



VICTORIA UNIVERSITY OF
WELLINGTON
TE HERENGA WAKA

School of Engineering and Computer Science Te Kura Mātai Pūkaha, Pūrorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Internet: office@ecs.vuw.ac.nz

Reinforcement Learning Part 3: Complete report of current Reinforcement algorithms and a new algorithm with evaluations

James Thompson

Supervisor: Dr Aaron Chen

June 22, 2025

Submitted in partial fulfilment of the requirements for
Master of Artificial Intelligence.

Abstract

This is submitted for the completion of AIML440 Directed individual study as part of the Master of Artificial Intelligence programme at Victoria University of Wellington. This report contains three things. Firstly an Introduction of reinforcement learning and basic RL algorithms. Secondly an introduction and explanation of modern and state of the art algorithms including experimental results. Lastly I propose a new algorithm Dynamic-Sunrise which is an extension of Sunrise to dynamically mutate and remove agents from the ensemble. Basic experiments are run and show the performance of the algorithm is similar to Sunrise but the diversity of the ensemble is maintained throughout learning. Further work would be needed to fully evaluate the potential and benefits of dynamic ensembles in the continuous control reinforcement learning setting.

Contents

1	Introduction	1
1.1	The reinforcement learning problem	1
1.2	Formalism	2
2	Algorithms	5
2.1	Traditional algorithms	5
2.1.1	Value Based Methods	5
2.1.2	Policy Based Methods	6
2.1.3	Challenges of traditional methods	7
2.2	Modern Deep learning algorithms	7
2.2.1	Deep Q Networks	7
2.2.2	Soft Actor-Critic	9
2.2.3	Proximal Polixy Optimizatin	10
2.2.4	Twin Delayed Deep Deterministic Policy Gradient	12
2.3	State of the Art Algorithms	13
2.3.1	Truncated Quantile Critics	14
2.3.2	Randomized Ensemble Double Q-Learning	15
2.3.3	Dropout Q-functions	16
2.3.4	Cross Q-learning	17
2.3.5	SUNRISE	19
3	Dynamic Sunrise Algorithm	21
3.1	Removal Check	21
3.2	Instantiation of new base learners	22
3.3	Basic algorithm	24
3.4	Hyper-parameters	24
4	Experiments	29
4.1	Introduction	29
4.2	Baseline Experiments	29
4.2.1	Sample Efficiency	29
4.2.2	Computational Efficiency	32
4.3	Ensemble diversity	33
4.3.1	Changing the ensemble	34
5	Conclusions	37
5.1	Further work	37
5.1.1	Different base learner	37
5.1.2	Hyperparameter tuning	37
5.1.3	Explore the effect of retraining the newly instantiated actors	37

5.1.4	Dynamic diversity measure	37
5.1.5	Add in the ability to add in new actors	38
5.1.6	Diversity in the objective	38
5.1.7	Different evaluation get_action method	38
5.1.8	Computational efficiency	38
5.2	Different algorithms	39
5.3	Conclusion	40
A		45
A.1	Experiment details	45
A.2	Experiment hyper-parameters	45
A.3	Source code	48
A.3.1	Experiment code	48
A.3.2	DSunrise code	49

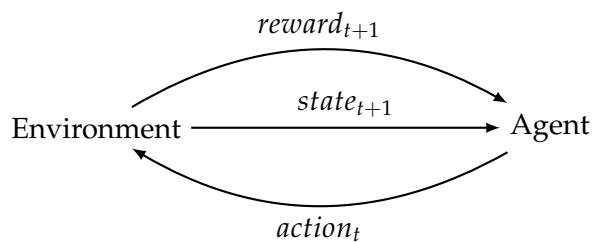
Chapter 1

Introduction

1.1 The reinforcement learning problem

Reinforcement learning is a framework that describes how an agent learns by interacting with the environment. The framework involves 2 entities the agent and environment. The agent is the only entity that we as designers have direct control of. We decide how its learns and decides on its actions. The environment is the context and situation that the agent is in. There are three important flows of information; the state of the environment to the agent, the action decision to the environment and lastly the reward to the agent. This can be best be understood in the figure below.

Figure 1.1 The flow of information between th environment and agent



The state is a representation of the information within the environment that the agent observes to make its decision on which action to take. The reward signal is treated as the final word on how good the previous state action was, the more reward the better, always. Lastly the action taken by the agent is sent to the environment and the environment will in turn return the next state and reward. This brings us to the reinforcement learning problem which can be framed as such "How does the agent decide which actions to take given the state such that it maximises the future cumulative reward". It is important that the agent maximises all *future cumulative* reward otherwise short term gains could be made to the sacrifice of larger long term gains. This idea has been formalised by Richard Sutton as the reward hypothesis

"That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)." [30]

1.2 Formalism

The reinforcement problem can be formalised as a Markov Decision Process (MDP). The MDP is a collection of states, actions and rewards along with a transition function which states the probability of the next reward and state given a state and reward. This makes the MDP a 4-tuple $(S, \mathcal{A}, \mathcal{P}, R)$ where S is the set of states, $\mathcal{A}(s)$ is the set of actions that can be taken from state s , \mathcal{P} is the transition function and $R \subset \mathbb{R}$ is the set of rewards. The transition function is defined as:

$$\mathcal{P}(s', r, s, a) = \Pr \{ S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a \} \quad (1.1)$$

Where $s, s' \in S, a \in \mathcal{A}(s)$ and $r \in R$.

This transition function completely characterises the dynamics of the environment. The abstraction of the environment to a MDP is widely applicable and serves as the basis for much of reinforcement learning. We can see here that transition function only looks at the previous state. It can do this because we assume that the state representation has the *Markov property* [30]. The Markov property states that including previous states in the conditional wont change the probability of next state reward tuple. In other words $\mathcal{P}(s_{t+1}, r_{t+1}, s_t, a_t) = \mathcal{P}(s_{t+1}, r_{t+1}, s_t, a_t, s_{t-1}, s_{t-2}, \dots, s_0)$. Therefore a state with the Markov property is a sufficient representation of the history of the agent-environment interaction. In some problems the agent only partially observes the state meaning the Markov property is not satisfied. This is called a partially observable MDP (POMDP) and is a more complex problem to solve which wont be addressed in this report.

The agent interacts with the MDP to produce a trajectory of states, actions and rewards. Because the agent will not know the transition function 1.1 it will have to learn about the MDP from the information in a trajectory 1.2.

$$S_1, A_1, R_2, \dots, S_n, A_n, R_{n+1}, \dots \quad (1.2)$$

In this trajectory 1.2 we have an infinite sequence as this is an example from a continuing MDP that has no end. Alternatively you could have episodic MDPs that have a start and end with a terminal state. Episodic MDPs require practical considerations in implementing algorithms. However for most of the theory it make no difference as you can think of a episode MDP as a continuing MPD with a state that transitions to itself and gives reward 0.

"The cumulative sum of a received scalar signal" part of the reward hypothesis 1.1 can be formalised to be the return G .

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=t}^{\infty} \gamma^{k-t} R_k \quad (1.3)$$

γ is called the discounting factor and weights future rewards to have less effect on the return. It is usually added for two reasons. Firstly because it makes the return finite which simplifies the mathematics. Secondly is due to the very natural intuition that the future is less predictable than the present, thus more distance rewards should have less weight as they are less certain.

A agent will use policy π which provides the probability of taking action a given state s . This policy could be deterministic or stochastic. To evaluate how good a policy is or how much reward we can expect at a state we need a function to tell us. This is called the value function and it is a expectation of the future rewards.

$$v_{\pi}(s) = \mathbb{E}[G_t | S_t = s] \quad (1.4)$$

The state value function depends on the state, policy and discounting factor γ which is used by the return. In a similar vain to the state value function we have the action-value function which is the expected future reward given a state and action taken.

$$q_\pi(s, a) = \mathbb{E}[G_t | S_t = s, A_t = a] \quad (1.5)$$

We can understand how to compute the expectation of the value function by looking at the Bellman equation [30]. The Bellman equation is based off the notion that the value of a situation should be the immediate reward you get plus the value of the situation you end up in. This can be written for the state value function as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) [r + \gamma v_\pi(s')] \quad (1.6)$$

$\pi(a|s)$ is the probability of taking action a given state s and $p(s', r|s, a)$ is the transition function. This equation can be solved for v_π by iterating over all states and actions until the value function converges. This is called the iterative policy evaluation and is a dynamic programming technique [5].

The maximising part of the reinforcement problem can be solved by the finding optimal actions. We can define both $v_* = \max_\pi v_\pi(s)$ and $q_* = \max_\pi q_\pi(s, a)$ as the optimal value given optimal actions afterwards. If we can find q_* then the problem is solved as we could make a policy π_* that is greedy with respect to q_* . Two things stop this from being so simple. Firstly is that in many real problems we don't know the true transition function, therefore we either have to learn/approximate or learn the value/policies in a way that doesn't involve the transition function. Secondly is the computational complexity of iterating over all states and actions as well as all possible policies. This is called the curse of dimensionality and is a major problem in reinforcement learning. If we try to learn and transition function and come up with a model of the environment then we are using model based methods, alternatively we ignore the model and learn the value or policy directly and these are called model free methods. In the rest of the report we will be focusing on model free methods.

The form of explicitly learning a value function then implicitly getting actions from it is called value based methods. Alternatively you can learn the policy directly with policy based methods. The effectiveness of these methods depend heavily on how easy the value function or policy is to learn.

Chapter 2

Algorithms

In the real world we don't get access to the transition function of the MDP. The agents only method to learn is through interacting with the environment. From this interaction we get a sample of the environment and must learn from it. Therefore stochastic learning methods must be used. Two learning ideas are core for many reinforcement learning algorithms. These are Monte Carlo [30] and Temporal Difference learning [28] [29].

2.1 Traditional algorithms

2.1.1 Value Based Methods

As mentioned earlier value based method focus on learning a value function and from that function we can derive a policy. The action value function is more powerful as simply knowing the value of a state is not enough to make a decision. The action value function is defined as the expected return given a state and action. Thus all you need to do is take the action that gives you the maximum Q value.

Monte Carlo learning works by estimating the value function as being the average return from that state. A basic algorithm would look like this.

Algorithm 1 Monte Carlo Control with Exploring Starts

```
1: Arbitrarily initialize  $\pi(s) \in \mathcal{A}(s)$  and  $Q(s, a) \in \mathbb{R}$ 
2:  $\text{Returns}(S, A) \leftarrow$  empty list
3: for each episode do
4:   Arbitrarily choose  $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}$ 
5:   Generate an episode by following  $\pi$ 
6:    $G \leftarrow 0$ 
7:   for each step of the episode  $t = T - 1, T - 2, \dots, 0$  do
8:      $G \leftarrow \gamma G + R_{t+1}$ 
9:     if  $(S_t, A_t)$  does not appear earlier in the episode then
10:       Append  $G$  to  $\text{Returns}(S_t, A_t)$ 
11:        $Q(S_t, A_t) \leftarrow \text{average}(\text{Returns}(S_t, A_t))$ 
12:        $\pi(S_t) \leftarrow \arg \max_a Q(S_t, a)$ 
13:     end if
14:   end for
15: end for
```

Because we need a well defined return Monte Carlo methods only work on episodic tasks. It will only update the states that it actually visits. This gives us the option of just

exploring the state space area we are interested in learning about.

Temporal difference learning works different incrementally updating the value of a state with the received reward and the value of the next state 2.1. As the update is using its own estimate it is *bootstrapping*. Because of bootstrapping it can work with both episodic and continuing environments.

$$V(S_t) = R_{t+1} + \gamma V(S_{t+1}) \quad (2.1)$$

Below is a basic implementation of TD learning for a continuing environment, however it is easy to change it to episodic by changing the loop to be over episodes.

Algorithm 2 TD(0)

```

1: Initialize  $Q(s, a)$  arbitrarily and set  $Q(\text{terminal-state}) = 0$ 
2: Initialize  $S$ 
3: while not converged do
4:   Take action  $A$ , observe  $R, S'$ 
5:   Choose  $A' \in A(S')$  using policy derived from  $Q$ 
6:    $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$ 
7:    $S \leftarrow S'; A \leftarrow A'$ 
8: end while
```

This particular update is known as TD(0) because it only looks one step into the future. However it can be generalised to be TD(n). One can see that if we had TD(∞) we would be back at Monte Carlo if working in episodic tasks.

The two methods of TD(n) and Monte carlo can be to get the best of both worlds using TD(λ) methods [29].

2.1.2 Policy Based Methods

A different way of approaching the problem is by learning the policy directly without a learning a concept of how good a state is. The most common policy based method is called policy gradient methods [31]. To use policy gradient methods you need to parametrize the policy π in any way as long as $\pi(a|s, \theta)$ is differentiable with respect to θ .

The first of these policy gradient methods was the Reinforce algorithm [41]. It is a Monte Carlo policy gradient method. The updates are made by moving the policy in the direction of the gradient of the log probability of the action taken. This works because it is proportional to the return which is given to use by the policy gradient theorem [31].

Algorithm 3 REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Require: A differentiable policy parameterization $\pi(a|s, \theta)$

Require: Step size $\alpha > 0$

```

1: Initialize policy parameter  $\theta \in \mathbb{R}^d$  (e.g., to 0)
2: loop
3:   Generate an episode  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$ , following  $\pi(\cdot | \cdot, \theta)$ 
4:   for  $t = 0, 1, \dots, T-1$  do
5:      $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$ 
6:      $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t | S_t, \theta)$ 
7:   end for
8: end loop
```

2.1.3 Challenges of traditional methods

There are main two issues with the simple methods of TD and Monte Carlo mentioned above. This is scalability and exploration.

