pi[r] - \frac{1}{\beta}\mathrm{KL}[\pi\|\pi_0] \tag{2.33}$$

$$= \mathbb{E}_\pi\left[\bar{r}(a) - \frac{1}{\beta}\log\frac{\pi(a)}{\pi_0(a)}\right] \tag{2.34}$$

, where π_0 [6] is some reference policy, and $\frac{1}{\beta}$ is referred to as a temperature parameter[7]. From Schulman et al. [23], $\eta(\pi)$ is maximized by the following quantity

$$\pi^B_{\bar{r}} = \frac{\pi_0(a)\exp(\beta\bar{r}(a))}{\mathbb{E}_{\pi_0}[\exp(\beta\bar{r}(a'))]} \tag{2.35}$$

2.5.2 Solving entropy-regularized bandit problem

The derivation in the previous section assumes that the reward distribution is known s.t. $\bar{r}$ is known. If it is not known initially, then we usually solve the entropy-regularized bandit problem either via policy-based or value-based methods.

For a policy-based approach, we can sample the policy gradient as mentioned previously in section 2.3, as an unbiased estimate. Parameterizing π with θ gives the following gradient as stated by Schulman et al. [23]

$$\nabla_\theta\eta(\pi_\theta) = \mathbb{E}\left[\nabla_\theta\log\pi_\theta(a)r - \frac{1}{\beta}\nabla_\theta\mathrm{KL}[\pi_\theta\|\pi_0]\right] \tag{2.36}$$

where the derivation is analogous to the unregularized bandit problem as demonstrated in section 2.3.1. For a value-based approach, we can parameterize the value function $Q(a)$

[6]Note that the original work [23] uses $\bar{\pi}$ as the reference policy. This is changed to keep notations consistent with the multitask algorithms.

[7]Note that the temperature coefficient is changed from τ to $\frac{1}{\beta}$ to keep notations consistent.

with θ, and optimize the state-value estimates under the actor-critic framework via least squares loss

$$L(\pi_\theta) = \frac{1}{2}\mathbb{E}\left[(Q_\theta(a) - r)^2\right] \tag{2.37}$$

where the gradient of the loss w.r.t θ is

$$\nabla L(\pi_\theta) = \frac{1}{2}\mathbb{E}\left[\nabla_\theta Q_\theta(a)(Q_\theta(a) - r)\right] \tag{2.38}$$

2.5.3 Setup for entropy-regularized MDPs

Consider the entropy-regularized version of MDP[8] such that the conventional discounted return is modified into

$$\begin{aligned} G_t &= \sum_{t=0}^{\infty} \gamma^t \left(R_t - \frac{1}{\beta}\mathrm{KL}[\pi_\theta(\cdot|S_t)\|\pi_0(\cdot|S_t)] \right) \\ &= \sum_{t=0}^{\infty} \gamma^t \left(R_t - \frac{1}{\beta}\mathrm{KL}_t \right) \end{aligned} \tag{2.39}$$

, where $\mathrm{KL}[\pi_\theta(\cdot|S_t)\|\pi_0(\cdot|S_t)] = \mathrm{KL}_t$. Based on this modified return, the new entropy state-value function and entropy state action-value function are

$$\begin{aligned} V_{\pi_\theta}(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \left(R_t - \frac{1}{\beta}\mathrm{KL}_t \right) \Big| S_t = s \right] \end{aligned} \tag{2.40}$$

$$\begin{aligned} Q_{\pi_\theta}(s,a) &= \mathbb{E}[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}\left[R_0 + \sum_{t=1}^{\infty} \gamma^t \left(R_t - \frac{1}{\beta}\mathrm{KL}_t \right) \Big| S_t = s, A_t = a \right] \end{aligned} \tag{2.41}$$

Note that the Q-function does not include the KL penalty for the first reward term, because it is defined to be independent of the action A_0 [23]. This makes some expressions later

[8]Note that this section assumes at $t = 0$ there in an action a_0 and reward r_0, and that rewards are obtained on the same time step as the action.

simpler, and leads to the following between Q_{π_θ} and V_{π_θ}

$$V_{\pi_\theta}(s) = \mathbb{E}[Q_{\pi_\theta}(s,a)] - \frac{1}{\beta}\mathrm{KL}[\pi_\theta(\cdot|s)\|\pi_0(\cdot|s)] \tag{2.42}$$

2.5.4 Soft Q-learning (SQL)

Under the framework of entropy regularized rewards as stated in (2.39), SQL minimizes the following loss function [23]

$$L(Q) = \mathbb{E}\left[\frac{1}{2}\left(Q(s_t,a_t) - y_t\right)^2\right] \tag{2.43}$$

$$y_t = \begin{cases} r_t + \gamma V(s_{t+1}) & \text{1-step TD-learning} \\ \frac{1}{\beta}\mathrm{KL}_t + \sum_{d=0}^{n-1}\gamma^d(r_{t+d} - \frac{1}{\beta}\mathrm{KL}_{t+d}) + \gamma^n V(s_{t+n}) & \text{n-step TD-learning} \end{cases} \tag{2.44}$$

$$\text{where } V(s_{t+1}) = 1/\beta \log \sum_{a'} \pi_0(a'|s_t)\exp\left(\beta Q(s,a')\right) \tag{2.45}$$

$$\mathrm{KL}_t = \mathrm{KL}[\pi_\theta(a_t|s_t)\|\pi_0(a_t|s_t)]$$

Note the major difference between conventional DQN as mentioned in section 2.2.1 is in the definition of the Bellman equation. In the conventional DQN, the state-value estimate is maximizing over actions, as stated in equation (2.17)

$$V(s_t) = \max_{a_t}(Q(a_t,s_t))$$

whereas SQL's state-value estimate (2.45) is as defined above.

2.5.5 Actor-Critic with Regularized Rewards

Actor-Critic under the context of a entropy regularized rewards often [36] have the following policy gradient estimate

$$\mathbb{E}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'\geq t} R_{t'} - \frac{1}{\beta}\nabla_\theta \mathrm{KL}_t\right] \tag{2.46}$$

However, Schulman et al. [23] states that the objective above is not accurate because the entropy regularization only applies to the current timestep, and that a proper entropy

regularized policy gradient would include entropy regularization in the entire trajectory beginning from timestep t. Thus resulting in the following alternative

$$g_\gamma(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(R_0 + \sum_{d=1}^{\infty} \gamma^d \left(R_{t+d} - \frac{1}{\beta}\mathrm{KL}_{t+d} \right) - \frac{1}{\beta}\nabla_\theta \mathrm{KL}_t \right)\right] \tag{2.47}$$

$$= \mathbb{E}\left[\sum_{t=0}^{\infty} \nabla_\theta \log \pi_\theta(a_t|s_t)(Q_{\pi_\theta}(S_t, A_t) - V_{\pi_\theta}(S_t)) - \frac{1}{\beta}\nabla_\theta \mathrm{KL}_t\right] \tag{2.48}$$

, where (2.47) and (2.48) are expressed in terms of empirical returns and values functions (2.40) and (2.41). We can then approximate g_γ via n-step methods [13] or TD-λ methods [24] to truncate the return and lower the variance. Schulman et al. [23] chose to focus on the n-step methods and arrived at the following estimation of regularized policy gradient

$$\mathbb{E}\left[\nabla_\theta \log \pi_\theta(A_t|S_t)\delta_{t+n}\right] \tag{2.49}$$

, where δ_{t+n} is the n-step temporal difference as stated in equation (2.30). Then the corresponding optimization for the critic or the value function estimate is

$$L(V) = \mathbb{E}\left[\frac{1}{2}(V(S_t) - y_t)^2\right] \tag{2.50}$$

$$\text{where } y_t = \frac{1}{\beta}\mathrm{KL}_t + \sum_{d=0}^{n-1} \gamma^d \left(R_{t+d} - \frac{1}{\beta}\mathrm{KL}_{t+d} \right) + \gamma^n V(S_{t+n}) \tag{2.51}$$

and $V(s)$ is as stated previously in equation (2.45).

2.5.6 Equivalence Between SQL and AC With Regularized Rewards

It is important to note that the SQL framework and AC with regularized rewards are equivalent under the objective functions stated in sections 2.5.4 and 2.5.5. Specifically, Schulman et al. [23] states that defining V_θ and π_θ as the following of any parameterized Q-function Q_θ

$$V_\theta(s) = \frac{1}{\beta} \log \mathbb{E}[\exp(\beta Q_\theta(s, a))] \tag{2.52}$$

$$\pi_\theta(a|s) = \pi_0(a|s) \exp(\beta(Q_\theta(s, a) - V_\theta(s))) \tag{2.53}$$

then the gradient of a least-squares error from SQL equals the regularized policy gradient plus the gradient of a least-squares error of the value function. Defining $\Delta_t = \sum_{d=0}^{n-1} \gamma^d \delta_{t+d}$, then under the context of n-step SQL

$$\nabla_\theta \mathbb{E}\left[\frac{1}{2}\|Q_\theta(s_t, a_t) - y_t\|^2\right] \equiv \mathbb{E}\left[-\frac{1}{\beta}\nabla_\theta \log \pi_\theta(a_t|s_t)\Delta_t + \frac{1}{\beta^2}\nabla_\theta \mathrm{KL}_t + \nabla_\theta \frac{1}{2}\|V_\theta(s_t) - \hat{V}_t\|^2\right] \tag{2.54}$$

This relationship of equivalence will be used in the following sections under the context of multitask reinforcement learning through policy distillation. **SHOULD CLARIFY WHAT V HAT IS HERE**

2.5.7 Soft Actor-Critic (SAC)

We should also note that there exist an algorithm called Soft Actor-Critic by Haarnoja et al. [4] [5], and is entirely different than the regular actor-critic framework with regularized rewards. Specifically, rather than using KL divergence between two policies as mentioned in section 2.5.3, the definitions of V_π and Q_π are defined in terms of entropy

$$V_\pi(s) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t \Big(R(s_t, a_t) + \kappa H(\pi(\cdot|s))\Big) \big| s_0 = s\right] \tag{2.55}$$

$$Q_\pi(s, a) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t R(s_t, a_t) + \sum_{t\geq 1} \kappa H(\pi(\cdot|s)) \big| s_0 = s, a_0 = a\right] \tag{2.56}$$

where H is the entropy of an action under the policy π and κ[9] is the coefficient controlling the influence of entropy. Note that similar to definitions stated in section 2.5.3, Q_π does not include the entropy regularization term at $t = 0$ and thus resulting in the following relationship be tween Q and V

$$V_\pi(s) = \mathbb{E}\left[Q_\pi(s, a)\right] + \kappa H(\pi(\cdot|s)) \tag{2.57}$$

$$Q_\pi(s, a) = \mathbb{E}\left[R(s, a) + \gamma V_\pi(s')\right] \tag{2.58}$$

Furthermore, the original work featuring this algorithm also uses a replay buffer unlike conventional actor-critic algorithms.

