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 2, 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 has three sections. Firstly an Introduction of reinforcement learning and basic algorithms. Secondly I will evaluate modern algorithms. Lastly I will describe a new algorithm and compare it to the state of the art models.

Contents

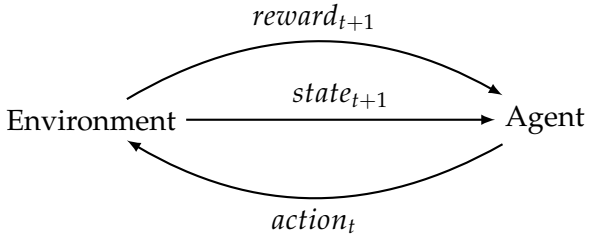
Chapter 1

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)." **[?]**

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:

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

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* [**?**]. The Markov property states that including previous states in the conditional wont change the probability of next state reward tuple. In other words $\mathcal{P}(s_{t+1}, r_{t+1}, s_t, a_t) = \mathcal{P}(s_{t+1}, r_{t+1}, s_t, a_t, s_{t-1}, s_{t-2}, ..., s_0)$. 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 **??** it will have to learn about the MDP from the information in a trajectory **??**.

$$S_1, A_1, R_2, \ldots S_n, A_n, R_{n+1}, \ldots \tag{1.2}$$

In this trajectory **??** 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 **??** can be formalised to be the return G.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots = \sum_{k=t}^{\infty} \gamma^{t-1} R_{t+1} \tag{1.3}$$

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 predicable 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.

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

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.

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

We can understand how to compute the expectation of the value function by looking at the Bellman equation [**?**]. 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:

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

$\pi(a|s)$ 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 [**?**].

The maximising part of the reinforcement problem can be solved by the finding optimal actions. We can define both $v_\star = \max_\pi v_\pi(s)$ and $q_\star = \max_\pi q_\pi(s,a)$ as the optimal value given optimal actions afterwards. If we can find $q_\star$ then the problem is solved as we could make a policy $\pi_\star$ that is greedy with respect to $q_\star$. 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 funtion 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 [**?**] and Temporal Difference learning [**?**] [**?**].

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

Arbitrarily initialize $\pi(s) \in \mathcal{A}(s)$ and $Q(s,a) \in \mathbb{R}$
Returns$(S,A) \leftarrow$ empty list
for each episode **do**
    Arbitrarily choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}$
    Generate an episode by following π
    $G \leftarrow 0$
    for each step of the episode $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$
        if (S_t, A_t) does not appear earlier in the episode **then**
            Append G to Returns(S_t, A_t)
            $Q(S_t, A_t) \leftarrow$ average(Returns(S_t, A_t))
            $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
        end if
    end for
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 **??**. As the update is using its own estimate it is *bootstrapping*. Because of bootstrapping it can work with both episodic and continuing environments.

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

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

Algorithm 2 TD(0)

Initialize $Q(s, a)$ arbitrarily and set $Q(\text{terminal-state}) = 0$
Initialize S
while not converged **do**
    Take action A, observe R, S'
    Choose $A' \in \mathcal{A}(S')$ using policy derived from Q
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
    $S \leftarrow S'$; $A \leftarrow A'$
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 [**?**].

2.1.2 Policy Based Methods

A different way of the 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 [**?**]. To use policy gradient methods you need to parametize 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 [**?**]. 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 [**?**].

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

Require: A differentiable policy parameterization $\pi(a|s, \theta)$
Require: Step size $\alpha > 0$
Initialize policy parameter $\theta \in \mathbb{R}^d$ (e.g., to 0)
loop
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    for $t = 0, 1, \ldots, T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
        $\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \theta)$
    end for
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 [**?**]. 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 [**?**] [**?**] . It works almost the same except the action value function is updated using the best possible action taken at the next state.

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

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 [**?**][**?**]. 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 [**?**] [**?**], 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 approximiator$Q$

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
Initialize action-value function Q_θ with random weights θ
Initialize target action-value function $\hat{Q_{\theta^-}}$ with weights $\theta^- \leftarrow 0$
for episode in range(1, M) **do**
    Initialize sequence $s_1 \leftarrow \{x_1\}$ and preprocessed sequence $\phi_1 \leftarrow \phi(s_1)$
    for each step t in episode **do**
        With probability ϵ, select a random action a_t
        Otherwise, select $a_t \leftarrow \arg\max_a Q(\phi(s_t), a; \theta)$
        Observe reward r_t and next state x_{t+1}
        Set $s_{t+1} \leftarrow (s_t, a_t, x_{t+1})$ and preprocess $\phi_{t+1} \leftarrow \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch B of size K of transitions from $\mathcal{D}$
        for each transition $(\phi_j, a_j, r_j, \phi_{j+1})$ in B **do**

$$y_j \leftarrow \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-), & \text{otherwise} \end{cases}$$

            Compute loss $L_j \leftarrow (y_j - Q(\phi_j, a_j; \theta))^2$
        end for
        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)
        if t mod C == 0 **then**
            Update target network: $\theta^- \leftarrow \theta$
        end if
    end for
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 [**?**], 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 [**?**].