Firstly with scalability we are storing the action value of a particular state in a large lookup table. This will immediately become a problem when dealing with large state or action spaces. Most real world application have tremendously large state/action spaces. Furthermore it is easy to imagine how most of the states have overlapping information and are actually really quite similar. Therefore instead of learning Q directly we can try and learn an approximator \hat{Q} .

Second problem is exploration. As we learn we are finding better and better actions. These better action are taken which will form our trajectory. What can happen here is that we miss large chunks of the state space. To solve this we can instead use a sub optimal policy that deliberately takes exploratory actions. ϵ -greedy is a good example of this as it will take a random action with probability ϵ and an optimal action the rest of the time. Alternatively you can use a different exploration policy to interact with the environment while you are learning the optimal. This method of learning from experience using a different policy is called off-policy learning. The previous methods we looked at are all on-policy.

2.2 Modern Deep learning algorithms

One of the problems with the above tabular methods is that the policy and value functions are stored explicitly in effectively a large table lookup. This means that for every state and or action there needs to be a new entry. This is problematic as the state and action space of the problems get very large. To resolve this problem we can use function approximators and the best ones we have are neural networks [12]. This allows for the combination of the deep learning ideas within the reinforcement learning framework to give rise to deep reinforcement learning.

2.2.1 Deep Q Networks

One can take the TD(0) learning algorithm and apply the notion of off-policy learning we get SARSAMAX otherwise known as Q-Learning [39] [40]. It works almost the same except the action value function is updated using the best possible action taken at the next state.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right) \quad (2.2)$$

As with TD(0) this fails when faced with a sufficiently large state and/or action space. Therefore we apply a neural network as an approximator \hat{Q} for Q . The algorithm we get is called Deep-Q-Learning [21][22]. It was applied very successfully to match and exceed human level performance on a large variety of Atari 2600 games (Space invaders, pong etc).

Here is the algorithm that Mnih et al used in their papers [21] [22], with some notational differences to make it more explicit.

Algorithm 4 Deep Q-Learning (DQN)

Require: Number of episodes M , replay memory capacity N , minibatch size K , discount factor γ , learning rate α , update frequency C , exploration probability ϵ . A neural network function approximator Q

- 1: Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
- 2: Initialize action-value function Q_θ with random weights θ
- 3: Initialize target action-value function \hat{Q}_{θ^-} with weights $\theta^- \leftarrow 0$
- 4: **for** episode in range(1, M) **do**
- 5: Initialize sequence $s_1 \leftarrow \{x_1\}$ and preprocessed sequence $\phi_1 \leftarrow \phi(s_1)$
- 6: **for** each step t in episode **do**
- 7: With probability ϵ , select a random action a_t
- 8: Otherwise, select $a_t \leftarrow \arg \max_a Q(\phi(s_t), a; \theta)$
- 9: Observe reward r_t and next state x_{t+1}
- 10: Set $s_{t+1} \leftarrow (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} \leftarrow \phi(s_{t+1})$
- 11: Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}
- 12: Sample random minibatch B of size K of transitions from \mathcal{D}
- 13: **for** each transition $(\phi_j, a_j, r_j, \phi_{j+1})$ in B **do**
- 14: $y_j \leftarrow \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases}$
- 15: Compute loss $L_j \leftarrow (y_j - Q(\phi_j, a_j; \theta))^2$
- 16: **end for**
- 17: Perform gradient descent step on the average loss: $\theta \leftarrow \theta - \alpha \nabla_\theta \frac{1}{|B|} \sum_j L_j$ ▷
Alternative gradient steps could be used here (e.g Adam was used in the paper)
- 18: **if** $t \bmod C == 0$ **then**
- 19: Update target network: $\theta^- \leftarrow \theta$
- 20: **end if**
- 21: **end for**

There are more features of this algorithm that are different than traditional Q-learning. These are important to solve problems the arise when you add a neural network approximator into the learning process of Q-learning.

The first of these ideas is called experience replay [19], which is collecting your experience in a replay memory D . Then to learn you can loop through your experience and extract as much as you can from your previous experiences. The idea in Deep Q-learning is that at each time step rather than updating \hat{Q} with your current experience you update it using a sampled transition from D . This helps with an important problem which is that stochastic gradient descent assumes independent and identically distributed data (**i.i.d**). As experience is collected each transition will be dependent on the previous transition and will affect the next transition thus the independent assumption is not held. Thus randomly sampling from D will give you a independent data set. This also makes DQN a off policy learning method

The second assumption is that the training data is identically distributed. Gradient descent is working towards the target γ . The problem is that the target involves our current prediction, so as we learn a better prediction our target will move. This results in a chase which decreases learning efficiency. The solution to this as seen in the algorithm is fixing \hat{Q} for a fixed number of steps C . It uses \hat{Q} in the target γ_j but then updates Q every C updates.

This gives the network a reasonable opportunity to tend towards the target γ_j while still using a recent estimate of the action value.

Lastly is the concept of pre-processing the state to speed up the learning process. One can decrease the computational complexity of the neural network by simplifying the state using a transformation that in theory doesn't lose the meaningful information. Minh et al used this in their 2013 paper to make the video input; smaller, conform to the ratio their model needed and making it gray scale. We can see that this pre-processing step is the only example of domain specific knowledge that would be needed for an implementation of the algorithm to a task.

2.2.2 Soft Actor-Critic

The DQN method discussed above, along with TD and MC learning, are all value-based methods. That is, they learn to evaluate how good a current situation is (whether that be a state or state-action), and then a policy can be generated by maximizing over the value function. Another approach is to directly learn the policy function. A combination of these ideas leads to the actor-critic framework.

In an actor-critic framework, there is a policy that controls how the agent acts, π , as well as a value function that evaluates how good the action taken was. A good way to update the policy is based on whether the result was better or worse than what the critic expected. Using the TD error of the critic for this is called A2C [20].

However, there are still challenges in expanding A2C to high-dimensional continuous control tasks. Many current methods are difficult to stabilize and require carefully tuned hyperparameters. To address this, several improvements can be introduced to the actor-critic framework, resulting in Soft Actor-Critic (SAC) [9].

First, similar to DQN [21, 22], making it off-policy allows for much better sampling efficiency as more information is extracted from experience. More importantly, SAC employs entropy maximization. Traditional RL agents aim to maximize the expected sum of rewards. However, the maximum entropy RL framework modifies this objective to maximize both the expected reward and the entropy of the policy, leading to the following objective function:

$$J(\pi) = \sum_{t=0}^T \mathbb{E}_{(s_t, a_t) \sim \rho_\pi} [r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot | s_t))] \quad (2.3)$$

This entropy term encourages exploration, with the temperature parameter α determining the strength of this encouragement. Initially a hyperparameter in [9], the SAC algorithm was later modified to learn and adjust α throughout training [10]. This not only improves efficiency—since choosing an optimal temperature is task-dependent and non-trivial [10]—but also allows the policy to explore uncertain regions while being more exploitative in familiar areas.

The final SAC algorithm [10] modified for clarity is shown below:

Algorithm 5 Soft Actor-Critic Algorithm

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ, λ_π and λ_Q , update frequency C , target smoothing coefficient τ , Updates to Data ratio UTD, environemt steps S , neural network function approximator Q , neural network function approximator π and initial entropy coefficient α .

- 1: Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
- 2: Initialize policy π_ϕ with random weights ϕ
- 3: Initialize both action-value function Q_{θ_i} with random weights θ_i
- 4: Initialize $\bar{\theta}_i \leftarrow \theta_i$ for $i \in \{1, 2\}$
- 5: **repeat**
- 6: **for** S steps **do**
- 7: select action: $a_t \sim \pi_\phi(a_t | s_t)$
- 8: Observe reward r_t and next state s_{t+1}
- 9: Store transition (s_t, a_t, r_t, s_{t+1}) in \mathcal{D}
- 10: **end for**
- 11: **for** UTD times **do**
- 12: Update Q-functions: $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
- 13: Update policy: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
- 14: Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
- 15: Update target Q-networks: $\bar{\theta}_i \leftarrow \tau \theta_i + (1 - \tau) \bar{\theta}_i$ for $i \in \{1, 2\}$
- 16: **end for**
- 17: **until** stopping criteria is met

One noteworthy difference from conventional actor-critic methods is that SAC uses two Q-functions. While not strictly necessary, this significantly speeds up training [10].

Between 2018 and 2019, when SAC was first introduced, it outperformed other popular methods such as TD3 [8], DDPG [18], and PPO [26] in continuous control benchmarks.

2.2.3 Proximal Polixy Optimizatin

One of the instabilities in RL algorithms is that a change in the actions taken by a policy can have large consequences, both good and bad. A natural idea is to try and limit the changes that can happen to a policy at any step. You can do this by using a surrogate objective function that is limited to be at least somewhat similar to the current policy. This was first implemented using a trust region (i.e an area that we can safely move the policy and is guaranteed to improve) and resulted in the TRPO algorithm [24]. Yet the most popular method is that of PPO [26], as it is computationally simpler. The method used in PPO is to use a clipped objective which results in a loss function like so:

$$L^{\text{CLIP}}(s, a, \theta_k, \theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a), \text{clip} \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1 - \epsilon, 1 + \epsilon \right) A^{\pi_{\theta_k}}(s, a) \right) \quad (2.4)$$

Where the advantage function ($A^{\pi_{\theta_k}}(s, a)$) is a way of measuring how much better the situation is than expected. The ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ is used to measure the difference between the current policy and the old policy. This ratio is what is clipped and used as a scalar for the advantage function. As we see in the situation that the action was better ($A > 0$) the clip function will apply and make sure that L wont be too large, however it can be as small as 0. In the other case when things were worse than expected ($A < 0$) it capped to make L not close to zero (note that L will be negative), yet L could be as large (in the negative sense) as it would like.

The reasoning is that allowing for these large and negative L means that we can do a better job of making sure that this action is not taken again.

The advantage function can be estimated in many ways. A simple advantage function could be actual return - estimated return ($G_t - V(s_t)$) for episodic tasks, or TD error ($r_t + \gamma V(s_{t+1}) - V(s_t)$) for continuing tasks. The implementation in the paper uses generalised advantage estimation which is an exponentially weighted sum of TD errors [25], as it better balances bias and variance. GAE is defined as:

$$\hat{A}_t^{GAE(\gamma, \lambda)} = \sum_{l=0}^{\infty} (\gamma \lambda)^l \delta_{t+l}^V \quad (2.5)$$

Where $\delta_t^V = r_t + \gamma V(s_{t+1}) - V(s_t)$. γ is the usual discounting factor but this is expanded on with λ which is from 0-1, at 0 it is TD(0) error and at 1 would be the full complete return.

With this advantage estimator we need to learn a value function. This puts PPO in the same style of actor-critic methods. It is however regarded as a Policy gradient method starting with the author Schulman designating it so [26].

The CLIP loss function can be augmented with other objectives to increase its effectiveness. For example using entropy maximisation to encourage exploring like what is done above with 2.2.2. This leads to this objective:

$$L^{CLIP+S}(s, a, \theta_k, \theta) = \hat{\mathbb{E}}_t \left[L^{CLIP}(s, a, \theta_k, \theta) + cS[\pi_\theta](s) \right] \quad (2.6)$$

Where S is the entropy of the policy. There is a large space of exploring different surrogate objective functions which incorporate the CLIP function.

Below is the algorithm from the paper [26] with some modifications to make it clearer.:

Algorithm 6 PPO Algorithm, Actor Critic style

Require: Clipping amount ϵ , Minibatch size K , epoch M , discount factor γ , learning rates α_θ and α_ϕ , number of actors N , environment steps T , neural network function approximator V , neural network function approximator π , value function coefficient c_1 and entropy coefficient c_2

- 1: Initialize policy π_θ and value function V_ϕ with random weights θ and ϕ
- 2: $\theta_{old} \leftarrow \theta$
- 3: **repeat**
- 4: **for** actor = 1, 2, ..., N **in parallel do**
- 5: Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
- 6: Collect trajectory $(s_t, a_t, r_t, s_{t+1})_{t=0}^{T-1}$
- 7: Compute advantage estimates $\hat{A}_1, \dots, \hat{A}_T$ using GAE
- 8: Compute returns $\hat{R}_t \leftarrow \sum_{i=t}^{T-1} \gamma^{i-t} r_i$
- 9: **end for**
- 10: Combine data from all actors: $\mathcal{D} = \{(s_t^n, a_t^n, \hat{A}_t^n, \hat{R}_t^n)\}$ for $n = 1, \dots, N$ and $t = 1, \dots, T$
- 11: **for** epoch = 1, 2, ..., M **do**
- 12: Shuffle data \mathcal{D} and split into minibatches of size $K \leq NT$
- 13: **for** each minibatch $\mathcal{B} \subset \mathcal{D}$ **do**
- 14: For each $(s, a, \hat{A}, \hat{R}) \in \mathcal{B}$:
- 15: Compute ratio $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$
- 16: Compute clipped surrogate objective:
- 17: $L_{clip}(\theta) = \min(r(\theta)\hat{A}, \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A})$
- 18: Compute value function loss:
- 19: $L_{VF}(\phi) = (V_\phi(s) - \hat{R})^2$
- 20: Compute entropy bonus:
- 21: $S[\pi_\theta](s) = -\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)$
- 22: Total objective: $L(\theta, \phi) = \mathbb{E}[L_{clip}(\theta) - c_1 L_{VF}(\phi) + c_2 S[\pi_\theta](s)]$
- 23: Update θ using SGD or Adam optimizer: $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta L(\theta, \phi)$
- 24: Update ϕ using SGD or Adam optimizer: $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\theta, \phi)$
- 25: **end for**
- 26: **end for**
- 27: $\theta_{old} \leftarrow \theta$
- 28: **until** stopping criteria is met

2.2.4 Twin Delayed Deep Deterministic Policy Gradient

A problem with using value function based methods with approximation is that they can lead to overestimation bias (an agent thinking a situation is better than it really is). This has been shown to be a problem in the continuous control setting [8]. A way to resolve this overestimation error is to use two critics to estimate the value of the action taken and then use the minimum of the two to update the policy. This is called Twin Delayed Deep Deterministic Policy Gradient [8] and is a variant of DDPG [17] that builds on Double Q-Learning [37]. It is a actor critic method that also uses experience replay, target networks and policy noise to stabilise learning.