[9]Note that the original papers [4] [5] used α instead of κ. We will replace it with κ as α is used in the multitask algorithms.

Under that context, SAC also differs from the aforementioned actor-critic with regularized rewards in that it learns the policy π_θ, two action value functions Q_{ϕ_1}, Q_{ϕ_2} and a state value function V_ψ. This is similar to the setup of actor-critic with separate parameters for Q and V as mentioned in equation (2.28), where a least square loss is used for V-function. But since there are 2 Q-functions, SAC uses a technique called clipped double-Q [5] [1]

$$L(V_\psi) = \mathbb{E}\left[\left(V_\psi(s) - \left(\min_{i=1,2} Q_{\phi_i}(s,\tilde{a}) - \kappa \log \pi_\theta(\tilde{a}|s)\right)\right)^2\right] \tag{2.59}$$

where action $\tilde{a}$ is sampled under the policy π_θ instead of taken from the replay buffer. As for learning the Q-functions, Haarnoja et al. [5] proposed the following

$$L(Q_{\phi_i}, D) = \mathbb{E}\left[\left(Q_{\phi_i}(s,a) - (r + \gamma(1-d)V_{\phi_{\text{targ}}}(s')\right)^2\right] \tag{2.60}$$

where d is a boolean indicating if the episode is done, and (r, s, a, s', d) is a transition sampled from the replay buffer D, and $V_{\phi_{\text{targ}}}$ is a target value network obtained via averaging network parameters over the course of training [4]. Lastly, SAC learn the policy by maximizing the following

$$\mathbb{E}\left[Q_\pi(s,a) - \kappa \log \pi(a|s)\right] \tag{2.61}$$

with the complete pseudocode attached at the appendix (fig A.3) as it is not the focus of this report.

2.6 Multitask Reinforcement Learning

2.6.1 Motivation

Since training separate agents for each task is inherently time consuming and expensive, multitask RL is an attractive approach as it presents an alternative solution where an agent leverages knowledge gained from different tasks to accelerate the learning process and/or apply transfer learning to unseen tasks.

Current reinforcement learning literature explores multitask-learning in four predominant

ways. The first approach is to learn multiple auxiliary tasks through pseudo-reward functions in addition to the original task in a continuous stream of data [9] [7]. The original motivation stems from the fact that some tasks may have very sparse reward signal, which makes learning extremely difficult as the value functions are not updated frequently. Work such as Jaderberg et al. [9], Sutton et al. [31] and others [10] [21] argue that constructing relevant auxiliary tasks that matches the long term goal of the main task improves the agent's performance as the frequency of reward signals increases.

The second approach is through transferring learning across multiple tasks such that the agent is immune to catastrophic forgetting [20], uses fewer data to learn compared to task-specific agents [12], and use prior knowledge obtained for learned features in previous tasks [20].

The third approach is to learn multiple tasks simultaneously in an online fashion with one single network [7]. Specifically, the network would sample transitions from harder tasks more often than the easy ones [26].

The fourth approach is distillation of task-specific policies into one general policy that governs the agent's behaviours [33] [22] [17]. This works by having iteratively optimizing the distilled policy and task-specific policies such that the distilled policy captures commonalities amongst tasks and transfer it to unseen ones.

In this report, we will focus on the fourth approach. Specifically, we will replicate some results demonstrated in Teh et al. [33], where distillation based multitask learning is experimented with different kinds of optimization frameworks.

2.6.2 Distral Architecture

Consider there are n tasks the agent needs to learn, and each task has a separate policy π_i assigned to it, plus one distilled policy π_0. The overarching goal is to regularize each task policy towards distilled using γ-discounted KL div with an entropy regularization term to encourage exploration

$$\mathbb{E}_{\pi_i}\left[\sum_{t\geq 0}\gamma^t \log\frac{\pi_i(a_t|s_t)}{\pi_0(a_t|s_t)} - \gamma^t \log \pi_i(a_t|s_t)\right]$$

Thus the resulting objective to maximize is:

$$J(\pi_0, \{\pi_i\}_{i=1}^n) = \sum_i \mathbb{E}_{\pi_i}\left[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t) - C_{\mathrm{KL}}\gamma^t \log \frac{\pi_i(a_t|s_t)}{\pi_0(a_t|s_t)} - C_{\mathrm{Ent}}\gamma^t \log \pi_i(a_t|s_t)\right]$$
$$= \sum_i \mathbb{E}_{\pi_i}\left[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t) + \frac{\gamma^t \alpha}{\beta} \log \pi_0(a_t|s_t) - \frac{\gamma^t}{\beta} \log \pi_i(a_t|s_t)\right] \quad (2.62)$$

, where $C_{\mathrm{KL}}, C_{\mathrm{Ent}} \geq 0$, $\alpha = \frac{C_{\mathrm{KL}}}{C_{\mathrm{KL}}+C_{\mathrm{Ent}}}$, $\beta = \frac{1}{C_{\mathrm{KL}}+C_{\mathrm{Ent}}}$.[10]

Specifically, we can think of the second term in (2.62) as a reward shaping term to encourage actions with high probability under π_0, and the third term as a mechanism to encourage exploration. In addition, α will be strictly less than 1 as $\log \pi_0(a_t|s_t)$ is interpreted as a soft prior onto π_i.

Two Distral architectures [33] were used to parameterize the task-specific and distilled policies. These two methods differ in the the logits of the task-specific policies π_i. The first method parameterizes each task-specific and distilled policies separately with the same feed-forward neural network, such that

$$\hat{\pi}_i(a_t|s_t) = \frac{\exp f_{\theta_i}(a_t|s_t)}{\sum_{a'} \exp f_{\theta_i}(a'_t|s_t)} \quad (2.63)$$
$$\hat{\pi}_0(a_t|s_t) = \frac{\exp h_{\theta_0}(a_t|s_t)}{\sum_{a'} \exp h_{\theta_0}(a'_t|s_t)} \quad (2.64)$$

, where the distilled policy is a the softmax of action preferences, and each task-specific policy is represented by the softmax of function approximation of the action values $Q_i(s_t, a_t) \approx f_{\theta_i}(a_t|s_t)$. This is based on the fact that policy gradient method is equivalent to Q-learning in the context of entropy-regularized rewards [37]. This method is called Distral 1 column, or Distral 1 col.

The second method parameterizes the distilled policy in the same manner as in (2.64),

[10]Note that the β coefficient is exactly the same as the one used in SQL and regularized AC.

with the augmentation of the task-specific policy in the following sense

$$\hat{\pi}_i(a_t|s_t) = \hat{\pi}_i(a_t|s_t)\exp\left(\beta\hat{A}_i(a_t|s_t)\right) \tag{2.65}$$

$$= \frac{\exp\left(\alpha h_{\theta_0}(a_t|s_t) + \beta f_{\theta_i}(a_t|s_t)\right)}{\sum_{a'}\exp\left(\alpha h_{\theta_0}(a'_t|s_t) + \beta f_{\theta_i}(a'_t|s_t)\right)} \tag{2.66}$$

where the soft advantages are estimated with θ_i and π_0

$$\begin{aligned}\hat{A}_i(a_t|s_t) &= \hat{Q}_i(a_t, s_t) - \hat{V}_i(s_t) \\ &= f_{\theta_i}(a_t|s_t) - \frac{1}{\beta}\log\sum_a \hat{\pi}_0^{\alpha}(a|s_t)\exp\left(\beta f_{\theta_i}(a|s_t)\right)\end{aligned} \tag{2.67}$$

as stated in [33][11] and proven in [18] [3] [16]. This is known as Distral 2 column or Distral 2 col, which is interpreted as each task-specific policy relying on the distilled policy plus the adaptation required to specialize to its task. The visual representation of the structures of Distral 1 and 2 col is shown in figure A.2 attached in the appendix.

With the two architectures for task-specific policies in mind, we can test for them via either a actor-critic (AC) with regularized rewards or soft Q-learning framework (SQL). These frameworks should result in similar performance as they are shown to be equivalent in [23]. In this report, we will use both SQL and AC framework for both distral algorithms.

2.6.3 Distral algorithms and SQL framework

Under the SQL framework, we will alternate between optimizing for task-specific and distilled policy. Specifically, we will base the framework on the DQN architecture as mentioned in [14] [15], where the action values are learned through sampling from a replay buffer.

At each step an episode, the task agent samples an action according to (2.66). We then do a n-step Temporal Difference (TD) learning by minimising the least-squares loss function as stated in equations 2.43, except the definition of the state-value estimate is replaced as a soft-Bellman equation (2.67) specific to the context of multitask learning.