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) [**?**].

First, similar to DQN [**?**, **?**], 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:

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

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

The final SAC algorithm [**?**] 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.

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
Initialize policy π_ϕ with random weights ϕ
Initialize both action-value function Q_{θ_i} with random weights θ_i
Initialize $\bar{\theta}_i \leftarrow \theta_i$ for $i \in \{1, 2\}$
repeat
  for S steps **do**
    select action: $a_t \sim \pi_\phi(a_t|s_t)$
    Observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  end for
  for UTD times **do**
    Update Q-functions: $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$
    Update policy: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
    Update target Q-networks: $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
  end for
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 [**?**].

Between 2018 and 2019, when SAC was first introduced, it outperformed other popular methods such as TD3 [**?**], DDPG [**?**], and PPO [**?**] 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 [**?**]. Yet the most popular method is that of PPO [**?**], 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 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 use generalised advantage estimation which is an exponentially weighted sum of TD errors [**?**], as it better balances bias and variance. GAE is defined as:

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

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 [**?**].

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 **??**. This leads to this objective:

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

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 [**?**] 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, environemt steps T, neural network function approximator V, neural network function approximator π, value function coefficient c_1 and entropy coefficient c_2

Initialize policy π_θ and value function V_ϕ with random weights θ and ϕ
$\theta_{old} \leftarrow \theta$
repeat
  for actor = 1, 2, . . . , N **in parallel do**
    Run policy $\pi_{\theta_{old}}$ in environment for T timesteps
    Collect trajectory $(s_t, a_t, r_t, s_{t+1})_{t=0}^{T-1}$
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ using GAE
    Compute returns $\hat{R}_t \leftarrow \sum_{i=t}^{T-1} \gamma^{i-t} r_i$
  end for
  Combine data from all actors: $\mathcal{D} = \{(s_t^n, a_t^n, \hat{A}_t^n, \hat{R}_t^n)\}$ for $n = 1, \ldots, N$ and $t = 1, \ldots, T$
  for epoch = 1, 2, . . . , M **do**
    Shuffle data $\mathcal{D}$ and split into minibatches of size $K \leq NT$
    for each minibatch $\mathcal{B} \subset \mathcal{D}$ **do**
      For each $(s, a, \hat{A}, \hat{R}) \in \mathcal{B}$:
        Compute ratio $r(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$
        Compute clipped surrogate objective:
        $L_{clip}(\theta) = \min(r(\theta)\hat{A}, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A})$
        Compute value function loss:
        $L_{VF}(\phi) = (V_\phi(s) - \hat{R})^2$
        Compute entropy bonus:
        $S[\pi_\theta](s) = -\sum_a \pi_\theta(a|s) \log \pi_\theta(a|s)$
        Total objective: $L(\theta, \phi) = \mathbb{E}[L_{clip}(\theta) - c_1 L_{VF}(\phi) + c_2 S[\pi_\theta](s)]$
        Update θ using SGD or Adam optimizer: $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta L(\theta, \phi)$
        Update ϕ using SGD or Adam optimizer: $\phi \leftarrow \phi + \alpha_\phi \nabla_\phi L(\theta, \phi)$
    end for
  end for
  $\theta_{old} \leftarrow \theta$
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 contiunous control setting [**?**]. 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 [**?**] and is a variant of DDPG [**?**] that builds on Double Q-Learning [**?**]. 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 similiar actions should have similar values.