The policy noise method is not seen in the other algorithms mentioned above. The clipped policy noise in the critic target is used to make sure that the critics aren't overfitting to peaks. This works on the intuition that similar actions should have similar values.

The algorithm from the original paper [8] with slight modifications for clarity:

Algorithm 7 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates α_π and α_Q , target smoothing coefficient τ , policy noise σ , target policy noise clip c , target policy noise $\tilde{\sigma}$, update frequency d and neural network function approximator Q and π

- 1: Initialize critic networks $Q_{\theta_1}, Q_{\theta_2}$, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
- 2: Initialize target networks $\theta'_1 \leftarrow \theta_1, \theta'_2 \leftarrow \theta_2, \phi' \leftarrow \phi$
- 3: Initialize replay buffer \mathcal{D}
- 4: **repeat**
- 5: increment timestep t
- 6: Select action with exploration noise $a \sim \pi_\phi(s) + \epsilon, \epsilon \sim \mathcal{N}(0, \sigma)$ and observe reward r and new state s'
- 7: Store transition tuple (s, a, r, s') in \mathcal{D}
- 8: Sample mini-batch of K transitions (s, a, r, s') from \mathcal{D}
- 9: Initialize critic loss $L_{\theta_1} \leftarrow 0, L_{\theta_2} \leftarrow 0$
- 10: **for** $i = 1$ to K **do**
- 11: Add noise to target action: $\tilde{a}_i \leftarrow \pi_\phi(s'_i) + \text{clip}(\epsilon_i, -c, c)$ where $\epsilon_i \sim \mathcal{N}(0, \tilde{\sigma})$
- 12: Compute target Q-value: $y_i \leftarrow r_i + \gamma \min(Q_{\theta'_1}(s'_i, \tilde{a}_i), Q_{\theta'_2}(s'_i, \tilde{a}_i))$
- 13: Accumulate critic losses: $L_{\theta_j} \leftarrow L_{\theta_j} + (y_i - Q_{\theta_j}(s_i, a_i))^2$ for $j = 1, 2$
- 14: **end for**
- 15: Update critics: $\theta_i \leftarrow \theta_i - \alpha_Q \nabla_{\theta_i}(L_{\theta_i}/K)$ for $i = 1, 2$
- 16: **if** $t \bmod d$ **then**
- 17: Update ϕ by the deterministic policy gradient:
$$\nabla_\phi J(\phi) = K^{-1} \sum_i^K \nabla_a Q_{\theta_1}(s_i, a)|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i)$$
- 18: Update target networks:
$$\theta'_i \leftarrow \tau \theta_i + (1 - \tau) \theta'_i$$
- 19: $\phi' \leftarrow \tau \phi + (1 - \tau) \phi'$
- 20: **end if**
- 23: **until** stopping criteria is met

2.3 State of the Art Algorithms

Since the introduction of the modern deep reinforcement learning algorithms, there have been many new algorithms that have been developed. These generally build on the ideas from the first generation of deep reinforcement learning algorithms. I will go into detail about 4 algorithms which are actor critic methods and build on the ideas of SAC 2.2.2. These are:

Truncated Quantile Critics (TQC) [14] which uses an ensemble of critics and a truncated distributional critic with the goal of reducing the overestimation bias.

Randomized Ensemble Double Q-Learning (REDQ) [7] which uses an ensemble of critics which are trained towards a target made up of the minimum of a random subset of the critics. This allows for a more stable learning process that can handle Updates to Data ratio (UTD) which are greater than 1.

Dropout Q-functions (DROQ) [11] which is a variant of REDQ that uses smaller number of critics and instead adds layer dropout and layer normalization to handle a UTD greater than 1.

Cross Q-learning (CrossQ) [6] which builds upon SAC to increase sample efficiency yet keep computation similar to SAC. It does this by removing the target networks, making the critic function take both current and the next state action pair to predict both the current and next step action value in one forward pass. applying batch normalization.

2.3.1 Truncated Quantile Critics

One fo the challenges in learning an accurate critic function is the overestimations bias. This comes from the fact that the critic is updated towards the maximum of the Q-values. TQC [14] mitigates this by using multiple distributional critics and truncating the top quantile of the distributions.

The distributional reinforcement learning framework [4] is different from traditional Q-learning because rather than learning the expected value for the state action pair, it learns the return.

$$Z_{\psi_n}(s, a) := \frac{1}{M} \sum_{m=1}^M \delta \left(\theta_{\psi_n}^m(s, a) \right)$$

It approximates the distribution with M number of quantiles. each predicting the value of the return at a given quantile θ_{ψ}^m .

Rather than just having a single critic, TQC uses N critics. It takes all of the quantiles from the critics and mixes them together to form a single distribution. From there it truncates the top k values. Kuznetsov et al [14] posits that the order of mixing then truncating rather than truncating and mixing is important. These quantiles are then used to create a target which the critics are trained towards. In practice target critic networks are also used as well to stabilize the learning process.

The algorithm from the paper [14] with slight modifications is shown below:

Algorithm 8 Truncated Quantile Critics (TQC)

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates; $\lambda_\alpha, \lambda_\pi, \lambda_Q$, number of critics C , truncation amount k , number of quantiles M , target critic update rate β , Updates to Data ratio UTD, and neural network function approximator Z and π

- 1: Initialize replay buffer \mathcal{D}
- 2: Initialize policy π_ϕ and C critics $Z_{\psi_i}, Z_{\bar{\psi}_i}$ with random parameters ψ_i
- 3: Set $\alpha = 1$ and $\mathcal{H}_T = -\dim \mathcal{A}$
- 4: **repeat**
- 5: Run policy π_ϕ in environment until termination or truncation
- 6: Append trajectories $(s_t, a_t, r_t, s_{t+1})_{t=0}^{T-1}$ to \mathcal{D}
- 7: **for** UTD times **do**
- 8: Sample a batch B of size K from the replay \mathcal{D}
- 9: Update α with gradient descent using
$$\nabla_\alpha \frac{1}{K} \sum_{(s,a) \in B} \log \alpha (-\log \pi_\phi(a|s) - \mathcal{H}_T)$$
- 10: Update ϕ with gradient descent using
$$\nabla_\phi \frac{1}{K} \sum_{(s,a) \in B} \left(\alpha \log \pi_\phi(a|s) - \frac{1}{NM} \sum_{n=1}^N \sum_{m=1}^M \theta_{\psi_n}^m(s, a) \right)$$
- 11: Update ψ_n with gradient descent using for $n = 1, \dots, C$:
$$\nabla_{\psi_i} \frac{1}{KkCM} \sum_{(s,a) \in B} \sum_{m=1}^M \sum_{i=1}^{kC} \rho_{\tau_m}^H \left(y_i(s, a) - \theta_{\psi_n}^m(s, a) \right)$$
- 12: $\bar{\psi}_n \leftarrow \beta \psi_n + (1 - \beta) \bar{\psi}_n$ for $n \in [1..C]$
- 13: **end for**
- 14: **until** stopping criteria is met

We can see that this algorithm is similar to that of SAC 2.2.2 in that it uses a replay buffer, target networks and entropy maximization. It varies from SAC in that the interaction of the environments is done until termination or truncation where the truncation. That means that a complete episode is completed in-between each update. It is also important to note that the policy is optimized using the untruncated distributions from the critics because there is not the same feedback that causes an overestimation bias.

2.3.2 Randomized Ensemble Double Q-Learning

Chen et al [7] made an algorithm to increase its updates to data ratio (UTD) to be greater than 1. Naively increasing the UTD to be grater than 1 in SAC 2.2.2 has poor results, which comes from the instability in the learning. To stabilize the learning REDQ using an ensemble of critics with a target made up of the minimum of a random subset of the critics.

The ensemble of C critics is trained towards a target which is the minimum of a random subset M of the critics. This target is in turn used to update all of the critics. The policy is optimized using the average of the critics. With an $C = 10$ and $M = 2$ the algorithm was able to handle a UTD of 20 [7]. The authors reference implementation of the algorithm is

built of SAC 2.2.2 and interesting turns back into SAC if $C = M = 2$ and $UTD = 1$.

At the time it was the first model-free algorithm to be able to handle a UTD greater than 1. This higher ratio allows for an increase in sample efficiency as it is extracting more information from the replay buffer per time step. This does however increase the computational cost as each time step requires $C \times UTD$ gradient descent steps and M forward passes. In practice this means that it is doing 200 gradient descents per time step rather than the 1 that TQC 2.3.1 and SAC 2.2.2 do.

Pseudo code for the algorithm taken from the paper [7] with some modifications for clarity is shown below:

Algorithm 9 Randomized Ensembled Double Q-learning (REDQ)

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ_π, λ_Q , number of critics C , target critic update rate β , entropy coefficient α critic subsets M , Updates to Data ratio UTD, and neural network function approximator Q and π

- 1: Initialize policy parameters θ , C Q-function parameters $\phi_i, i = 1, \dots, N$, empty replay buffer \mathcal{D} . Set target parameters $\phi_{\text{targ},i} \leftarrow \phi_i$, for $i = 1, 2, \dots, N$
- 2: **repeat**
- 3: Select action: $a_t \sim \pi_\theta(a_t | s_t)$ and observe reward r_t and next state s_{t+1}
- 4: Store transition (s_t, a_t, r_t, s_{t+1}) in \mathcal{D}
- 5: **for** UTD times **do**
- 6: Sample a batch B of size K from the replay \mathcal{D}
- 7: Sample a set M of M distinct indices from $\{1, 2, \dots, C\}$
- 8: Compute the Q target y (same for all of the C Q-functions):

$$y = r + \gamma \left(\min_{i \in M} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' | s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot | s')$$

- 9: **for** $i = 1, \dots, C$ **do**
- 10: Update ϕ_i with gradient descent using

$$\nabla_\phi \frac{1}{K} \sum_{(s,a,r,s') \in B} (Q_{\phi_i}(s, a) - y)^2$$

- 11: Update target networks with $\phi_{\text{targ},i} \leftarrow \beta \phi_{\text{targ},i} + (1 - \beta) \phi_i$
- 12: **end for**
- 13: **end for**
- 14: Update policy parameters θ with gradient ascent using

$$\nabla_\theta \frac{1}{K} \sum_{s \in B} \left(\frac{1}{C} \sum_{i=1}^C Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s) | s) \right), \quad \tilde{a}_\theta(s) \sim \pi_\theta(\cdot | s)$$

- 15: **until** stopping criteria is met

The algorithm uses some of the same stabilization tricks like replay buffer, entropy maximization and target networks. However the structure is a bit different, it only does one action followed by some number of gradient updates and lastly a single policy update.

2.3.3 Dropout Q-functions

The big draw back of REDQ 2.3.2 is that it requires a large number of critics to be able to handle a UTD greater than 1. Hiracka et al [11] proposed a method called Dropout Q-

functions (DroQ) which uses a smaller number of critics and instead uses dropout and layer normalization to handle the UTD. This allows for a similar sample efficiency at about half the computational cost (DroQ is about 2x faster than REDQ) [11].

Dropout layers [27] are used within the critic networks to randomly drop out neurons during training. It is used within the deep learning community to help prevent overfitting. The intuition being that randomly removing neurons forces the networks to learn features that are not reliant on single neurons. However in this setting it is being used to model the uncertainty in the Q-value. Layer normalization [3] is used after the dropout layer which increases the effectiveness of the dropout layer.

As DroQ uses a smaller ensemble of critics it does not take a subset and instead uses all of the critics to create the critic target for critic updating. This is in contrast to REDQ 2.3.2 which uses a random subset of the critics. Other than this the algorithm is almost identical to REDQ as seen below (red highlights the differences):

Algorithm 10 Dropout Q-functions (DroQ)

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ_π, λ_Q , number of critics C , target critic update rate β , entropy coefficient α , **dropout rate**, Updates to Data ratio UTD, and neural network function approximator Q and π

- 1: Initialize policy parameters θ , C Q-function parameters $\phi_i, i = 1, \dots, N$, empty replay buffer \mathcal{D} . Set target parameters $\phi_{\text{targ},i} \leftarrow \phi_i$, for $i = 1, 2, \dots, N$
- 2: **repeat**
- 3: Select action: $a_t \sim \pi_\theta(a_t | s_t)$ and observe reward r_t and next state s_{t+1}
- 4: Store transition (s_t, a_t, r_t, s_{t+1}) in \mathcal{D}
- 5: **for** UTD times **do**
- 6: Sample a mini-batch $B = \{(s, a, r, s')\}$ from \mathcal{D} of size K
- 7: Compute the Q target y (same for all of the C Q-functions):

$$y = r + \gamma \left(\min_{i \in C} Q_{Dr, \phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}' | s') \right), \quad \tilde{a}' \sim \pi_\theta(\cdot | s')$$

- 8: **for** $i = 1, \dots, C$ **do**
- 9: Update ϕ_i with gradient descent using

$$\nabla_\phi \frac{1}{K} \sum_{(s,a) \in B} (Q_{Dr, \phi_i}(s, a) - y)^2$$

- 10: Update target networks with $\phi_{\text{targ},i} \leftarrow \beta \phi_{\text{targ},i} + (1 - \beta) \phi_i$
- 11: **end for**
- 12: **end for**
- 13: Update policy parameters θ with gradient ascent using

$$\nabla_\theta \frac{1}{K} \sum_{s \in B} \left(\frac{1}{C} \sum_{i=1}^C Q_{Dr, \phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s) | s) \right), \quad \tilde{a}_\theta(s) \sim \pi_\theta(\cdot | s)$$

- 14: **until** stopping criteria is met
-

2.3.4 Cross Q-learning

The goal of both REDQ 2.3.2 and DROQ 2.3.3 is to increase the sample efficiency and they achieve it by using a UTD $\gg 1$ (typically around 20). However they both come at a com-

tational cost which increases which increases linearly with the UTD. CrossQ [6] instead is just a simple improvement on SAC 2.2.2 which removes the target networks allowing it to use batch normalization [13]. This combination allows it to achieve higher sample efficiency than REDQ 2.3.2 and DROQ 2.3.3 while being computationally simpler resulting in about 4 times faster (using wall clock measurements). It gains this computational advantage by using a UTD of 1. Much wider critic networks (2000) are also used as it further increases the sample efficiency [6].