After one step of SQL, the algorithm uses stochastic gradient descent on the maximum

[11] Note that the definition of $\hat{V}_i(s_t)$ differs from SQL's definition in that the reference policy π_0 is taken to power of α.

likelihood estimator, obtained from frequency of visits for each state action pair for each task [33]. This corresponds to terms dependent on π_0 in the original objective as stated in (2.62)

$$\frac{\alpha}{\beta} \sum_i \mathbb{E}_{\pi_i} \left[\sum_{t \geq 0} \gamma^t \log \pi_0(a_t|s_t) \right] \tag{2.68}$$

which have be shown [8] [19] [17] to lead to the distillation of π_i to π_0. The contents within each expectation can then be sampled and updated using stochastic gradient ascent. Below is the detailed pseudo-code describing the process for Distral 1 col under the SQL framework. The implementation for Distral 2 col is identical except in line 9, where the task-specific policy is derived from f_{θ_i} and h_{θ_0} via (2.66). Also, note that the use of bootstrapping from an old/target policy for stability from DQN [14] was not mentioned in Teh et al's work, therefore we are assuming that the target policy in the multitask-learning framework is simply the distilled policy.

Algorithm 1 Distral 1 column via SQL

Input: number of episodes N, list of envs, size of memory buffers
Initialize $\hat{\pi}_0$'s action preferences h_{θ_0} with random weights θ_0
Initialize $\hat{\pi}_i$'s action value estimate $f_{\theta_i} \approx Q_i$ as with random weights θ_i
Initialize memory buffers B_i for each env
Initialize state S for each env
Initialize $t = 0$ for each env to keep track of timestep
while min number of episodes for any task $< N$ **do**
    for each env i **do**
        Choose A from S using ϵ-greedy policy derived from f_{θ_i} ▷ Equation 2.64
        Take action A, observe R, S'
        Push transition (S, A, R, S', t) to B_i
        if len($B_i \geq$ min sampling length) **then** ▷ SQL w.r.t. task i
            Sample random minibatch transitions (S_i, A_i, R_i, S'_i, t) from B_i
            Set $y_i = \begin{cases} R_i & \text{if } S'_i = \text{Goal State} \\ R_i + \gamma/\beta \log \sum_a \hat{\pi}_0^{\alpha}(a|S_i) e^{\beta f_{\theta_i}} & \text{otherwise} \end{cases}$ ▷ Equation 2.67
            Perform a gradient descent step on $(y_i - f_{\theta_i}(S_i, A_i))^2$ w.r.t. parameters θ_i
        $S \leftarrow S'$
    $J(\theta_0) \leftarrow 0$
    for each env i **do** ▷ MLE w.r.t. π_0
        $m \leftarrow min\big(\text{batch size}, \text{len}(B_i)\big)$
        Sample m transitions (S_i, A_i, R_i, S'_i, t) from B_i
        $J(\theta_0) \leftarrow J(\theta_0) - \sum_i^m \left[\gamma^i \log \hat{\pi}_0(A_i|S_i)\right]$
    $J(\theta_0) \leftarrow \frac{\alpha}{\beta} J(\theta_0)$ ▷ Equation 2.68
    Perform a gradient descent step on $J(\theta_0)$ w.r.t. parameters θ_0

2.6.4 Distral algorithms and Regularized AC framework

An alternative implementation is via the actor-critic framework with regularized returns as discussed in section 2.5.5. Under this context, we would perform the conventional actor-critic optimization for each task-specific policy π_i, then optimize against the policy gradient of the distilled policy.

Specifically, for each task we rollout the MDP until the agent reaches the goal state or the maximum number of steps per episode (T) is reached. Then we optimize π_i by sampling

the partial derivative of the overall objective 2.62 w.r.t. to task i

$$\nabla_{\theta_i} J = \mathbb{E}_{\theta_i} \left[\sum_{t \geq 1} \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \left(\sum_{u \geq t} \gamma^u R_i^{\text{reg}}(a_u, s_u) \right) \right] \tag{2.69}$$

with $R_i^{\text{reg}}(a, s) = R_i(a, s) + \frac{\alpha}{\beta} \log \hat{\pi}_0(a|s) - \frac{1}{\beta} \log \hat{\pi}_i(a|s)$ as the regularized reward.

Furthermore, the score function trick as mentioned in A2C section in equation (2.27) also applies here. That is

$$\begin{aligned}
\mathbb{E}_{\theta_i} \left[\nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \right] &= \sum_{\pi_i} \pi_i(a_t|s_t) \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \\
&= \sum_{\pi_i} \nabla_{\theta_i} \hat{\pi}_i(a_t|s_t) \\
&= \nabla_{\theta_i} \sum_{\pi_i} \hat{\pi}_i(a_t|s_t) \\
&= 0
\end{aligned}$$

Therefore we can subtract the regularized rewards with a baseline to reduce gradient estimate's variance. Using the critic or state-value estimate as baselines and rewriting (2.69) yields

$$\begin{aligned}
\nabla_{\theta_i} J &= \mathbb{E}_{\theta_i} \left[\sum_{t \geq 1} \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \left(\sum_{u \geq t} \gamma^u R_i^{\text{reg}}(a_u, s_u) - \hat{V}_i(s_t) \right) \right] \\
&= \mathbb{E}_{\theta_i} \left[\sum_{t \geq 1} \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \left(Q_i(a_t, s_t) - \hat{V}_i(s_t) \right) \right] \\
&= \mathbb{E}_{\theta_i} \left[\sum_{t \geq 1} \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) A_i(a_t, s_t) \right] \qquad (2.70) \\
&= \mathbb{E}_{\theta_i} \left[\sum_{t \geq 1} \nabla_{\theta_i} \log \hat{\pi}_i(a_t|s_t) \delta_{i,t+n} \right] \qquad (2.71)
\end{aligned}$$

where $A_i(a_t, s_t)$ is the advantage function as stated in (2.67) and $\delta_{i,t+n}$ similar to the regularized TD error as stated in section 2.5.5, with the modified rewards R_i^{reg}. Analogous to the regularized AC, the task-specific state-value estimates will be optimized using a

least squares loss

$$L(V_i) = \mathbb{E}\left[\frac{1}{2}(\delta_{i,t+n})^2\right] = \mathbb{E}\left[\frac{1}{2}\left(y_{i,t} - \hat{V}_i(S_t)\right)^2\right] \tag{2.72}$$

$$\text{where } y_{i,t} = \begin{cases} R_i^{\text{reg}}(a_t, s_t) + \gamma \hat{V}(s_{t+1}) & \text{1-step TD-learning} \\ \sum_{d=0}^{n-1} \gamma^d R_i^{\text{reg}}(a_{t+d}, s_{t+d}) + \gamma^n \hat{V}_i(S_{t+n}) & \text{n-step TD-learning} \end{cases} \tag{2.73}$$

After one iteration of optimizing for each task under the actor-critic framework, we will update the distilled policy π_0 by sampling the partial derivative of the overall objective (2.62) w.r.t. to the parameters θ_0

$$\begin{aligned} \nabla_{\theta_0} J = & \sum_i \mathbb{E}_{\theta_i}\left[\sum_{t\geq 1} \nabla_{\theta_0} \log \hat{\pi}_i(a_t|s_t)\delta_{i,t}\right] \\ & + \frac{\alpha}{\beta}\sum_i \mathbb{E}_{\theta_i}\left[\sum_{t\geq 1}\gamma^t \sum_{a'_t}(\hat{\pi}_i(a'_t|s_t) - \hat{\pi}_0(a'_t|s_t))\nabla_{\theta_0} h_{\theta_0}(a'_t|s_t)\right] \end{aligned} \tag{2.74}$$

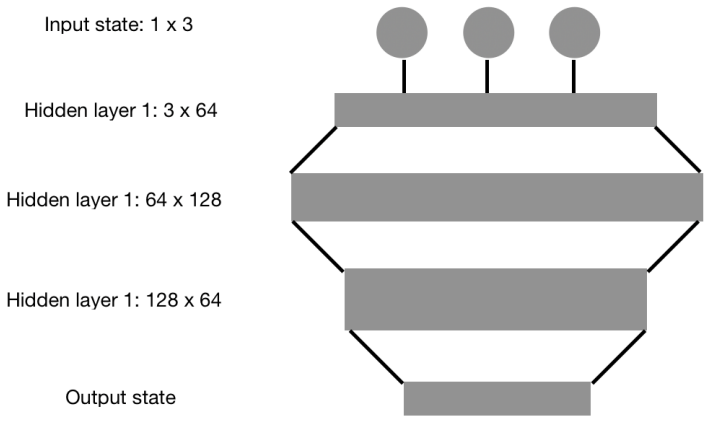
where the first term is identical to (2.71) and the second term is interpreted as the sum differences of the probabilities of $\hat{\pi}_i$ and $\hat{\pi}_0$. Below is the detailed pseudocode for Distral 1 col under the actor-critic framework. The implementation for Distral 2 col is identical except in line 11, where the task-specific policy is parameterized by f_{θ_i} and h_{θ_0} via (2.66) instead of Q_{θ_i}.