To allow the batch normalization to work without the target networks we combine the calculation of the current Q-value and the next Q-value. This is done by concatenating the current state and action with the next state and action. This means that the loss function is changed from what we see in SAC 2.2.2 which is:

$$J_Q(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[(Q_\theta(s, a) - r - \gamma Q_{\bar{\theta}}(s', \pi_\phi(s')))^2 \right] \quad (2.7)$$

To no longer use the target network and instead the output from a single Q-function:

$$J_{Q_{BRN}}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[(q_t - r - \gamma q_{t+1})^2 \right] \text{ and } Q_{BRN,\theta} \left(\begin{bmatrix} S \\ S' \end{bmatrix}, \begin{bmatrix} A \\ \pi_\psi(s') \end{bmatrix} \right) = \begin{bmatrix} q_t \\ q_{t+1} \end{bmatrix} \quad (2.8)$$

This concatenation is important as it allows the batch normalization to work as half of the data is from the distribution made with the current policy and the other half is from the distribution made with the updated policy. Because it is half and half none of the data is out of distribution so that normalization works as expected.

These changes are applied to SAC 2.2.2 in the paper however the changes could be applied to other off policy TD learning based algorithms. The algorithm is almost identical to the SAC algorithm which can be found here 5. The differences are highlighted in red below:

Algorithm 11 Cross Q-learning (CrossQ)

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ, λ_π and λ_Q , update frequency C , target smoothing coefficient τ , Updates to Data ratio UTD, environment steps S , neural network function approximator Q , neural network function approximator π and initial entropy coefficient α .

- 1: Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
- 2: Initialize policy π_ϕ with random weights ϕ
- 3: Initialize both action-value function Q_{θ_i} with random weights θ_i
- 4: **repeat**
- 5: **for** S steps **do**
- 6: select action: $a_t \sim \pi_\phi(a_t | s_t)$
- 7: Observe reward r_t and next state s_{t+1}
- 8: Store transition (s_t, a_t, r_t, s_{t+1}) in \mathcal{D}
- 9: **end for**
- 10: **for** UTD times **do**
- 11: Update Q-functions: $\theta_i \leftarrow \theta_i - \lambda_{Q_{BRN}} \hat{\nabla}_{\theta_i} J_{Q_{BRN}}(\theta_i)$ for $i \in \{1, 2\}$
- 12: Update policy: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
- 13: Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
- 14: **end for**
- 15: **until** stopping criteria is met

The differences are in the loss function as mentioned above, critic networks that have batch normalisation and the removal of target networks.

2.3.5 SUNRISE

Sunrise stands for 'Simple Unified Framework for Ensemble Learning in Deep Reinforcement Learning' [16]. It is built with the goal of using the ensemble to help stabilize learning and manage the exploration and exploitation trade off. The ideas of Sunrise can be applied to any off policy RL algorithm, however for illustration purposes focus will be put on the SAC variant that uses Soft actor-critic agents [9]. The algorithm works by applying two different methods. Firstly is weighted bellman backups which works with the idea that the amount to update a network should be determined by how confident the agent is in the given update. An understanding of confident can be achieved by using the variance of the ensemble of Q-functions. The actual weight is achieved using the equation.

$$w(s, a) = \sigma(-\bar{Q}_{\text{std}}(s, a) * T) + 0.5 \quad (2.9)$$

Where σ is the sigmoid function and T is the temperature. One can observe that it will always be in the bounds of $[0.5, 1]$ which means implementing the weighted bellman backup is as simple as multiplying the loss function by the weight function.

The second method that Sunrise uses is UCB exploration which involves choosing the next action from the ensemble so that it maximises both the expected reward and the uncertainty. This is to directly help manage the exploration and exploitation tradeoff. The uncertainty is modeled with $\gamma Q_{\text{std}}(s, a)$ where γ is a hyperparameter which determines the importance of the uncertainty in choosing the next action. Importantly it can be noted that as the sunrise algorithm learns the std of the Q-functions will decrease as they converge on Q^* .

As this is an ensemble method it relies on the fact that each of the base learners are different from one another. In sunrise variation is guaranteed using random initialisation and different learning experience. The random initialisation is simply using a different initialised weights for the actor and critic networks. Different learning experience is achieved by giving each transition a mask which determines which base learners will use it when they update parameters. This mask is made using the Bernoulli distribution.

Put together the algorithm for Sunrise is as follows.

Algorithm 12 Sunrise Algorithm

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ, λ_π and λ_Q , update frequency C , target smoothing coefficient τ , Updates to Data ratio UTD, environment steps S , neural network function approximator Q , neural network function approximator π , initial entropy coefficient α , warmup steps W , ensemble size E and Bellman weight temperature T

- 1: Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
- 2: **for** $i \in \{1, 2, \dots, E\}$ **do**
- 3: Initialize critic $Q_{\theta_{i,j}}$ with random weights $\theta_{i,j}$ for $j \in \{1, 2\}$
- 4: Initialize target critic $\bar{Q}_{\theta_{i,j}} \leftarrow Q_{\theta_{i,j}}$ for $j \in \{1, 2\}$
- 5: Initialize policy π_{ϕ_i} with random weights ϕ_i
- 6: **end for**
- 7: **repeat**
- 8: **for** S steps **do**
- 9: // UCB exploration alternatively randomly select a base learner to be used for each episode.
- 10: Collect E action samples $a_t^{(i)} = \pi_\phi(s_t)$ for $i \in \{1, 2, \dots, E\}$
- 11: select action: $a_t \sim \pi_\phi(a_t | s_t)$
- 12: Choose the action that maximizes the UCB exploration term:
- 13: $a_t = \arg \max_a (Q_{\theta_1}(s_t, a) + \alpha \cdot \sigma(Q_{\theta_2}(s_t, a)))$
- 14: Observe reward r_t and next state s_{t+1}
- 15: Sample bootstrap masks $m_t = \{m_{t,i} \sim \text{Bernoulli}(0.5) \text{ for } i \in \{1, \dots, E\}\}$
- 16: Store transition $(s_t, a_t, r_t, s_{t+1}, m_t)$ in \mathcal{D}
- 17: **end for**
- 18: **if** Size of \mathcal{D} is less than W **then**
- 19: Continue to next iteration without training or removal checks
- 20: **end if**
- 21: **for** UTD times **do**
- 22: Sample a batch B of size K from \mathcal{D}
- 23: **for** each agent i **do**
- 24: Use Mask $m_{t,i}$ to select transitions from B for agent i
- 25: Update Q-functions to minimize $(\text{sigmoid}(-\bar{Q}_{\text{std}}(s_{t+1}, a_{t+1}) * T) + 0.5)(Q_{\theta_i}(s_t, a_t) - r_t - \gamma \bar{V}(s_{t+1}))$
- 26: Update policy to minimize $\mathbb{E}_{a_t \sim \pi_\phi} [\alpha \log \pi_\phi(a_t | s_t) - Q_\theta(s_t, a_t)]$
- 27: Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
- 28: Update target Q-networks: $\bar{\theta}_i \leftarrow \tau \theta_i + (1 - \tau) \bar{\theta}_i$ for $i \in \{1, 2\}$
- 29: **end for**
- 30: **end for**
- 31: **until** stopping criteria is met

Chapter 3

Dynamic Sunrise Algorithm

Dynamic Sunrise is a algorithm that applies some of the ideas from Sunrise [16] and SAC [10] to create a new algorithm that increases exploration by using a dynamic ensemble. The DSunrise algorithm is the built off the sunrise algorithm with the addition of a dynamic ensemble. That is on a regular interval (every 10,000 steps) the ensemble is checked to see if any of the base learners can be removed. On removal of a base learner a new learner will be instantiated and added to the ensemble. Therefore the two important aspects of the DSunrise algorithm is the removal check and the instantiation of new base learners.

3.1 Removal Check

A removal check is a routine that is run on some regular interval (every step, episode, n episodes etc). As there is no managing critic the check only has the current base learners and the historic performance of these base learners to look at. From this information two classes of metrics can be created one is a performance check and the second is a diversity check.

The performance check can either look at historic performance and/or the predicted performance of the base learner. Historic performance involve a type of time weighted average from the last n episodes (which involves each base learner tracking its past performance). Alternatively predicted performance could be used which involves comparing the predicted return for each base learner for a sample of states.

The diversity check is trying to quantify how different the base learners are from each other. As the goal of an ensemble is that the base learners are different from each other removing base learners that happen to be too similar to each other will reduce the computational cost of the ensemble and provide opportunity for new base learners to be added to increase diversity. The diversity of a base learners can be understood by either how similar the critics are or how similar the proposed actions are from the policies. Furthermore as there is a unique optimal policy the ensemble of base learners should converge to this optimal policy and so the size of the ensemble will naturally decrease as training progresses.

In practice a diversity based metric is used to identify pairs of base learners that are too similar to each other. The lower performing base learner is then removed from the ensemble. The performance and diversity metrics are defined below.

$$\text{performance}_i^{(t)} = \sum_{\tau=t-n}^t \omega_\tau \cdot R_i^{(\tau)} \quad \text{where} \quad \omega_\tau = \frac{\gamma^{t-\tau}}{\sum_{k=t-n}^t \gamma^{t-k}}$$

$$\text{diversity}_i^{(t)} = \min_{\substack{j=1, \dots, |\mathcal{E}| \\ j \neq i}} \frac{\frac{\sum_{s \in \mathcal{S}} \frac{\|\mathbf{a}_i(s) - \mathbf{a}_j(s)\|_2}{D_{\max}}}{|\mathcal{S}|}}{|\mathcal{S}|}$$

Where:

- $\text{performance}_i^{(t)}$: Performance metric for learner i at episode t
- $\text{diversity}_i^{(t)}$: Diversity metric for learner i at episode t
- D_{\max} : Maximum l2 distance between actions in the action space
- $R_i^{(\tau)}$: Reward of learner i in its τ -th transition
- γ : Temporal discount factor ($0.9 \leq \gamma \leq 0.99$)
- n : Window size for performance evaluation
- \mathcal{S} : State sample from replay buffer ($|\mathcal{S}| = m$)
- \mathcal{E} : Current ensemble of learners
- $\mathbf{a}_i(s)$: Action vector of learner i in state s

These metrics are used to determine if a base learner should be removed from the ensemble. The removal check is run on a regular interval and will find if there are any base learners that are too similar to each other and if so remove the base learner that is performing worse. This is because diversity measure is symmetric so similar base learners will always be found in pairs. The removal check is defined as follows:

$$\text{remove}_i^{(t)} = \begin{cases} 1 & \text{if } \exists j \neq i \text{ such that } \text{diversity}_i^{(t)} < d_{\text{thre}} \text{ and } \text{performance}_i^{(t)} < \text{performance}_j^{(t)} \\ 0 & \text{otherwise} \end{cases}$$

With:

- d_{thre} : Diversity threshold (≈ 0.2)

3.2 Instantiation of new base learners

When a base learner is removed from the ensemble a new base learner can be instantiated. There are many different ways to instantiate new models. One way of instantiating a new base learner is to randomly instantiate a new base learner then train it up on some random subset of the replay buffer. This will add variation to it through the random initialization and the random subset of the replay buffer. However depending on the size of the experience subset this could be computationally similar to simply having the base learner as part of the ensemble since the beginning of the training. Another way to instantiate a new base learner is for it to somehow be a variation of a current base learner. This could be done in a variety of ways which can broadly be separated into either cross over (involving multiple base learners) or mutation (involving a single base learner). Once the instantiation of

the new base learner is complete it could pass through a removal check to see if it is worth keeping in the ensemble. If it is not worth keeping then the ensemble size will naturally decrease. This naturally decreasing ensemble size is a feature that trends towards a single base learner which is computationally efficient (and potentially the optimal policy, depending on the guarantees of the removal check).

The current method of instantiation involves applying random mutation to the actor network and then training it on a small random subset of the experience replay buffer.

Algorithm 13 Generate Replay Subset

Require: Replay memory \mathcal{D} , subset size m , Reward ratio r_{top} , Error ratio r_{error} , Random sample ratio r_{random}

1: Select top transitions by reward:

$$\mathcal{D}_{\text{reward}} \leftarrow \text{Top } \lfloor r_{\text{top}} \cdot m \rfloor \text{ transitions by reward}$$

2: Select top transitions by error:

$$\mathcal{D}_{\text{error}} \leftarrow \text{Top } \lfloor r_{\text{error}} \cdot m \rfloor \text{ transitions by error}$$

3: Randomly sample transitions from \mathcal{D} :

$$\mathcal{D}_{\text{random}} \leftarrow \text{Randomly sample } \lfloor r_{\text{random}} \cdot m \rfloor \text{ transitions}$$

4: Combine subsets:

$$\mathcal{D}_{\text{subset}} \leftarrow \mathcal{D}_{\text{reward}} \cup \mathcal{D}_{\text{error}} \cup \mathcal{D}_{\text{random}}$$

5: **return** $\mathcal{D}_{\text{subset}}$

Note that in the current implementation the replay subset is simply a random subset of the replay buffer. This was done due to time constraints.

Algorithm 14 Instantiate New Base Learner

Require: Replay buffer \mathcal{D} , Ensemble \mathcal{E} , Mutation noise σ , Diversity threshold p_{crit} , Base learner θ_{base} , Retrain steps R

1: Generate mutation noise $\delta \sim \mathcal{N}(0, \sigma^2)$

2: Instantiate new base learner:

$$\theta_{\text{new}} \leftarrow \theta_{\text{base}} + \epsilon \cdot \delta$$

3: Call **Generate Replay Subset** to create training data:

replay subset \leftarrow **Generate Replay Subset**($\mathcal{D}, m, r_{\text{top}}, r_{\text{error}}, r_{\text{random}}$)

4: **for** retraining steps R **do**

5: Sample a batch B from the replay subset

6: Train new base learner with batch B ▷ Using same train function as regular base learner training

7: **end for**

8: **if** diversity_{new} $< p_{\text{crit}}$ **then**

9: Discard new base learner

10: **else**

11: Add new base learner to ensemble \mathcal{E}

12: **end if**

3.3 Basic algorithm

We can put the removal check and the instantiation of new base learners together to create a dynamic ensemble of soft actor critics. The algorithm is based on the Sunrise algorithm [16] with the addition of the dynamic ensemble. The algorithm is outlined in Algorithm 15. A implementation of the algorithm can be found at <https://github.com/1jamesthompson1/sunrise> and further discussion on the code can be found in the appendix A.3.

The structure of the algorithm is the same as the Sunrise algorithm 12. The only difference is that the algorithm has a removal check and instantiation of new base learners. Note that this algorithm does not take into account the various extra bits of information that will need to be stored for the proposed removal check and the instantiation of new base learners.

3.4 Hyper-parameters

This algorithm adds various new hyper-parameters compared to the Sunrise algorithm. These new hyper-parameters include functions which will each have their own hyper-parameters discussed below.

- **Warmup steps:** The number of steps before the algorithm starts training and performing removal checks.
- **Removal check frequency:** The number of steps between each removal check. If the removal check is aggressive in mutating/removing base learners then this should be set to a low value. However if it is not aggressive then it can be a higher value.

Algorithm 15 Dynamic Sunrise Algorithm

Require: Replay buffer size N , Minibatch size K , discount factor γ , learning rates λ, λ_π and λ_Q , update frequency C , target smoothing coefficient τ , Updates to Data ratio UTD, environment steps S , neural network function approximator Q , neural network function approximator π , initial entropy coefficient α , ensemble size E , Bellman weight temperature T , Diversity threshold d_{thre} , Critical diversity threshold d_{crit} , max number of removals r , removal check frequency R , warmup steps W , removal function $\text{remove}_i^{(t)}$, instantiation function instantiate_i

```

1: Initialize replay memory  $\mathcal{D} \leftarrow \emptyset$  with capacity  $N$ 
2: for  $i \in \{1, 2, \dots, E\}$  do
3:   Initialize critic  $Q_{\theta_{i,j}}$  with random weights  $\theta_{i,j}$  for  $j \in \{1, 2\}$ 
4:   Initialize target critic  $\bar{Q}_{\theta_{i,j}} \leftarrow Q_{\theta_{i,j}}$  for  $j \in \{1, 2\}$ 
5:   Initialize policy  $\pi_{\phi_i}$  with random weights  $\phi_i$ 
6: end for
7: repeat
8:   for  $S$  steps do
9:     Collect  $E$  action samples  $a_t^{(i)} = \pi_\phi(s_t)$  for  $i \in \{1, 2, \dots, E\}$ 
10:    select action:  $a_t \sim \pi_\phi(a_t | s_t)$ 
11:    Choose the action that maximizes the UCB exploration term:
12:     $a_t = \arg \max_a (Q_{\text{mean}}(s_t, a) + \alpha \cdot \sigma(Q_{\text{std}}(s_t, a)))$ 
13:    Observe reward  $r_t$  and next state  $s_{t+1}$ 
14:    Sample bootstrap masks  $m_t = \{m_{t,i} \sim \text{Bernoulli}(0.5)$  for  $i \in \{1, \dots, E\}\}$ 
15:    Store transition  $(s_t, a_t, r_t, s_{t+1}, m_t)$  in  $\mathcal{D}$ 
16:   end for
17:   if Size of  $\mathcal{D}$  is less than  $W$  then
18:     Continue to next iteration without training of removal checks
19:   end if
20:   for UTD times do
21:     Sample a batch  $B$  of size  $K$  from  $\mathcal{D}$ 
22:     for each agent  $i$  do
23:       Use Mask  $m_{t,i}$  to select transitions from  $B$  for agent  $i$ 
24:       Update Q-functions to minimize  $(\text{sigmoid}(-\bar{Q}_{\text{std}}(s_{t+1}, a_{t+1}) * T) + 0.5)(Q_{\theta_i}(s_t, a_t) - r_t - \gamma \bar{V}(s_{t+1}))$ 
25:       Update policy to minimize  $\mathbb{E}_{a_t \sim \pi_\phi} [\alpha \log \pi_\phi(a_t | s_t) - Q_\theta(s_t, a_t)]$ 
26:       Update entropy coefficient:  $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$ 
27:       Update target Q-networks:  $\bar{\theta}_i \leftarrow \tau \theta_i + (1 - \tau) \bar{\theta}_i$  for  $i \in \{1, 2\}$ 
28:     end for
29:   end for
30:   if the current episode is a multiple of  $R$  then
31:     Calculate  $\text{remove}_i^{(t)}$  for each base learner  $i \in \{1, 2, \dots, E\}$  using the removal check
32:     for Lowest  $r$  base learners  $i$  such that  $\text{remove}_i^{(t)} = 1$  do
33:       Call  $\text{instantiate}_i$  and see if the base learner  $i$  can be replaced with a sufficiently diverse new base learner, otherwise remove it from the ensemble.
34:     end for
35:     Update  $T$  to reflect the new ensemble size (if the new base learner was not diverse enough and removed)
36:   end if
37: until stopping criteria is met

```

- **Max number of removals:** The maximum number of base learners that can be removed in a single removal check. This is to prevent the ensemble from shrinking too quickly. However if a way of adding new base learners is found then this can be increased and/or the ensemble size is greatly increased (≥ 100) then this could be set to a higher value.
- **Removal check function:** This function flags which base learners should be removed from the ensemble. The function will take the current ensemble and use performance and diversity metrics to determine which base learners should be removed. The current implementation uses a diversity metric based on euclidean distance and l2 norm.
- **Instantiation function:** This function is responsible for taking a flagged for removal base learner and attempting to instantiate a new base learner. This function can be as simple or complex as desired. The current implementation is a simple mutation of the actor network and training on a small random subset of the replay buffer. If the new base learner is not diverse enough (due to convergence to the optimal policy) then it will be removed from the ensemble. This is done to produce a smaller ensemble as training progresses.

Both the removal check function and instantiation function can be heavily modified to create very different algorithms. I will expand and list the hyper-parameters for the removal check and instantiation functions that were proposed in 3.2 and 3.1.

The first two functions are the performance $_{(t)}^{(t)}$ and diversity $_{(t)}^{(t)}$ functions which are used in the removal check function.

- **Performance function hyper-parameters:**

- γ : Discount factor for the performance metric. This is a hyper-parameter that can be tuned to change how much weight is given to recent performance.
- n : Window size for the performance metric. This is how far to look back for 'recent' performance.

- **Diversity function hyper-parameters:**

- $|\mathcal{S}|$: Size of the state sample used to calculate the diversity metric. This is a hyper-parameter that can be tuned to change how many states are used to calculate the diversity metric.

These functions are used by the removal check function. The only hyperparameter in the removal function is the diversity threshold d_{thre} which determines what is considered too similar.

The instantiation function has three steps each with their own hyperparameters. Firstly is mutation of the actor network which requires the mutation noise σ . Then generation of the replay subset used for retraining the new base learner. The replay subset generation has three hyper-parameters:

- r_{top} : Ratio of the top transitions by reward to include in the replay subset.
- r_{error} : Ratio of the top transitions by error to include in the replay subset.
- r_{random} : Ratio of random transitions to include in the replay subset.

Lastly is the retraining of the new base learner which has the hyper-parameter R which is the number of steps that the new base learner is trained for before being added to the ensemble. Note that training hyperparameters are the same as the base learner training hyperparameters.

When this has been completed the new base learner completes a diversity check (which uses the same hyperparameters that the removal function uses) and if it is below the critical diversity threshold d_{crit} then it is discarded. If it is above the critical diversity threshold then it is added to the ensemble.

Chapter 4

Experiments

4.1 Introduction

All the algorithms above excluding DQN can work on continuous action spaces. This makes them suited for continuous control tasks that are common within the robotic control domain. I will conduct experiments using the Gymnasium [36] library which provides a framework to use among other things some Mujoco [35] environments. Specifically I will use Ant-v5, Humanoid-v5, Pusher-v5, HalfCheetah-v5, HumanoidStandup-v5, Swimmer-v5, Hopper-v5, InvertedDoublePendulum-v5, Walker2d-v5 and Hopper-v5 which cover a range of locomotion tasks. I will run the experiments we have looked at above on these environments to understand how well they learn.

There are many different dimensions in which we can judge the learning performance of an algorithm. The most common and useful ones are general across not continuous control problem but all reinforcement learning problems. These common and overarching dimensions are:

- Sample efficiency: How many environment interactions does it take to learn a task?
- Computational efficiency: How much computation does it take to learn a task?
- Stability: How stable is the learning process?
- Asymptotic performance: What is the maximum performance of the algorithm?
- Generalization: How well does the algorithm generalise to new tasks, be it new environments or agent?

4.2 Baseline Experiments

I will look at two things for all of my baseline algorithms. The first is the sample efficiency and the second is the computational efficiency. It is important to understand both of these metrics as it can sometimes be easier to trade computational efficiency for sample efficiency. Therefore to make a true improvement to a model it must be better at either/both of these metrics without reducing the other metric.

4.2.1 Sample Efficiency

To measure the sample efficient I will look at the average reward over 10 episodes every 1000 steps. This will give a learning curve to demonstrate how effective its actions are given how many interactions it has had with the environments.

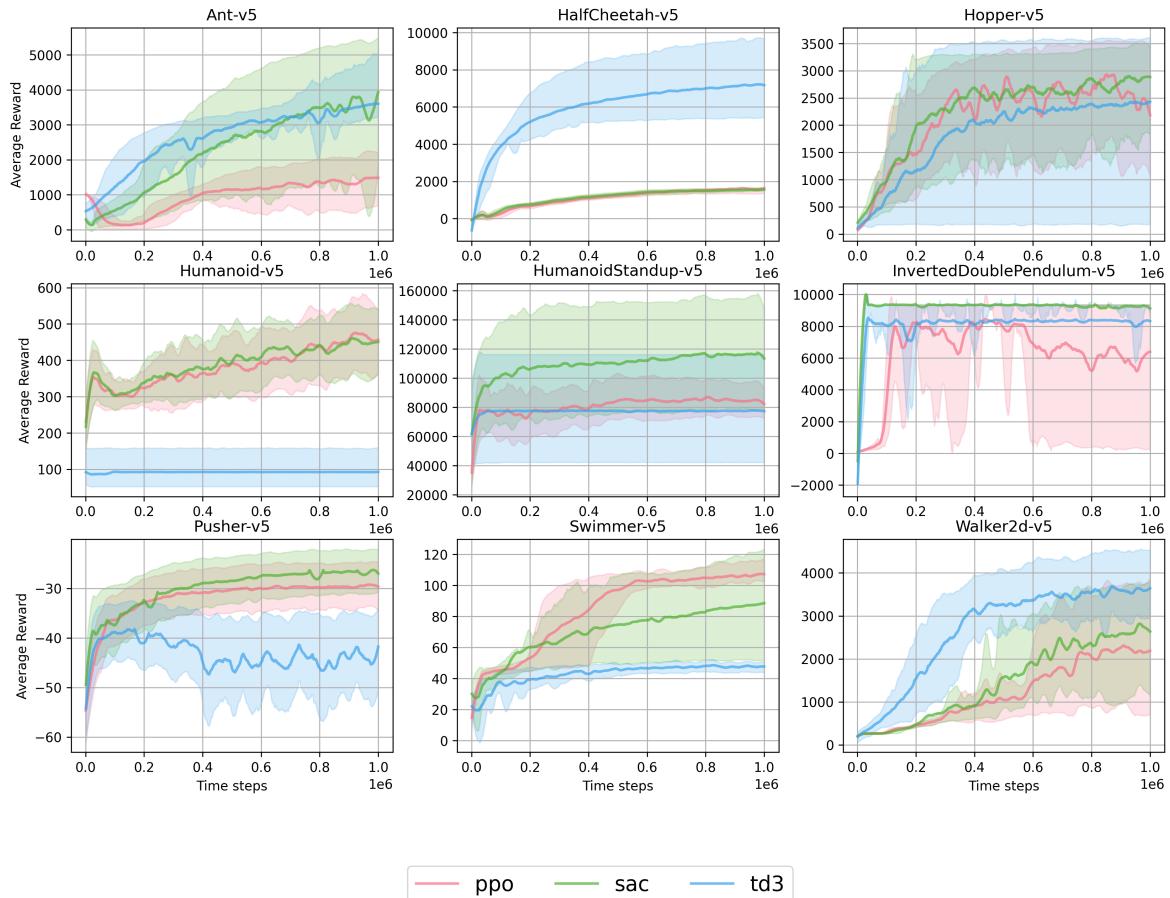


Figure 4.1: Sample Efficiency of older algorithms. The line is a smoothed average return across the 100 episodes made by each seed every 1000 time steps. The shaded area is a smoothed 70% confidence interval highlighted