Algorithm 2 Distral 1 column via Actor-Critic framework

Input: number of episodes N, list of envs, size of memory buffers
Initialize $\hat{\pi}_0$'s action preferences h_{θ_0} with random weights θ_0
Initialize $\hat{\pi}_i$'s state value estimate $\hat{V}_{\theta_i}$ as with random weights θ_i
Initialize memory buffer B_i for each env to store the trajectory of current episode
Initialize state S for each env
Initialize $t = 0$ for each env to keep track of timestep
while number of episodes $< N$ **do**
    $J(\theta_i) \leftarrow 0$ for each env i
    for each env i **do**
        for $t = 1$ **to** T **do**
            Choose A_t from S_t using π_i derived from Q_{θ_i} ▷ Equation 2.63
            Take action A_t, observe R_t, S'_t
            Append transition (S_t, A_t, R_t, S'_t) to B_i
            if $S'_t =$ Goal state **then**
                $t \leftarrow t + 1$
                break
            else
                $S_t \leftarrow S'_t$
        Retrieve trajectory $(s_j, a_j, r_j, s'_j)_{j=1}^{t}$ from B_i ▷ A2C with regularised rewards
        for $i = 1$ to t **do**
            Set $\hat{V}_{\theta_i}(s_j) = \begin{cases} 0 & \text{if } s_j = \text{Goal State} \\ \text{Network output of } \hat{\pi}_i & \text{otherwise} \end{cases}$
            $\hat{Q}(a_j, s_j) \leftarrow r_j + \frac{\alpha}{\beta} \log \hat{\pi}_0(a_j s_j) - \frac{1}{\beta} \log \hat{\pi}_i(a_j|s_j) + \hat{V}_{\theta_i}(s_{j+1})$
            $A(a_j, s_j) \leftarrow \hat{Q}(a_j, s_j) - \hat{V}_{\theta_i}(s_j)$
        Perform a gradient descent step on $\sum_{j=1}^{t} \left(\hat{Q}(a_j, s_j) - \hat{V}_{\theta_i}(s_j)\right)^2$ w.r.t. parameters θ_i
        Perform a gradient ascent step on $\sum_{j=1}^{t} A(a_j, s_j) \log \hat{\pi}_i(a_j|s_j)$ w.r.t. parameters θ_i
        $J(\theta_i) \leftarrow \sum_{j=1}^{t} A(a_j, s_j) \log \hat{\pi}_i(a_j|s_j)$
    $J(\theta_0) \leftarrow 0$
    for each env i **do** ▷ perform gradient descent w.r.t. π_0
        $J(\theta_0) \leftarrow J(\theta_0) + J(\theta_i)$
        $J(\theta_0) \leftarrow J(\theta_0) + \frac{\alpha}{\beta} \left[\sum_{j=1}^{t} \gamma^j \sum_{a'_j} (\hat{\pi}_i(a'_j|s_j) - \hat{\pi}_0(a'_j|s_j)) h_{\theta_0}(a'_j|s_j)\right]$ ▷ Equation 2.74
    Perform a gradient descent step on $J(\theta_0)$ w.r.t. parameters θ_0
    Reset $J(\theta_i) = 0$, B_i for each env i

2.6.5 Network Architectures of Distral Algorithms

Since Teh et al. [33] did not specify the network architecture used in the function approximations for π_i and π_0's action value estimates and action preferences for the 2D maze experiments, we will use a simple 3 layer deep neural network. Specifically, all neural networks will have the following architecture



where the output vector could be a 1×4 vector of action preferences or 1×1 state value estimate depending on the use of the neural network. Note that we will also be using leaky-ReLU for all non-linear activation functions.

Chapter 3

Experimental Results and Extension on Original Work

3.1 Gridworld Environment

We focus on replicating the results observed using a simple 2D maze environment. Specifically, the original work [33] uses an environment consisting of two rooms connected by a corridor as shown in (3.1):

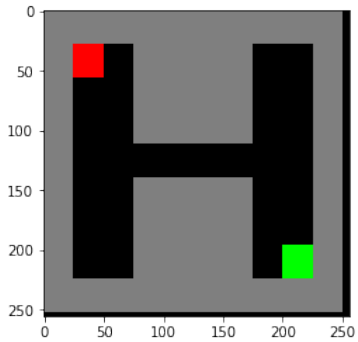


Figure 3.1: Environment 4 rendered from a Github's [2] implementation of the Distral paper

Furthermore, the original paper states that each state consists of the agent's current location, action and reward from the previous time step.

The original paper then compared the following algorithms:

- distral trained using only KL reg and optimization algorithm that alternates between soft Q-learning and policy distillation, with each soft Q-learning iteration using roll-out length 10 (n-step TD learning where n = 10)
- Distral agent to SQL that learns separate policy for each task.

We will focus on the comparing the Distral algorithms with SQL or regularized Actor-Critic depending on the framework of Distral algorithms. The complete list of available environments (fig A.1) is attached in the appendix. This report will mainly focus on environments 4, 5, 6, 7 and 8, as they are identical to the grid world presented in the original work.

3.2 Experiments under SQL framework

In this section we will compare the performance of Distral 1 col versus SQL and Distral 2 col versus SQL in different versions of the maze environment as demonstrated in [33]. Both Distral algorithms will be implemented under the SQL framework as demonstrated in the pseudocode for algorithm 1.

3.2.1 Reproducing results from original work

In the original work, Distral 2 cols via SQL was tested with $\alpha = 1$. That is, only the KL divergence penalty was included in the rewards. In addition, each SQL iteration uses a 10 step TD learning. The average rewards across tasks were then plotted as shown below

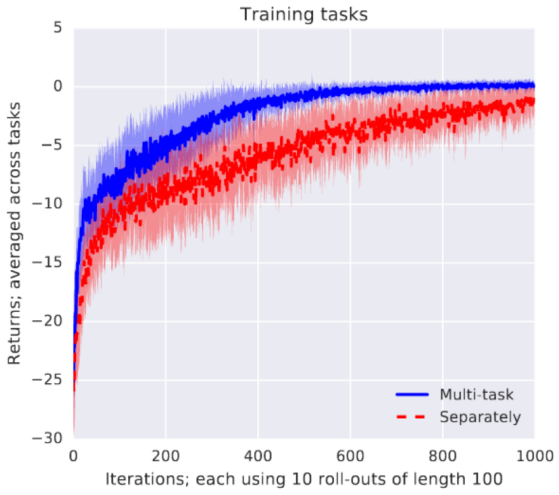
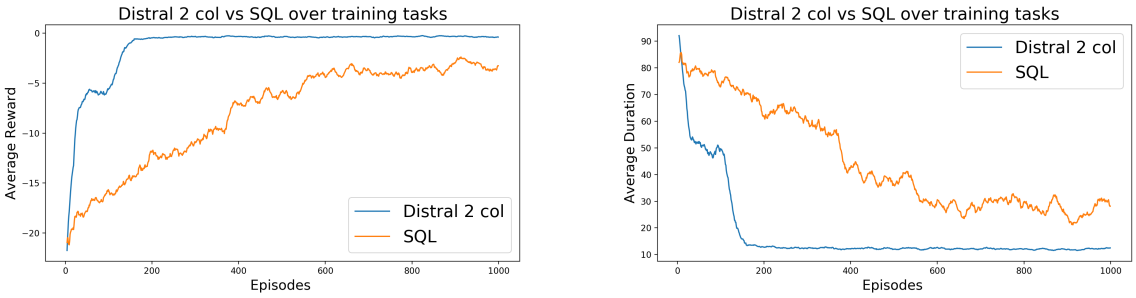


Figure 3.2: Distral 2 col's original results for GridWorld

As shown above, Teh et al. [33] demonstrates Distral's ability to learn faster than training SQL on each task separately. Since it was not mentioned in the original work, we will use a simple 3 layer feedforward neural network with ReLU as the activation functions as function approximations of the policies, state and action value estimates.

This result is compared to this report's replication of the experiment



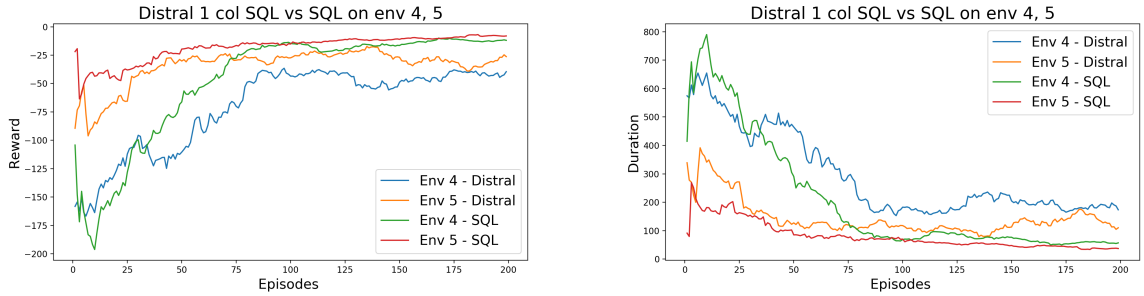
As shown above, the results matches the behaviour demonstrated in the original paper. Distral 2 col via SQL learns significantly faster and has lower averaged duration and higher rewards across 5 tasks in the 2-corridor environment. Note that performance may vary depending on the architecture of the neural networks used in to approximate policies, state and action value estimates.

3.3 Extended Experiments under SQL framework

3.3.1 Distral 1 col vs SQL

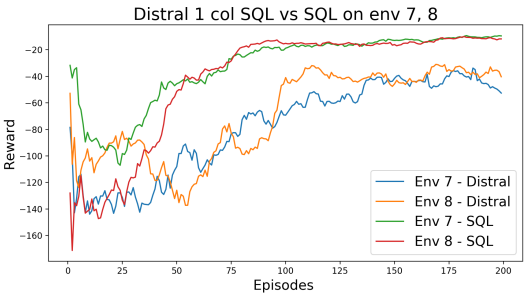
Although the results were replicated through 10-step TD learning and setting the maximum number of episodes to 100, we should note that it is substantially more time consuming to conduct the experiments with such settings. Therefore, we will set $n = 1$ instead of $n = 10$ and maximum number of steps per episode to 1000 instead of 100. Given these new settings, we will also reduce the number of episodes from 1000 to 200 as one would expect the algorithms to converge quicker if the maximum number of steps per episode is increased by 10 folds. Further, we will also set $\alpha = 0.8$ in order for π_0 to act as a soft prior on π_i. Finally, we will also set $\beta = 5$ to match the setting as mentioned in the original work [33]. Below are the results of Distral 1 col via the SQL framework, versus SQL trained separately on each task.

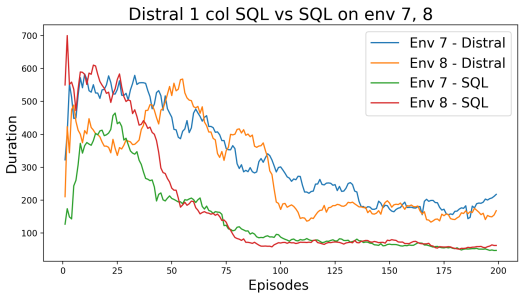
Environment 4 and 5



Here we see that SQL is performing better than Distral 1 col, converging earlier and to a lower duration per episode. We also see that Distral 1 col has converged to a local optimum at around 100 steps for environment 4 and 80 steps for environment 5.

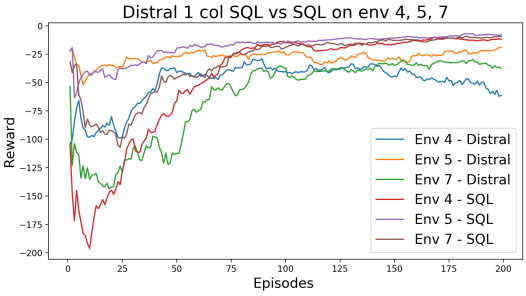
Environment 7 and 8

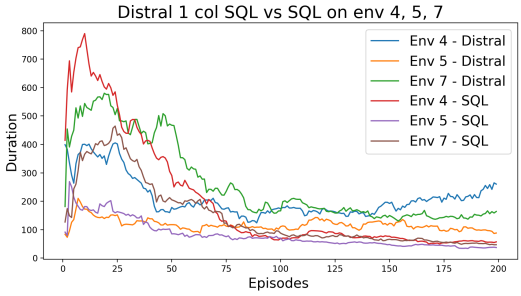




Here we see that SQL is once again performing better than Distral 1 col in terms of both convergence and the duration it takes to reach the goal state.

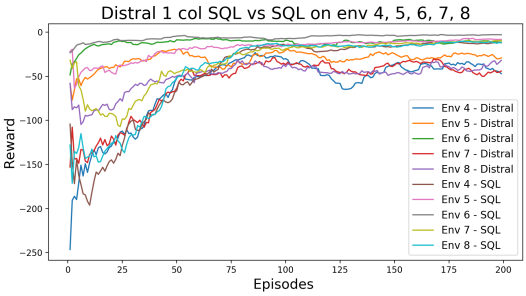
Environments 4, 5 and 7

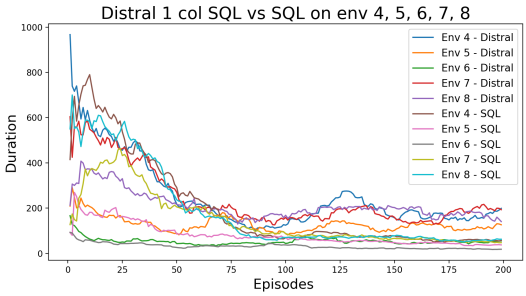




Here we see that Distral 1 col is performing significantly worst than SQL. This suggests that Distral 1 col's performance becomes highly volatile when learning more than 2 environments. We should also note that environment 4 and 7 are "hard" in a sense that the distance it takes for the agent to traverse from start to goal state is the furthest among all testing environments.

Environments 4, 5, 6, 7, 8



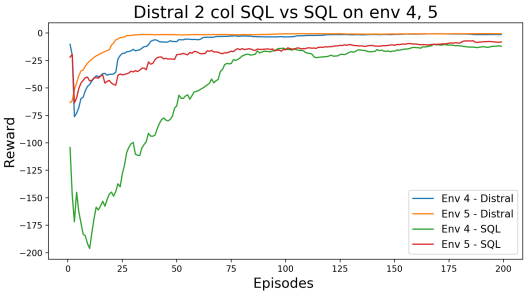


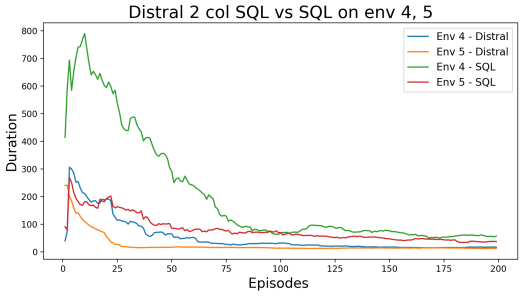
As shown above, the duration for most tasks are around 200 steps, indicating Distral 1 col did not solve most of the environments when it is tasked to solve all 5 environments at the same time.

3.3.2 Distral 2 col vs SQL

Here are the benchmark results for training Distral 2 col under SQL framework versus SQL, with the same settings as mentioned in the previous subsection. Again, the Distral algorithm is conducted under the SQL framework and compared against SQL trained separately on each task.

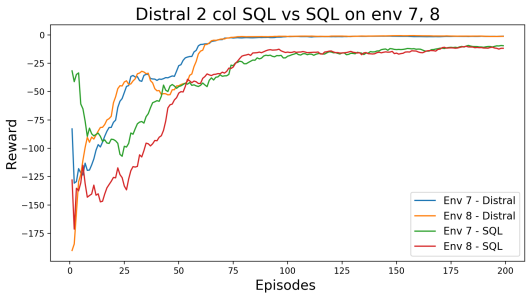
Environment 4 and 5

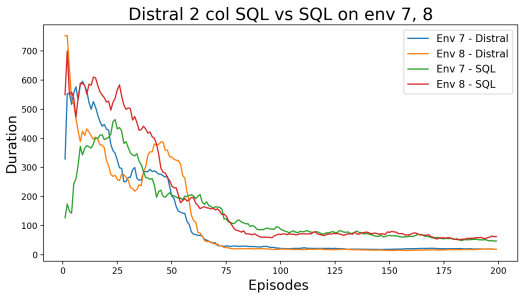




The diagrams above shows a significant difference in performance between Distral 1 col and 2 col. Here we see that Distral 2 col converged much faster than Distral 1 col did and significantly outperforms the SQL baseline.

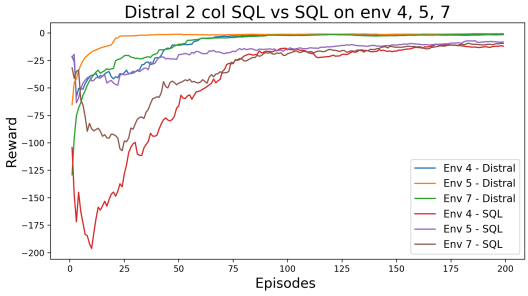
Environment 7 and 8

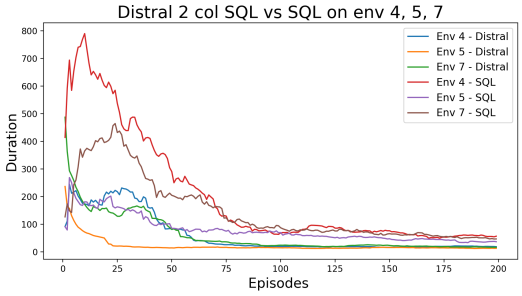




For environments 7 and 8 we see again that Distral 2 col converges at around episode 50, although performing worse than the SQL baseline initially.

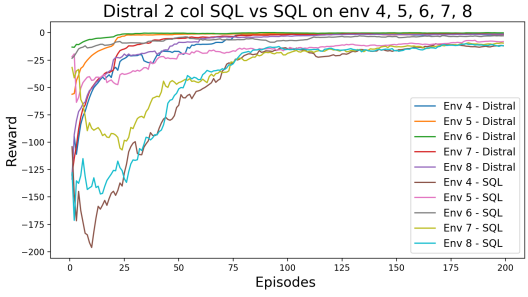
Environments 4, 5 and 7

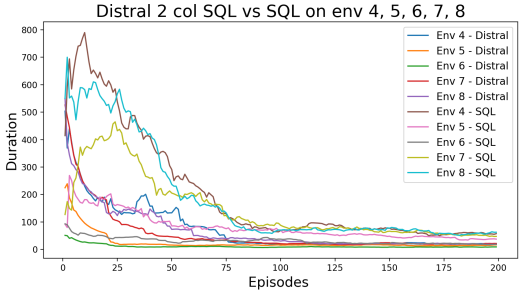




We see that for environments 4, 5 and 7 Distral 2 col converges quicker and performs better than the baseline for the entire 200 episodes.

Environments 4, 5, 6, 7, 8





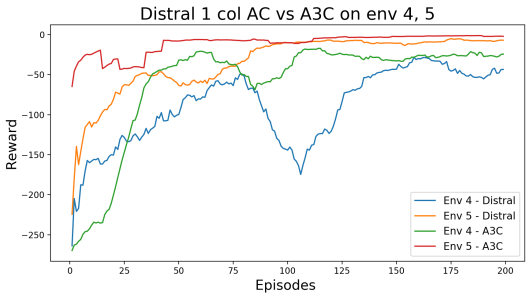
We also see that the algorithm performs worse when it is dealing with more environments. This can be further demonstrated in the following results, where Distral 2 col is trained simultaneously on environments 4, 5, 6, 7, and 8.

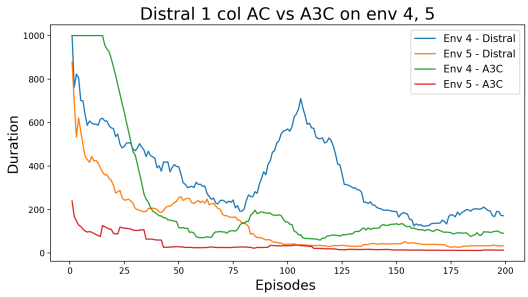
3.4 Experiments under regularized AC framework

In this section we will compare the performance of Distral 1 col and Distral 2 col versus A3C in same maze environments as the previous experiments with SQL. Both Distral algorithms will be implemented under the regularized AC framework as demonstrated in the pseudocode for algorithm 2.