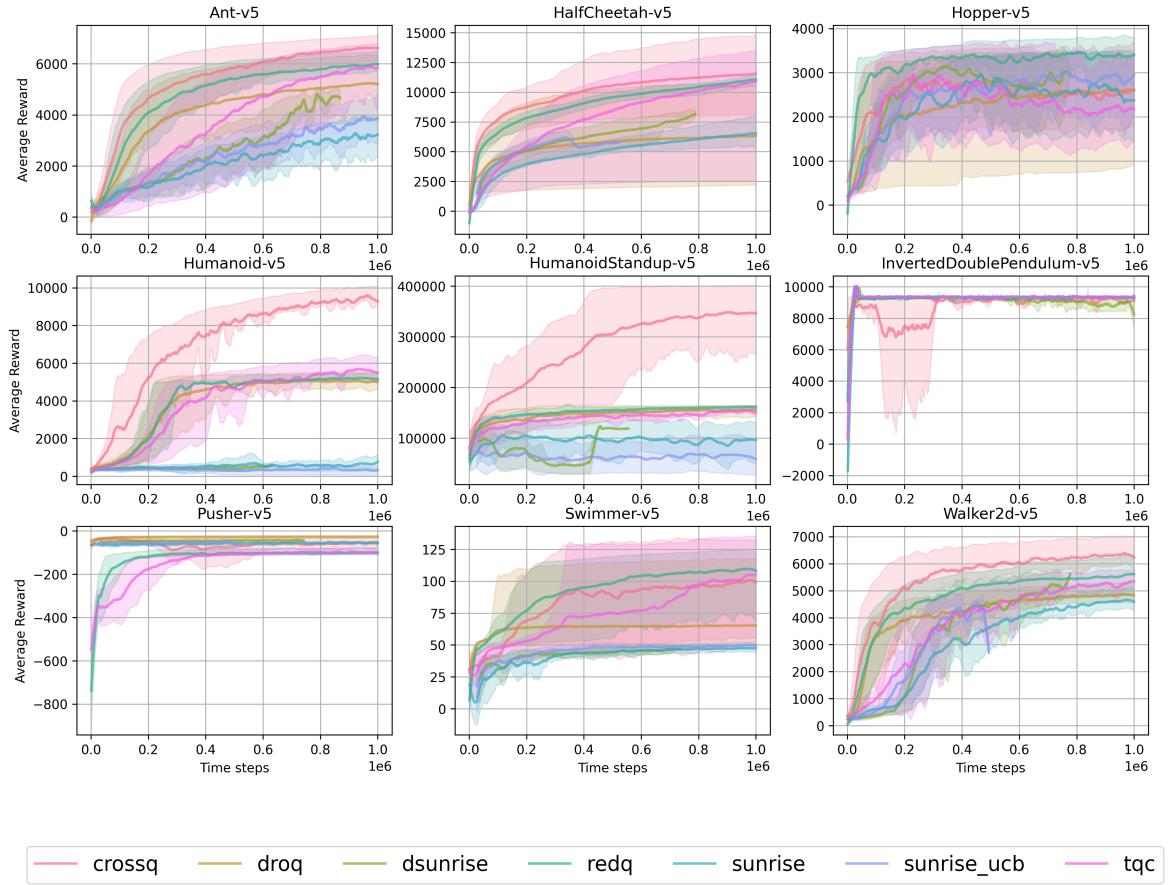


Figure 4.2: Sample Efficiency of the modern algorithms. The line is a smoothed average return across the 100 episodes made by each seed every 1000 time steps. The shaded area is a smoothed 70% confidence interval highlighted. Note that DSunrise experiments were not completed to 1 million steps, therefore their lines end at the last step completed.

Information on the implementations of the algorithms and the hyperparameters can be found in appendix A. We see across the algorithms that the newer algorithms outperform the older algorithms in both sample efficiency (how steep the line is) and asymptotic performance (the maximum value of the line). It can be seen that REDQ and TQC both outperform SAC even though they have the same foundations.

We can see that the newest algorithms are performing the best. Particularly CrossQ performs very well on the more complex environments like Humanoid and HumanoidStandup. It is also observed that CrossQ variance is a lot larger than the other algorithms which is most stark in the HumanoidStandup environment. This is likely due to the fact that CrossQ uses layer normalization which can lead to more variance in the learning process.

The older algorithms perform particularly poorly on the more complex environments like Humanoid and HumanoidStandup. Sunrise which is an ensemble of SAC agents also performs poorly on these environments. Sadly DSunrise also performs poorly on these environments which is understandable given that the base learning algorithm performs poorly. It is conceivable that future work could be done to switch the base algorithm to be CrossQ and use the stability of the ensemble to stabilize the learning process of CrossQ.

4.2.2 Computational Efficiency

The measure of computational efficiency is about measuring the amount of computation it takes to get a an agent to learn a certain task. As there is a variety of different algorithms the most general method is to measure the wall clock time it takes the algorithm to complete a certain number of environment steps.

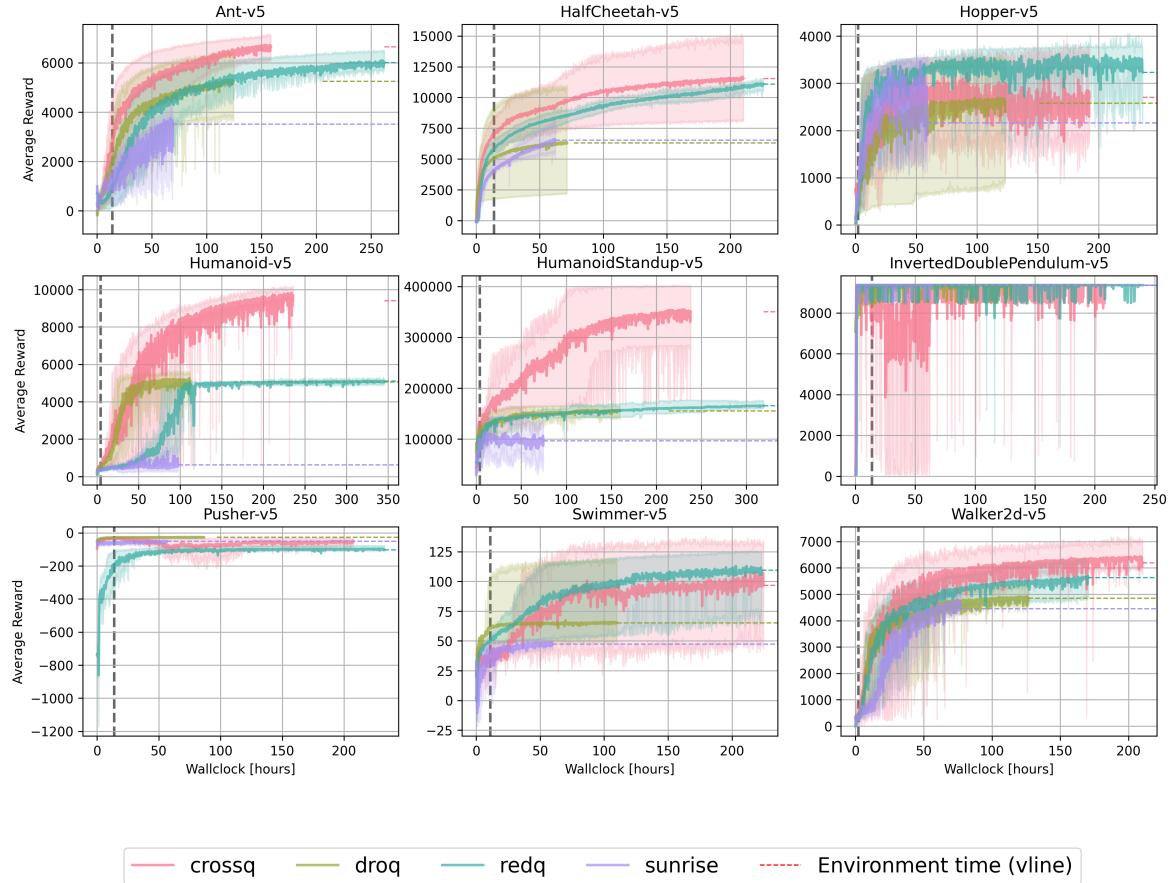


Figure 4.3: Wall clock time to get to 1 million environment time steps. Vertical lines is how long a million time-steps is in the environment. Horizontal line is the mean average reward at the end of the million steps of learning to help comparisons.

All of experiments were run on similar CPU only machines. See the appendix A for more information. The goal of graph is not to look at absolute time (however that is interesting to know) it is about the comparing the difference between algorithms given the same hardware. DSunrise is not included in this graph due to the fact that 1 million environment steps of training was not completed by the deadline of this report. Extrapolating from about 500,000 steps suggests that it would take about 250 hours to complete 1 million steps on the hardest environment HumanoidStandup.

It is interesting to note that the most modern algorithms REDQ and CrossQ are the longest running in terms of wall clock time. More importantly we can see that the method of REDQ having a lot of critics is computationally expensive and results in a wall clock time almost 4 times that of Sunrise which is also a modern ensemble method. This is due to the recent research focus on sample efficiency over computational efficiency.

Looking at the horizontal lines can give us a rough idea of whether we are learning in realtime, faster than realtime or slower than realtime. Importantly we can note that

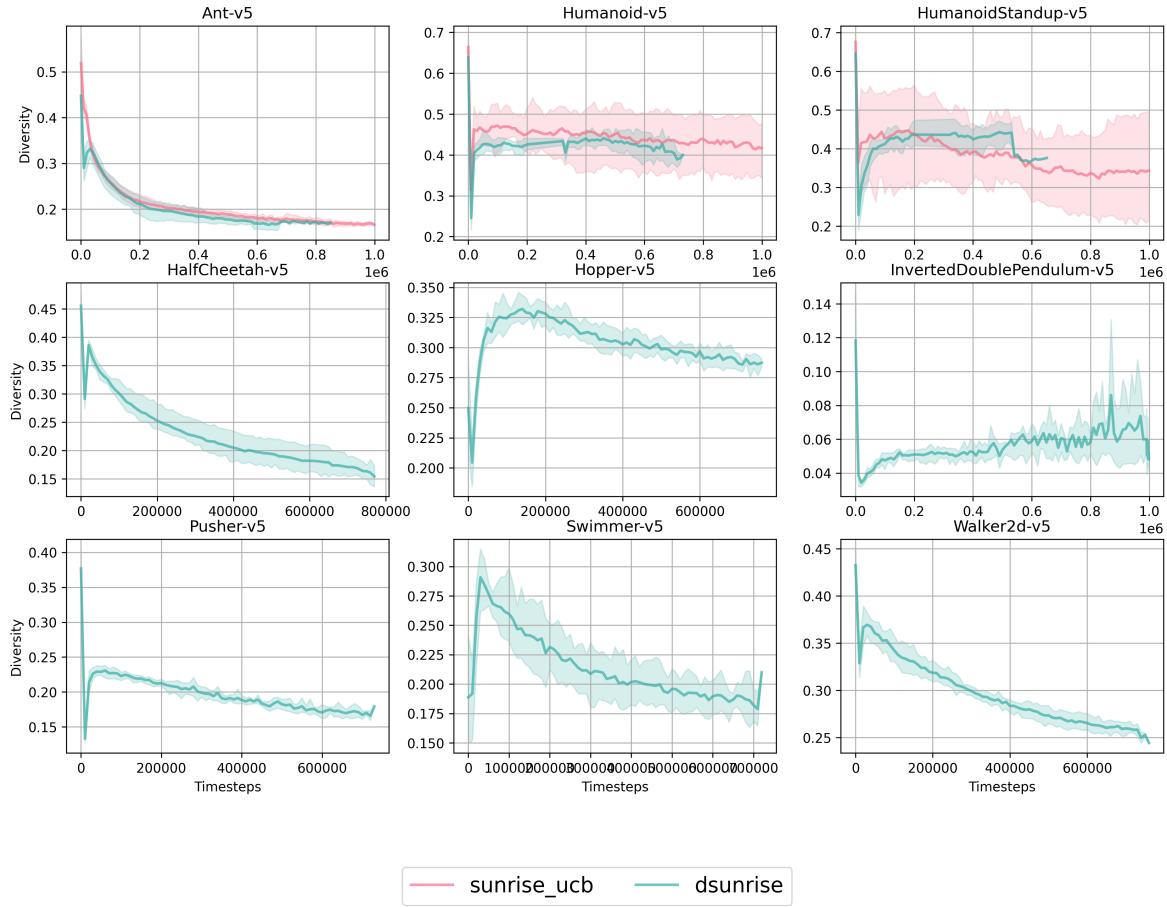


Figure 4.4: Diversity of the ensemble over time. The shaded area is a smoothed 70% confidence interval highlighted.

each experiments was run using minimal hardware that is within reach of small Single board computers such as a Raspberry Pi 5. We can see that for a more complicated task like HumanoidStandup we are learning significantly slower than realtime. Whereas the simpler environments are not so drastically slower than realtime. Given the hyperparameters are the same for all environments the only difference in environments is the action and state dimension as well as simulation time. Deeper analysis would have to be done to detangle the affect of simulation time and the action/state dimension in the wall clock time.