3.4.1 Distral 1 col vs A3C

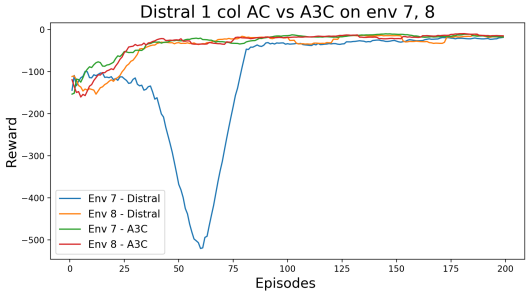
Environment 4 and 5

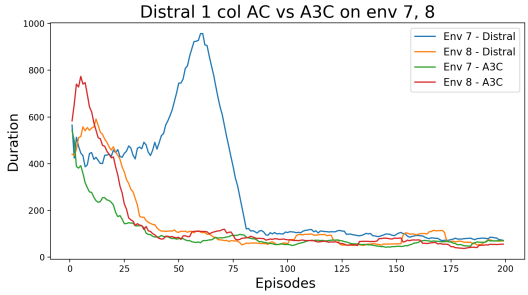




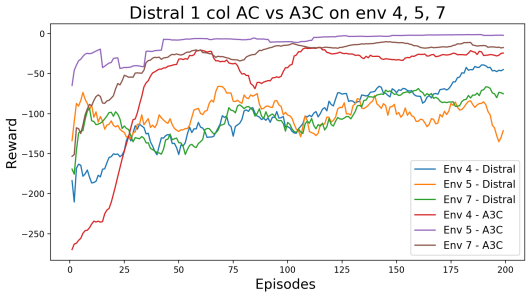
As shown above, Distral 1 col via regularized Actor Critic exhibits the similar behaviour to Distral 1 col implemented using SQL. For both algorithms, the duration for environment 4 plateaus around 200 steps.

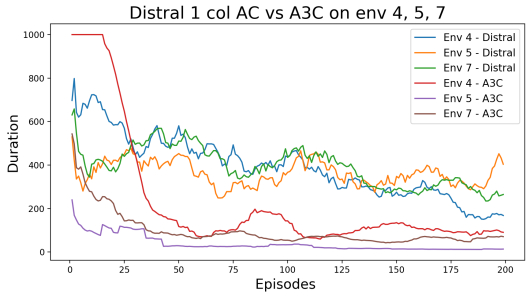
Environment 7 and 8





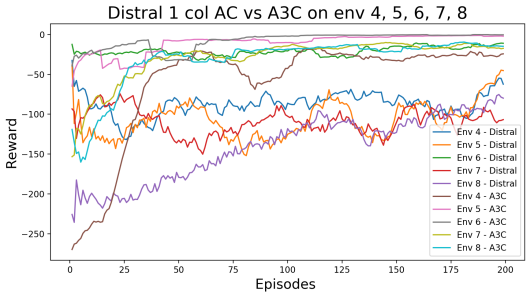
Environments 4, 5 and 7

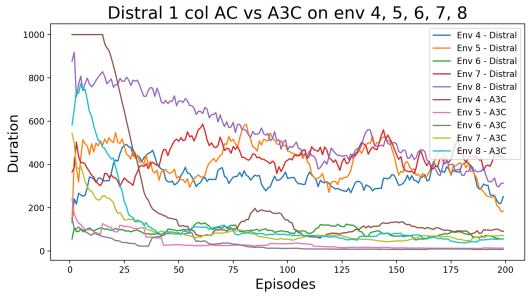




Here we observe that the Distral algorithm performs worse as it takes on more environments. Specifically, we see that it did not even manage to find the solution for enviroment 5, which is an easy environment compared to 5 and 7.

Environments 4, 5, 6, 7, 8

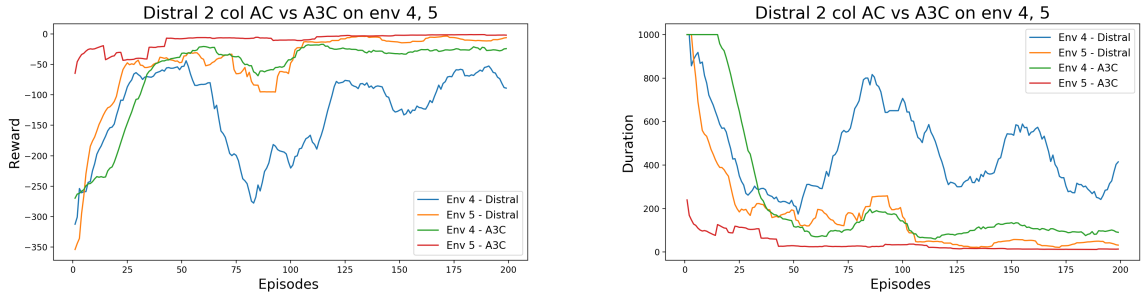




Here we see that the behaviour of the algorithm is consistent with previous observations. None of the environments were solved.

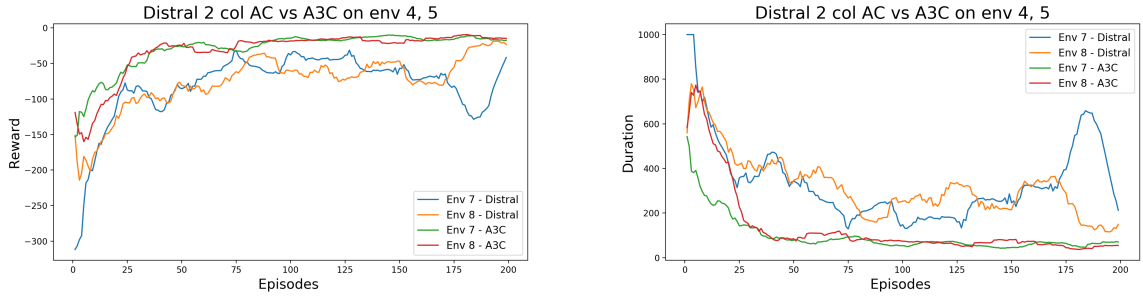
3.4.2 Distral 2 col vs A3C

Environment 4 and 5



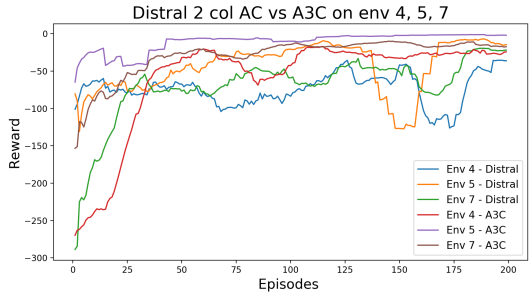
Here we see that Distral 2 col under AC framework exhibits similar behaviour compared to Distral 1 col under Ac. Specifically, we see that environment 4 did not converge and has consistently taken over 200 episodes to reach the terminal state. However, we should also note that Distral 2 col under AC managed to solve environment 5.

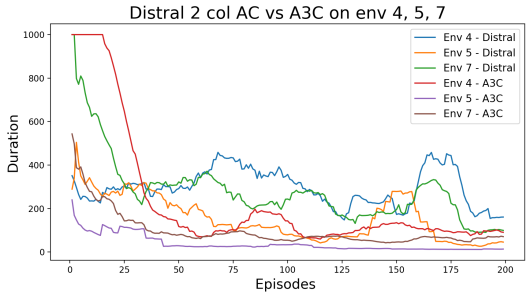
Environment 7 and 8



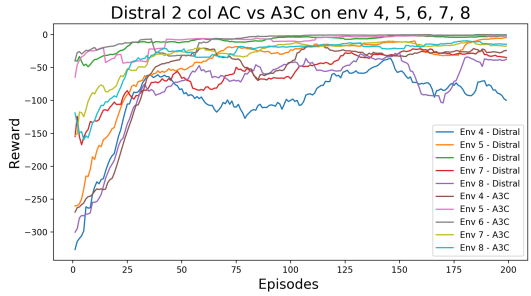
Here we see that the algorithm did not solve either of the environments. Again, the algorithm appears to have numerical stability issues consistent to the results for environments 4 and 5, and the behaviour of Distral 1 col AC.

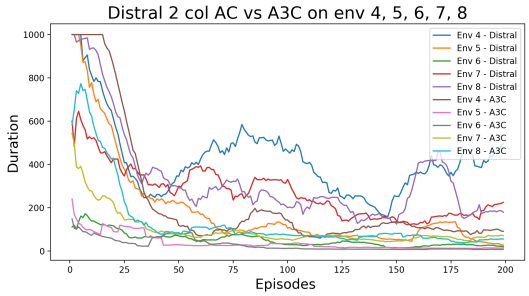
Environments 4, 5 and 7





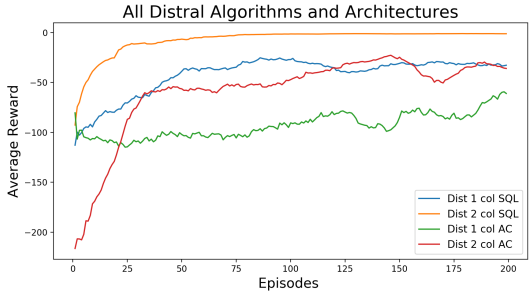
Environments 4, 5, 6, 7, 8

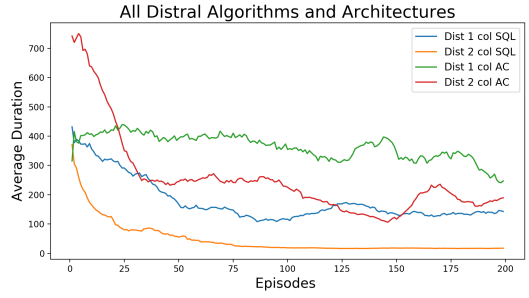




3.5 Distral 1 col vs Distral 2 col

To assess the overall performance of all Distral algorithms across different architectures, we plot the average rewards and duration across environments 4 to 8





As shown above, Distral 2 columns implemented via SQL is by far the most effective algorithm comparing to other variants. Several reasons can be considered in order to explain this phenomenon. One possible explanation for the difference in performance

between Distral 1 col and Distral 1 col in both frameworks is likely due to the fact that the architecture of Distral 2 col defines task-specific policy in terms of the distilled policy. First, recall that the task-specific policy defined in Distral 2 col is

$$\hat{\pi}_i(a_t|s_t) = \frac{\exp(\alpha h_{\theta_0}(a_t|s_t) + \beta f_{\theta_i}(a_t|s_t))}{\sum_{a'} \exp(\alpha h_{\theta_0}(a'_t|s_t) + \beta f_{\theta_i}(a'_t|s_t))}$$

whereas the task-specific policy in Distral 1 is simply softmax over $f_{\theta_i}(a_t|s_t)$, which gives Distral 1 col a significant disadvantage. Specifically, we can interpret the softmax over f_{θ_i} in terms of an Ising model, which is typically defined as

$$P(x|\beta) = \frac{1}{Z(\beta)} \exp[\beta \cdot (-E(x))] \tag{3.1}$$

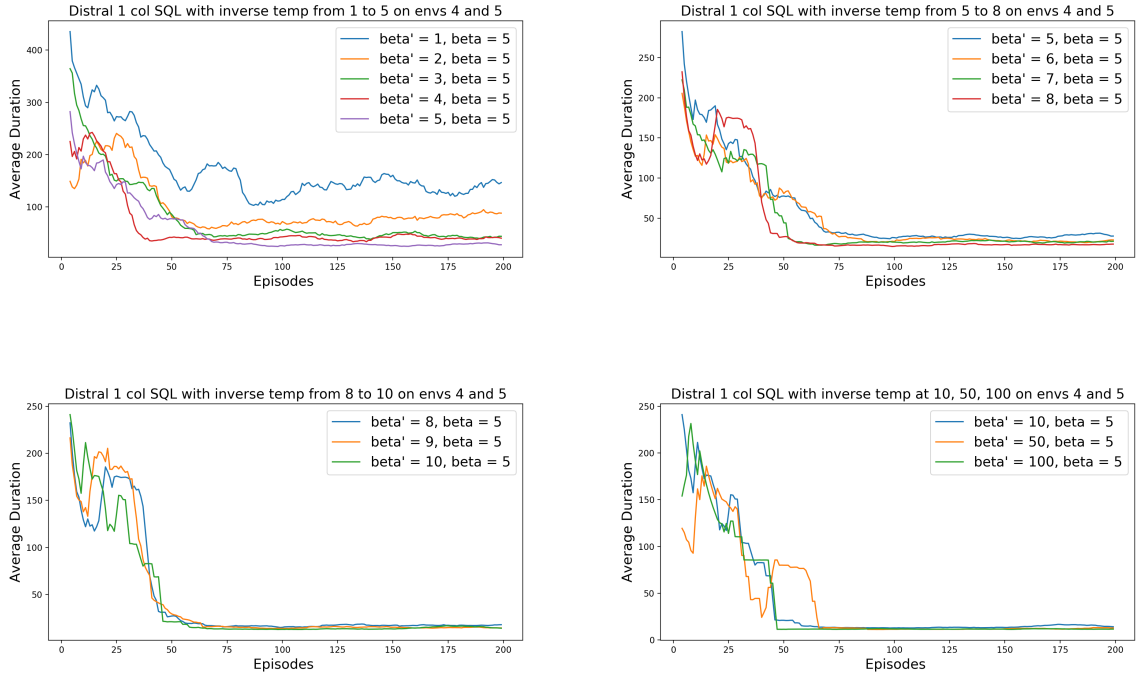
, where the LHS is interpreted as the probability of an atom at state x given the inverse temperature β, Z is the normaliser and $E(x)$ as the energy of the atom. β is known as the inverse temperature because it can be rewritten as $\beta = \frac{1}{k_B T}$, where k_B is the Boltzmann constant and T is the temperature of the model.

Interpreting the task-specific policy for Distral 1 col under the lens of an Ising model means the inverse temperature β is at 1

$$\hat{\pi}_i(a_t|s_t) = \frac{\exp(\beta f_{\theta_i}(a_t|s_t))}{\sum_{a'} \exp(\beta f_{\theta_i}(a'_t|s_t))} = \frac{\exp(f_{\theta_i}(a_t|s_t))}{\sum_{a'} \exp(f_{\theta_i}(a'_t|s_t))}$$

Whereas, the coefficients for Distral 2 col are $\alpha = 0.8$ and $\beta = 5$. This serves as an explanation to why Distral 1 col performed substantially worse than Distral 2 col. Specifically, we know from information theory and statistical mechanics [11] that as β decreases or T increases, the magnitude of $\exp[\beta \cdot (-E(x))] = \exp(\beta f_{\theta_i}(a_t|s_t))$ becomes roughly equal across all states and vice versa for large β or small T. This implies that setting $\beta = 1$ for Distral 1 col makes the task-specific probabilities all similar, which negatively disrupts the algorithm's efforts in solving the tasks at hand.

This can be verified empirically by comparing the performance of Distral 1 col via SQL on environments 4 and 5 for different settings of β during the agent's decision making process. Specifically, let β' be a new coefficient dedicated exclusively for $\pi_i(a_t|s_t)$ during action selection, whilst keeping the original β fixed at 5 in all other equations. Also, let this new algorithm variant be "Distral 1 col beta via SQL".

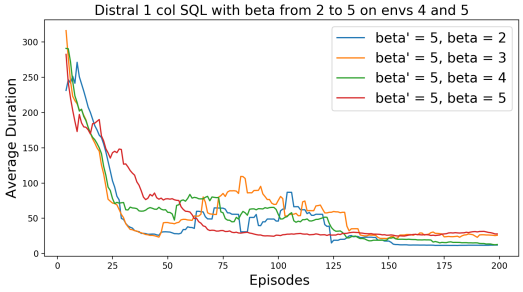


As we can see on the top-left figure, the effects of adjusting β' from 1 to 5 is very apparent. We can also see that the average duration at $\beta' = 4$ decreases faster than $\beta' = 5$ initially, but at around episode 60 setting $\beta' = 5$ results in lower average duration across tasks. Similar observation can be made on the top-right figure, where Distral 1 col's performance continued to improve as β' is increased from 5 to 8. However, we can also see on the bottom-left figure that increasing β' from 8 to 10 only provided minor improvements on the algorithm's performance. A similar argument can be made for the bottom-right figure, where $\beta' = 10, 50, 100$ is compared. Thus, we can conclude that a large enough β' (i.e. from 5 to 8) will allow the algorithm to make decisive choices instead of sampling from a uniform-like distribution.

We should also note that instead of understanding the effects of β' from the information theory's point of view, the original work [33] also mentioned that as $\beta' \to \infty$, the soft-Bellman updates as stated in equation (2.45) hardens to a `max` operator. The same behaviour applies to the task-specific policy $\pi_i(a_t|s_t)$, such that multiplying a large enough coefficient to f_{θ_i} results in a one-hot like distribution across actions. This also implies that for a very large β', the algorithm may not converge because of acting greedy too early.

Conversely, if we fix $\beta = 5$ and decrease the value of β, we discover that Distral 1 col

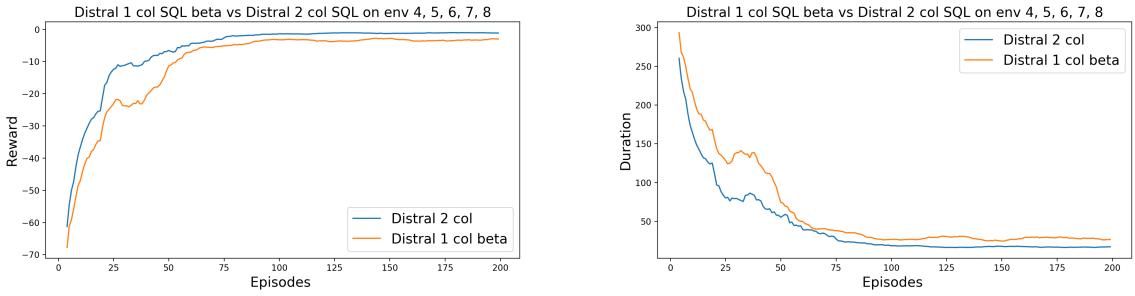
diverges. This is due to taking the exponent of large numbers when calculating $\hat{V}$ as defined in equation (2.67). For $\beta = 2, 3, 4$, we can avoid this numerical overflow by subtracting f_{θ_i} with its largest number. However, we should note that this technique could not prevent numeric overflow for setting $\beta = 1$. The resulting performance on environment 4 and 5 are as follows



As we can see, Distral 1 col manages to converge to a solution for environments 4 and 5 after fixing β' at 5, lowering β and the numerical stabilization. This indicates the inverse temperature β' plays a vital role in helping the algorithm make correct actions. We also see that lowering β makes the algorithm substantially less stable than setting $\beta' \geq 5$ and $\beta = 5$. Specifically, we can see that the average duration increases from episodes 50 to 130 before decreasing and converging to a final solution. Whereas, fixing $\beta = 5$ and using $\beta' \geq 5$ did not exhibit this behaviour.

3.5.1 Distral 1 col beta vs Distral 2 col under SQL framework

Given the discovery of the importance of multiplying the β' coefficient to f_{θ_i}, we should compare the performance of Distral 1 col beta and Distral 2 col under the SQL framework



As shown above, the average duration and reward for Distral 1 col beta rivals Distral 2 col under the SQL framework, confirming that we have discovered a modification on the

original algorithm that results in significant performance boost.