4.3 Ensemble diversity

The key idea of DSunrise is to ensure that the diversity of the ensemble of actors is maintained. As DSunrise can be directly compared to Sunrise, we can look at the diversity of the ensemble throughout training.

As one would expect the diversity of the ensemble decreases over time as the agents learn to do the task and converge towards the optimal policy. The goal of DSunrise is to maintain the diversity of the ensemble. It is maintaining this diversity by dynamically mutating the ensemble. We can observe that interestingly the diversity of the ensemble on average is the same as Sunrise which suggests that the dynamic mutation of the ensemble is not having a significant effect on the diversity of the ensemble. As mentioned in further work minimal hyperparameter tuning was done on the mutation parameters.

Looking at different environments we can see different scales of diversity. It is roughly correlated with the complexity of the environment and how well the agent learns. Because the agent never learns in the Humanoid and HumanoidStandup environments they have a consistent diversity that is quite high (almost random). Furthermore as the environments have different action dimensions the diversity metric is not directly comparable. The normalization using max l2 norm means that the diversity is relative to the maximum possible diversity of the action space. Yet larger action spaces will have more ways in which to diverge. The current method of a fixed threshold across all environments and all timesteps may not be the best way to measure diversity.

The environments that learn until convergence like Pusher and InvertedDDoublePendulum have lower diversities. With an interesting phenomenon that the diversity of InvertedDoublePendulum gets higher and much more variable post convergence. Which could be the effect of the mutation slowly “eating” away at the learned policy. As retrain steps was zero for these experiments one could expect that a small amount of retraining would remove this increase in variability.

4.3.1 Changing the ensemble

In these experiments the removal check was conducted every 10,000 steps. It had 3 outcomes

- The ensemble was not changed
- An actor was removed from the ensemble
- An actor was mutated

The removal of actors from the ensemble is not designed to help with learning but instead to reduce the computational cost of the algorithm. In the experiments it was observed that the removal of actors almost never took place as the ensemble never converged enough to a similar policy. The reason could be that the agents were not converging on the optimal policy (likely), the mutation was too strong and without any retraining the policy was different enough (likely), or that the diversity measure was not sensitive enough (plausible).

env	prob of replacement
Ant-v5	0.08
HalfCheetah-v5	0.08
Hopper-v5	0.01
Humanoid-v5	0.00
HumanoidStandup-v5	0.01
InvertedDoublePendulum-v5	0.20
Pusher-v5	0.12
Swimmer-v5	0.12
Walker2d-v5	0.00

Table 4.1: Percent of checks that resulted in a replacement of an actor.

As seen in table 4.1 the average probability of a replacement is low but not negligible. It is interesting to observe that the more complex environments have fewer replacements than the simpler environments. It is reasonable given that the simpler environments are easier to learn and so will converge quicker and once it is near convergence the removal check will always be triggered.

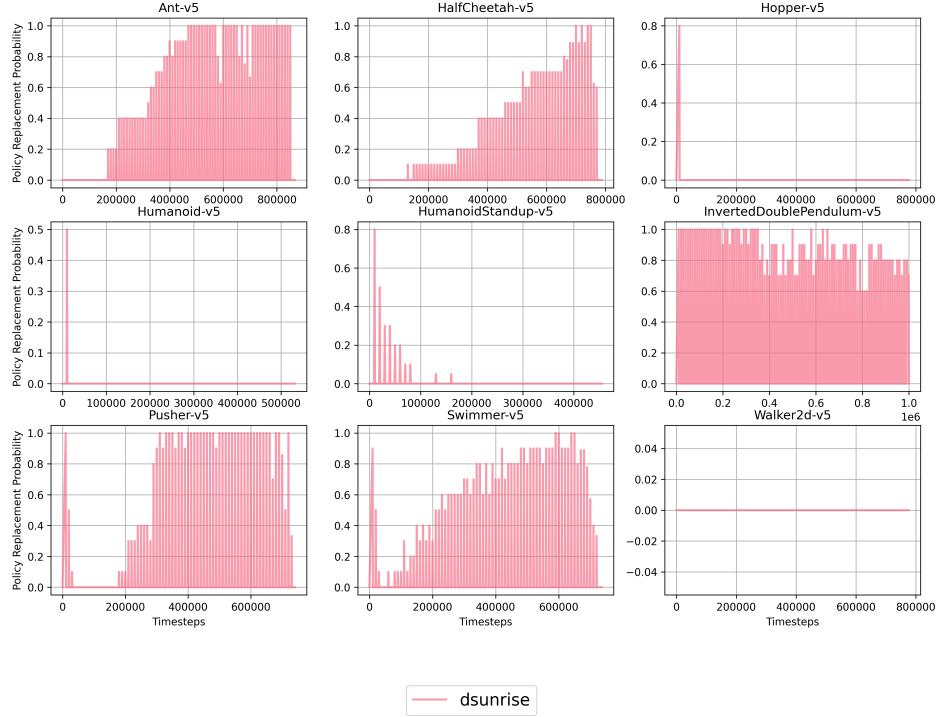


Figure 4.5: Ensemble replacement over time. Not the most rigorous as there are only 10 seeds to average over.

Looking at 4.4 we can see that the diversity of the InvertedDoublePendulum-v5 drops below the threshold of 0.2 almost immediately. Whereas the other environments have a diversities well above the threshold. Because InvertedDoublePendulum-v5 is below the threshold it will then replace many of the actors in the ensemble.

As one expects that more replacement will happen as training progresses and policies get closer together. 4.5 shows what our expectation are with a few peculiarities. Firstly there are 2 environments that have very low very low replacement rates that also have good performance. These are Hopper-v5 and Walker2d-v5. Secondly InvertedDoublePendulum-v5 has a very high replacement at the start of training and reduces as the effects of mutation take away its convergence and performance which we can see in 4.2.

Chapter 5

Conclusions

5.1 Further work

Finite time and resources have limited the scope of this report. There are many areas that could be explored further. I will outline some of the areas that I would have liked to explore further and would have done so if I had more time. Most of these are related to the new algorithm proposed in this report.

5.1.1 Different base learner

The current base learner used is Soft-Actor Critic (SAC) which is no longer a state of the art algorithm. As seen in the experiments section not only does SAC perform poorly in complex environments but so too does DSunrise. Using the ideas of DSunrise with a more modern base learner like CrossQ could greatly improve the performance of the algorithm. CrossQ is a state of the art algorithm and learns well although can be unstable. An ensemble can provide stability to the learning process. CrossQ is also very similar to SAC so the changes to the algorithm and implementation would be minimal.

5.1.2 Hyperparameter tuning

There are new hyper-parameters that are introduced in the new algorithm. Given limited time minimal exploration of these hyper-parameters was done once the algorithm was working reasonably well. It is plausible that tweaking and tuning these hyper-parameters could lead to notable improvement in the performance of the algorithm. The hyper-parameters are introduced in 3.4. I believe of most interest is exploring the trade off between regular replacement and mutation of these ensemble learners with the stability of the algorithm.

5.1.3 Explore the effect of retraining the newly instantiated actors

When the actor is slated for removal it is mutated and then checked for its diversity from the other actors. Retraining the actor before it is added back to the ensemble could lead to better performance and "targeted exploration". Specifically a subset of the replay buffer that is made up of high reward transitions, high error transitions and random transitions.

5.1.4 Dynamic diversity measure

The current diversity measure is static and is a measure of the distance between the actions selected by the actors normalized by the maximum possible distance between two actions.

The distance is averaged across a sample of states from the replay buffer. Then if diversity is below a threshold it is marked for "removal". There are two ways this could be changed. Firstly the diversity measure could be dynamic in that it is calculated based on the current diversity of the ensemble. Secondly the diversity measure could take into account the more so than the average distance between the actions of the closest policy. Instead it could incorporate the distance between all the policies, or the variation in distance of actions from the state space sample. This could allow the algorithm to better identify when a actor is no longer useful to the ensemble.

5.1.5 Add in the ability to add in new actors

Once a mutated actor is too close to the other actors it is removed from the ensemble. Meaning it triggers the shrinking of the ensemble. It is plausible that instead of removing the actors we could add in completely new actors that are warmed up on the most recent high error transitions. This would allow the ensemble to grow and explore areas that it is struggling with. Furthermore it would add potential for the ensemble to work in a continual learning environment where concept drift is a problem.

5.1.6 Diversity in the objective

adding a diversity-promoting term to the policy objective. For example, if you have N policies $\{\pi_{\phi_i}\}_{i=1}^N$, you can encourage diversity by penalizing similarity between policies. One common approach is to add a term based on the Kullback-Leibler (KL) divergence between the current policy and the other ensemble members:

$$\text{Diversity term: } \lambda \sum_{j \neq i} D_{\text{KL}} \left(\pi_{\phi_i}(\cdot | s_t) \| \pi_{\phi_j}(\cdot | s_t) \right)$$

So the updated policy objective for each policy π_{ϕ_i} becomes:

$$\mathbb{E}_{a_t \sim \pi_{\phi_i}} [\alpha \log \pi_{\phi_i}(a_t | s_t) - Q_{\theta}(s_t, a_t)] + \lambda \sum_{j \neq i} D_{\text{KL}} \left(\pi_{\phi_i}(\cdot | s_t) \| \pi_{\phi_j}(\cdot | s_t) \right)$$

where λ controls the strength of the diversity regularization.

5.1.7 Different evaluation get_action method

The current method to get an action from the ensemble during evaluation is to take the mean of the actions from the ensemble. This is a simple and effective method and is used in Sunrise. However it is plausible that a more sophisticated method could be used to get the actions from the ensemble. Firstly could be a weighted sum that would weight the actions based on the performance of the actor (either historic or from critics).

5.1.8 Computational efficiency

A drawback of DSunrise and any ensemble based method is the computational efficiency. Modern algorithms like CrossQ [6] use layer normalization to reduce the computational cost of the algorithm and speed up wall clock time. It is plausible that DSunrise could be made more computationally efficient by using a more efficient base learner like CrossQ.

5.2 Different algorithms

There are many ways of splitting the field of reinforcement learning. The binary splits we have looked at are model based and model free, value based and policy based and on policy and off policy. However the different solutions can merge the boundary between these splits. For example actor critic methods are explicitly both value based and policy based. Below is a table which helps summarize the differences between the modern algorithms.

Feature	DQN	SAC	PPO	TD3
Algorithm type	Value based	Actor-critic	Policy based (with actor critic style)	Actor-critic
Policy learning	Off policy	Off policy	On policy	Off policy
Learnt policy	Deterministic	Stochastic	Stochastic	Deterministic
Exploration strategy	ϵ -greedy	entropy maximisation	entropy maximisation	noise
Action space	Discrete	Continuous	Both	Continuous
Extra features	Experience replay, ensemble Q-functions	Experience replay, ensemble Q-functions	-	Experience replay

Table 5.1: Differences between modern deep reinforcement learning algorithms.

These models at their time of release were all considered state of the art for model free algorithms but due to the fast moving field are now transitioning to more foundational model which the latest algorithms are building on. These new algorithms increase not just the sample efficiency but also the asymptotic performance of the algorithms. In making these gains the newer algorithms might sacrifice some computational efficiency or stability.

The state of the art deep learning algorithms also having some differences. All of the algorithms that we have looked at however are all Actor Critic based method so all are Off policy, stochastic and use entropy maximization as the exploration strategy. There are still important differences between the algorithms as seen below.

Algorithm	Target networks	UTD $\gg 1$	Policy delay	No. critics	Special features
TQC	Yes	No	No	5	Truncated distributional critics
REDQ	Yes	Yes	Yes	10	Subset of critic ensemble for critic target, Minimization for Q-target
DroQ	Yes	Yes	Yes	2	Dropout layers, Minimization for Q-target
CrossQ	No	No	Yes	2	Layer normalization
Sunrise	Yes	No	No	3	Ensemble-based exploration
DSunrise (My algorithm)	Yes	No	No	Dynamic (start 10)	Dynamic ensemble with removal and instantiation of new actors

Table 5.2: Differences between state-of-the-art reinforcement learning algorithms.

5.3 Conclusion

Reinforcement Learning is a powerful framework for learning how to act. The recent combination of addition of deep learning has made reinforcement learning more powerful than ever. In this report I have covered the basics of reinforcement learning and the algorithms that are used to solve the problem. I have also explored some modern and state of the art algorithms which solve continuous control problems. In this exploration I have run some experiments that investigate the difference in sample efficiency and computational efficiency of the algorithms presented. Lastly I have proposed a new algorithm that builds on the idea of ensemble learners to make the ensemble inside Sunrise dynamic. This new algorithm, DSunrise shows some peculiar traits that have not yet been fully explored. It slows down the decrease in diversity while not sacrificing performance. Further work is needed to explore how to capitalize on the ability to maintain diverse base learners to better explore and search for the optimal policy.

The field of reinforcement learning is fast moving and new. There are still many unexplored ideas including the continuous control domain. The state of the art algorithms presented demonstrate the power of using different ideas from other areas of machine learning to improve the basic reinforcement learning algorithms framework. It is a very exciting space with potential for different combinations of idea to lead to great new algorithms.