NEED TO EXPLAIN WHY AC FRAMEWORK DOESNT WORK WELL
- AC framework needs the entire episode to finish before an update, whereas for SQL it was sampling only from memory buffer, this means the updates for AC are highly correlated which is bad for learning? (need to check that)

- SQL framework updates π_0 by fitting it to a mixture of state-action distribution for each task under each task-specific policy. This is basically MLE, whereas in the AC framework we need to do policy gradient for both π_i and π_0 and each policy gradient introduces more variances to the update. We can see that as more task-specific policies are involved the performance of the algorithms under the AC framework gets substantially worse as shown in section **ac results**

- can also be due to the fact that SQL updates more frequently and thus gets to use to most "updated" policies

- can also be because AC framework needs to wait for all envs to finish one episode at each iteration before updating, whereas SQL doesn't need to

- don't know if this helps but the algorithms under AC has a much smaller learning rate than SQL. setting the learning rate at 0.001 will make AC algos diverge

Chapter 4

Conclusion and Suggestions on Further Work

In this thesis we have provided an in-depth explanation of the algorithms proposed in the original work through a detailed pseudo code for the Distral algorithms under both the SQL and regularized AC frameworks. We have also provided a chronological description of previous algorithms which led up to the multitask algorithms defined in Teh et al. Following that, we replicated the results of the original work for Distral 2 col via SQL on the 2D mazes. In addition, we extended the original experiments on the 2D mazes by testing both algorithms on the regularized AC frameworks and have found that their performance are consistently worse than the same algorithms implemented under the SQL framework. The discussion of the difference in performance between the algorithms implemented in SQL and regularized AC was not discussed in the original work. This lack of analysis is addressed in the results section, in which we discuss in-depth regarding the possible explanations for the algorithms' poor performance when implemented under regularized AC framework. Furthermore, we have discovered the importance of multiplying the inverse temperature coefficient β when calculating the task-specific policies. This discovery gave rise to a new variant called Distral 1 col beta, which has significantly better than Distral 1 col and has a similar performance to Distral 2 col under the SQL framework.

For future work we can implement the Distral algorithms in SQL framework, but integrate other concepts known to improve performance of DQN. Some examples include prioritized experience replay and double Q-learning, which were core concepts in the Rainbow algorithm [6] that were responsible for the performance boost. We can also try to implement

the Distral algorithms with the SAC framework instead of regularized policy gradient, as mentioned in [4]. Furthermore, under a less restrictive time constraint, we may be able to train the Distral algorithms on DeepMind lab's environments to further study the behaviours of the algorithms under more sophisticated environments. Lastly, another approach that would help the algorithms generalize to more tasks is to find an alternative way to circumvent the requirement of same action and state space across tasks. **A possible alternative could be ..**

in all the other experiments done by deepmind in the depmind lab, Distral 2 col consistently outperform Distral 1 col, so don't know why we need distral 1 col in the first place?

Bibliography

[1] Josh Achiam et al. Spinning up in deep rl, 2018–. [Online; accessed 9-8-2019].

[2] Alfredo de la Fuente and Anastasia Koloskova. Robust-Multitask-RL, 2019.

[3] Roy Fox, Ari Pakman, and Naftali Tishby. Taming the noise in reinforcement learning via soft updates. *arXiv preprint arXiv:1512.08562*, 2015.

[4] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[5] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[6] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Matteo Hessel, Hubert Soyer, Lasse Espeholt, Wojciech Czarnecki, Simon Schmitt, and Hado van Hasselt. Multi-task deep reinforcement learning with popart. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3796–3803, 2019.

[8] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[9] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[10] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[11] David JC MacKay and David JC Mac Kay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[12] Tom M Mitchell and Sebastian B Thrun. Explanation-based neural network learning for robot control. In *Advances in neural information processing systems*, pages 287–294, 1993.

[13] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] 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. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[16] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 2775–2785, 2017.

[17] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

[18] Konrad Rawlik, Marc Toussaint, and Sethu Vijayakumar. On stochastic optimal control and reinforcement learning by approximate inference. In *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.

[19] Andrei A Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation. *arXiv preprint arXiv:1511.06295*, 2015.

[20] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.

[21] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015.

[22] Simon Schmitt, Jonathan J Hudson, Augustin Zidek, Simon Osindero, Carl Doersch, Wojciech M Czarnecki, Joel Z Leibo, Heinrich Kuttler, Andrew Zisserman, Karen Simonyan, et al. Kickstarting deep reinforcement learning. *arXiv preprint arXiv:1803.03835*, 2018.

[23] John Schulman, Xi Chen, and Pieter Abbeel. Equivalence between policy gradients and soft q-learning. *arXiv preprint arXiv:1704.06440*, 2017.

[24] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[25] Guy Shani and Ronen I Brafman. Resolving perceptual aliasing in the presence of noisy sensors. In *Advances in Neural Information Processing Systems*, pages 1249–1256, 2005.

[26] Sahil Sharma, Ashutosh Jha, Parikshit Hegde, and Balaraman Ravindran. Learning to multi-task by active sampling. *arXiv preprint arXiv:1702.06053*, 2017.

[27] Sahil Sharma and Balaraman Ravindran. Online multi-task learning using active sampling. *arXiv preprint arXiv:1702.06053*, 2017.

[28] David Silver. Lecture notes in reinforcement learning, 2015.

[29] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[31] Richard S Sutton, Joseph Modayil, Michael Delp, Thomas Degris, Patrick M Pilarski, Adam White, and Doina Precup. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 761–768. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[32] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[33] Yee Teh, Victor Bapst, Wojciech M Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 4496–4506, 2017.

[34] Philip S Thomas and Emma Brunskill. Policy gradient methods for reinforcement learning with function approximation and action-dependent baselines. *arXiv preprint arXiv:1706.06643*, 2017.

[35] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[36] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[37] Xiaoqin Zhang, Yunfei Li, Huimin Ma, and Xiong Luo. Pretrain soft q-learning with imperfect demonstrations. *arXiv preprint arXiv:1905.03501*, 2019.

Appendix A

Figures, tables and pseudocode

Here are the gridworld environments taken from the **github project**. All environments except 1, 2, and 3 are identical to the environment demonstrated in the original work [33]:

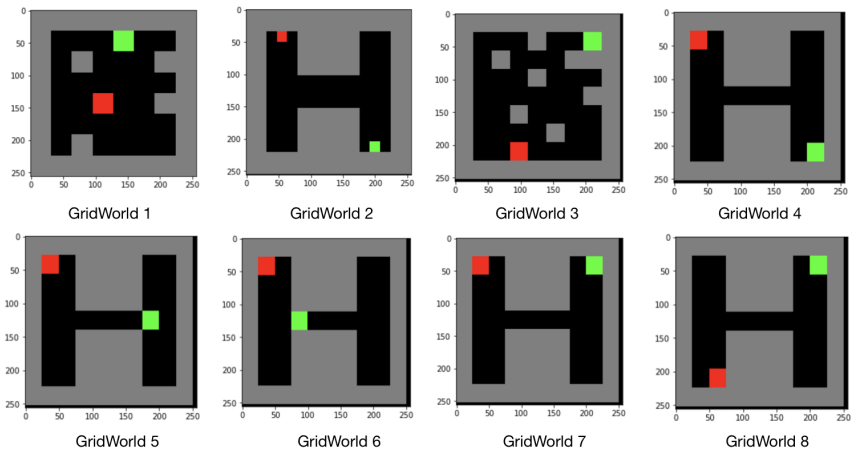


Figure A.1: GridWorlds taken from the **github project**

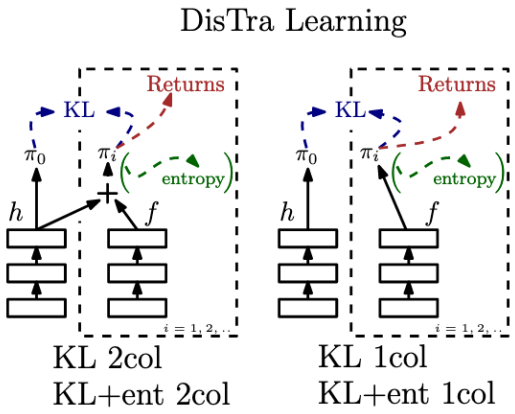


Figure A.2: Visual representation of differences between Distral 1 col vs Distral 2 col

Algorithm Soft Actor-Critic

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, V-function parameters ψ, empty replay buffer $\mathcal{D}$

Set target parameters equal to main parameters $\psi_{\text{targ}} \leftarrow \psi$

repeat

  Observe state s and select action $a \sim \pi_\theta(\cdot|s)$

  Execute a in the environment

  Observe next state s', reward r, and done signal d to indicate whether s' is terminal

  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$

  If s' is terminal, reset environment state.

  if it's time to update **then**

    for j in range(however many updates) **do**

      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$

      Compute targets for Q and V functions:

$$y_q(r, s', d) = r + \gamma(1 - d)V_{\psi_{\text{targ}}}(s')$$
$$y_v(s) = \min_{i=1,2} Q_{\phi_i}(s, \tilde{a}) - \alpha \log \pi_\theta(\tilde{a}|s), \qquad \tilde{a} \sim \pi_\theta(\cdot|s)$$

      Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi,i}(s, a) - y_q(r, s', d)\right)^2 \qquad \text{for } i = 1, 2$$

      Update V-function by one step of gradient descent using

$$\nabla_\psi \frac{1}{|B|} \sum_{s \in B} \left(V_\psi(s) - y_v(s)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(Q_{\phi,1}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta\left(\tilde{a}_\theta(s)|\, s\right)\Big),$$

      where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable wrt θ via the reparametrization trick.

      Update target value network with

$$\psi_{\text{targ}} \leftarrow \rho\psi_{\text{targ}} + (1 - \rho)\psi$$

    end for

  end if

until convergence

Figure A.3: Pseudocode [1] for Soft Actor-Critic in the context of continuous action spaces