Bibliography

- [1] Rāpoi Cluster Documentation. <https://vuw-research-computing.github.io/raapoi-docs/>.
- [2] SamsungLabs/tqc_pytorch: Implementation of Truncated Quantile Critics method for continuous reinforcement learning. <https://bayesgroup.github.io/tqc/>. https://github.com/SamsungLabs/tqc_pytorch.
- [3] BA, J. L., KIROS, J. R., AND HINTON, G. E. Layer Normalization, July 2016.
- [4] BELLEMARE, M. G., DABNEY, W., AND MUNOS, R. A Distributional Perspective on Reinforcement Learning, July 2017.
- [5] BELLMAN, R. *Dynamic Programming*. Princeton University Press, 1957.
- [6] BHATT, A., PALENICEK, D., BELOUSOV, B., ARGUS, M., AMIRANASHVILI, A., BROX, T., AND PETERS, J. CrossQ: Batch Normalization in Deep Reinforcement Learning for Greater Sample Efficiency and Simplicity, Mar. 2024.
- [7] CHEN, X., WANG, C., ZHOU, Z., AND ROSS, K. Randomized Ensembled Double Q-Learning: Learning Fast Without a Model, Mar. 2021.
- [8] FUJIMOTO, S., VAN HOOF, H., AND MEGER, D. Addressing Function Approximation Error in Actor-Critic Methods, Oct. 2018.
- [9] HAARNOJA, T., ZHOU, A., ABBEEL, P., AND LEVINE, S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor, Aug. 2018.
- [10] HAARNOJA, T., ZHOU, A., HARTIKAINEN, K., TUCKER, G., HA, S., TAN, J., KUMAR, V., ZHU, H., GUPTA, A., ABBEEL, P., AND LEVINE, S. Soft Actor-Critic Algorithms and Applications, Jan. 2019.
- [11] HIRAOKA, T., IMAGAWA, T., HASHIMOTO, T., ONISHI, T., AND TSURUOKA, Y. Dropout Q-Functions for Doubly Efficient Reinforcement Learning, Mar. 2022.
- [12] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural Networks* 2, 5 (Jan. 1989), 359–366.
- [13] IOFFE, S. Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models. In *Advances in Neural Information Processing Systems* (2017), vol. 30, Curran Associates, Inc.
- [14] KUZNETSOV, A., SHVECHIKOV, P., GRISHIN, A., AND VETROV, D. Controlling Overestimation Bias with Truncated Mixture of Continuous Distributional Quantile Critics, May 2020.

- [15] LEE, K. Pokaxpoka/sunrise, May 2025.
- [16] LEE, K., LASKIN, M., SRINIVAS, A., AND ABBEEL, P. SUNRISE: A Simple Unified Framework for Ensemble Learning in Deep Reinforcement Learning, June 2021.
- [17] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning, Feb. 2016.
- [18] LILLICRAP, T. P., HUNT, J. J., PRITZEL, A., HEESS, N., EREZ, T., TASSA, Y., SILVER, D., AND WIERSTRA, D. Continuous control with deep reinforcement learning, July 2019.
- [19] LIN, L.-J. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, USA, 1992.
- [20] MNIIH, V., BADIA, A. P., MIRZA, M., GRAVES, A., LILLICRAP, T. P., HARLEY, T., SILVER, D., AND KAVUKCUOGLU, K. Asynchronous Methods for Deep Reinforcement Learning, June 2016.
- [21] MNIIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing Atari with Deep Reinforcement Learning, Dec. 2013.
- [22] MNIIH, V., KAVUKCUOGLU, K., SILVER, D., RUSU, A. A., VENESS, J., BELLEMARE, M. G., GRAVES, A., RIEDMILLER, M., FIDJELAND, A. K., OSTROVSKI, G., PETERSEN, S., BEATTIE, C., SADIK, A., ANTONOGLOU, I., KING, H., KUMARAN, D., WIERSTRA, D., LEGG, S., AND HASSABIS, D. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (Feb. 2015), 529–533.
- [23] RAFFIN, A., HILL, A., GLEAVE, A., KANERVISTO, A., ERNESTUS, M., AND DORMANN, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research* 22, 268 (2021), 1–8.
- [24] SCHULMAN, J., LEVINE, S., MORITZ, P., JORDAN, M. I., AND ABBEEL, P. Trust Region Policy Optimization, Apr. 2017.
- [25] SCHULMAN, J., MORITZ, P., LEVINE, S., JORDAN, M., AND ABBEEL, P. High-Dimensional Continuous Control Using Generalized Advantage Estimation, June 2015.
- [26] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal Policy Optimization Algorithms, Aug. 2017.
- [27] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.
- [28] SUTTON, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984.
- [29] SUTTON, R. S. Learning to predict by the methods of temporal differences. *Machine Learning* 3, 1 (Aug. 1988), 9–44.
- [30] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning, Second Edition: An Introduction*, 2nd edition ed. Bradford Books, Nov. 2018.

- [31] SUTTON, R. S., MCALLESTER, D., SINGH, S., AND MANSOUR, Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In *Advances in Neural Information Processing Systems* (1999), vol. 12, MIT Press.
- [32] THOMPSON, J. 1jamesthompson1/REDQ, Apr. 2025.
- [33] THOMPSON, J. 1jamesthompson1/sunrise, May 2025.
- [34] THOMPSON, J. 1jamesthompson1/tqc_pytorch, Apr. 2025.
- [35] TODOROV, E., EREZ, T., AND TASSA, Y. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), IEEE, pp. 5026–5033.
- [36] TOWERS, M., KWIATKOWSKI, A., TERRY, J., BALIS, J. U., DE COLA, G., DELEU, T., GOULÃO, M., KALLINTERIS, A., KRIMMEL, M., KG, A., ET AL. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032* (2024).
- [37] VAN HASSELT, H., GUEZ, A., AND SILVER, D. Deep Reinforcement Learning with Double Q-learning, Dec. 2015.
- [38] WATCHERNYU. Watchernyu/REDQ, Mar. 2025.
- [39] WATKINS, C. *Learning from Delayed Reward*. PhD thesis, King's College London, England, May 1989.
- [40] WATKINS, C. J. C. H., AND DAYAN, P. Q-learning. *Machine Learning* 8, 3 (May 1992), 279–292.
- [41] WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (May 1992), 229–256.

Appendix A

A.1 Experiment details

All of the experiments were run using the same gymnasium version 1.1.1 using all of the version 5 environments. The environments were run using the default Mujoco settings. The source code for the different algorithms can be seen in the table below:

Table A.1: Source code for the different algorithms.

Algorithm	Source Code
SAC, PPO, DQN and TD3	Stable baselines3 [23]
TQC	My own updated copy of the authors implementation [34] [2]
REDQ	My own copy of the authors implementation [32] [38]
DroQ and CrossQ	Using SBX which is part of Stable Baselines 3 [23]
Sunrise	My own updated copy of the authors implementation [33] [15]
DSunrise	My own implementation of the algorithm built off Sunrise code. [33]

The experiments were run across multiple computers CPUs by using a grid computing facility at the School of Engineering and Computer Science at Victoria University of Wellington as well as the Raapoi HPC system [1]. The timed experiments were all run on Raapoi with CPU only. The CPUs are AMD EPYC Zen 3 models. The effect of the slightly different CPU speeds is mitigated as the 10 seeds will be run across a variety of CPU models. All experiments except dsunrise were run with only 2 CPU threads per job and about 10gb of memory (slightly higher for more complex environments like Humanoid). dsunrise was run on 6 CPU threads and more memory in an attempt to speed up the training process. It has little effect on the computational speeds.

The code used to run the experiments can be found at: [https://github.com/1jamesthompson1/ AIML440_code](https://github.com/1jamesthompson1/AIML440_code). With further discussion of source code in section A.3.

A.2 Experiment hyper-parameters

The hyper-parameters fro the different algorithms were all taken from the respective authors original implementation. No additional hyperparameter tuning was done.

The first table shows the hyperparameters for the older deep learning algorithms A.2 and the second table shows the hyperparameters for the state of the art deep learning algorithms A.4 the last table is for Sunrise and the new algorithm proposed.

Table A.2: Learning Hyperparameters for modern deep learning algorithms.

Parameter	SAC	TD3	PPO
Discount Factor (γ)		0.99	
Learning Rate (Actor & Critic)	0.0003	0.001	0.0003
Replay Buffer Size		10^6	
Batch Size	256	100	64
Activation Function		relu	Tanh
Critic Width	256	400 and 300	64
Critic Depth		2	
Actor Width	256	400 and 300	64
Actor Depth		2	
Optimizer		Adam	
Target Update Rate (τ)		0.005	N/A
Learning start delay	100	1000	N/A
Update-To-Data ratio (UTD)		1	
Policy Delay	1	2	1
Number of Critics		2	1
Algorithm Specific Parameters	N/A	target policy clip = 0.5 target policy noise = 0.2	n epochs = 10 gae- λ = 0.95 environment steps = 2048 clip range = 0.2

Table A.3: Learning Hyperparameters for sota deep learning algorithms.

Parameter	TQC	REDQ	DroQ	CrossQ
Discount Factor (γ)		0.99		
Learning Rate	0.001	0.0003	0.003	0.003
Replay Buffer Size		10^6		
Batch Size		256		
Activation Function		relu		
Critic Width	512	256	256	2048
Critic Depth	3		2	
Actor Width		256		
Actor Depth		2		
Target Update Rate (τ)		0.005		N/A
Adam β_1		0.9		0.5
Learning start delay	256	5000		N/A
Update-To-Data ratio	1	20		1
Policy Delay	1	20		3
Number of Critics	5	10		2
Algorithm Specific	n quantiles = 25 dropped quantiles = 2	sampled critics = 2 initial entropy = 0.2	dropout = 0.01 Layer normilisation	BatchNorm = BRN Momentum = 0.99 BRN Warmup = 10^5

Table A.4: Learning Hyperparameters for Sunrise and DSunrise.

Parameter	Sunrise	DSunrise
Discount Factor (γ)		0.99
Learning Rate		0.0003
Replay Buffer Size		10^6
Batch Size		256
Activation Function		<code>relu</code>
Critic Width		256
Critic Depth		2
Actor Width		256
Actor Depth		2
Target Update Rate (τ)		0.005
Adam β_1		0.9
Learning start delay	1000	10000
Update-To-Data ratio	1	1
Policy Delay	1	1
Ensemble start size	3	10
Actor temperature	0	0
Critic temperature	20	20
Bernoulli mean	0.5	0.5
UCB exploration	True	True
Algorithm Specific	N/A	<div style="margin-left: 20px;"> diversity_threshold = 0.2 diversity_critical_threshold = 0.1 performance_gamma = 0.99 window_size = 1000 noise = 0.1 retrain_steps = 0 removal_check_frequency = 10,000 removal_check_buffer_size = 10,000 </div>

A.3 Source code

This project involved a non-trivial amount of code to conduct the various baseline experiments and to implement the new algorithm. As mentioned in the experiments appendix for a few of the modern algorithms I used my own fork which was used to update the original code to work with the latest version of gymnasium and MuJoCo. These forks will be omitted from this discussion due to their triviality. The only code that is discussed here is the code that was used to run the experiments and the code that was used to implement the new algorithm DSunrise.

A.3.1 Experiment code

The experiment code is available at https://github.com/1jamesthompson1/AIML440_code. This repository has two sections that may be of interest.

Firstly is the `running-baseline-experiments` directory. This contains the bash and

python scripts that were used to run all but the DSunrise experiments. The scripts are designed to be run on a grid computing system. Particularly run as inner scripts to a `slurm.sl` script that is called by a batch job. The scripts are then called by `slurm.sl` and passed the `SLURM_ARRAY_TASK` and extra parameters. The `slurm.sl` script also defines the `OUTPUTDIR` which is where the results of the experiments are saved. There is also a `results.ipynb` notebook that is used to process the results of the experiments and generate the figures in the report. *These submission scripts and results notebooks are not designed to be used by anyone else and are specific to my experiment and cluster setup. However I believe they would still be of interest to anyone wanting somewhere to start.*

Secondly is the `my-algorithm-implementation` directory. None of the implementations in these directories were finished as each ran into enough roadblocks to prevent completion.

The `snac` directory contains the implementation of the Soft n Actor Critic algorithm. The SNAC algorithm was a novel algorithm that was designed to extend the SAC algorithm to use an ensemble of actors. The algorithm never reached a point of learning that was on par with some of the other modern algorithms. Therefore it was shelved in favour of other ideas that involved adapting SUNRISE. A write up to introduce the algorithm can be found here `SnAC.pdf`.

The `SUNRISE` and `DSUNRISE` directory. Given the original authors implementation of SUNRISE was implemented with tight coupling to an very outdated version of the RLkit library I wanted to re-implement the algorithm in a standalone way. The result of the attempt can be found in the `SUNRISE` directory. The implementation is close to working but there is a bug in the code that prevents it from learning after a few thousand steps (i.e it just plateaus at just above random actions). After weeks of bug searching and fixing I decided to shelve the implementation and instead use the original authors implementation of SUNRISE.

A.3.2 DSunrise code

The code for my DSunrise implementation is found at <https://github.com/1jamesthompson1/dsunrise>. The code started off as a fork of the original SUNRISE implementation but has since diverged and added in a new algorithm. Due to the nature of the implementation changes to the code base are spread across many files. However the two most important files are the `dsunrise.py` file and the `sunrise_ensemble.py` file. The `dsunrise.py` file is the entry point for the DSunrise algorithm and from there calls all relevant parts of the algorithm. The `readme` file in the repository contains instructions on how to run the algorithm and how to use the code.