The algorithm from the original paper [**?**] 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

Initialize critic networks Q_{θ_1}, Q_{θ_2}, and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{D}$
repeat
    increment timestep t
    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'
    Store transition tuple (s, a, r, s') in $\mathcal{D}$
    Sample mini-batch of K transitions (s, a, r, s') from $\mathcal{D}$
    Initialize critic loss $L_{\theta_1} \leftarrow 0$, $L_{\theta_2} \leftarrow 0$
    for $i = 1$ to K **do**
        Add noise to target action: $\tilde{a}_i \leftarrow \pi_{\phi'}(s_i') + \text{clip}(\epsilon_i, -c, c)$ where $\epsilon_i \sim \mathcal{N}(0, \tilde{\sigma})$
        Compute target Q-value: $y_i \leftarrow r_i + \gamma \min(Q_{\theta_1'}(s_i', \tilde{a}_i), Q_{\theta_2'}(s_i', \tilde{a}_i))$
        Accumulate critic losses: $L_{\theta_j} \leftarrow L_{\theta_j} + (y_i - Q_{\theta_j}(s_i, a_i))^2$ for $j = 1, 2$
    end for
    Update critics: $\theta_i \leftarrow \theta_i - \alpha_Q \nabla_{\theta_i}(L_{\theta_i}/K)$ for $i = 1, 2$
    if t mod d **then**
        Update ϕ by the deterministic policy gradient:
        $\nabla_\phi J(\phi) = K^{-1} \sum_i^K \nabla_a Q_{\theta_1}(s_i, a)|_{a=\pi_\phi(s_i)} \nabla_\phi \pi_\phi(s_i)$
        Update target networks:
        $\theta_i' \leftarrow \tau\theta_i + (1-\tau)\theta_i'$
        $\phi' \leftarrow \tau\phi + (1-\tau)\phi'$
    end if
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 **??**. These are:

Truncated Quantile Critics (TQC) [**?**] 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) [**?**] 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) [**?**] 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) [**?**] 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 [**?**] mitigates this by using multiple distributional critics and truncating the top quantile of the distributions.

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

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

It approximates the distribution with M number of quantiles. each predicting the value of the return at a given quantile θ^m_ψ.

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 [**?**] 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 [**?**] 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

Initialize replay buffer $\mathcal{D}$
Initialize policy π_ϕ and C critics Z_{ψ_i}, $Z_{\overline{\psi}_i}$ with random parameters ψ_i
Set $\alpha = 1$ and $\mathcal{H}_T = -\dim \mathcal{A}$
repeat
    Run policy π_ϕ in environment until termination or truncation
    Append trajectories $(s_t, a_t, r_t, s_{t+1})_{t=0}^{T-1}$ to $\mathcal{D}$
    for UTD times **do**
        Sample a batch B of size K from the replay $\mathcal{D}$
        Update α with gradient descent using

$$\nabla_\alpha \frac{1}{K} \sum_{(s,a)\in B} \log \alpha(-\log \pi_\phi(a|s) - \mathcal{H}_T)$$

        ϕ with gradient descent using

$$\nabla_\phi \frac{1}{K} \sum_{(s,a)\in B} \left(\alpha \log \pi_\phi(a|s) - \frac{1}{NM} \sum_{n=1}^{N} \sum_{m=1}^{M} \theta_{\psi_n}^m(s,a) \right)$$

        Update ψ_n with gradient descent using for $n = 1, \ldots, C$:

$$\nabla_{\psi_i} \frac{1}{KkCM} \sum_{(s,a)\in B} \sum_{m=1}^{M} \sum_{i=1}^{kC} \rho_{\tau_m}^H \left(y_i(s,a) - \theta_{\psi_n}^m(s,a) \right)$$

        $\overline{\psi}_n \leftarrow \beta\psi_n + (1-\beta)\overline{\psi}_n$ for $n \in [1..C]$
    end for
until stopping criteria is met

We can see that this algorithm is similar to that of SAC **??** 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 [**?**] 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 **??** 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 [**?**]. The authors reference implementation of the algorithm is

built of SAC **??** 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 **??** and SAC **??** do.

Pseudo code for the algorithm taken from the paper [**?**] 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

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

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

    for $i = 1, \ldots, C$ **do**
      Update ϕ_i with gradient descent using

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

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

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

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 **??** is that it requires a large number of critics to be able to handle a UTD greater than 1. Hiraoka et al [**?**] 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) [**?**].

Dropout layers [**?**] 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 [**?**] is used after the dropout layer which increases the effectiveness of the dropout layer.

As DroQ uses a smaller ensemble of critics it does to take a subset and instead uses all of the critics to create the critic target for critic updating. This is in contrast to REDQ **??** 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

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

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

    for $i = 1, \ldots, C$ **do**
      Update ϕ_i with gradient descent using

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

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

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

until stopping criteria is met

2.3.4 Cross Q-learning

The goal of both REDQ **??** and DROQ **??** 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 computational

cost which increases which increases linearly with the UTD. CrossQ [**?**] instead is just a simple improvement on SAC **??** which removes the target networks allowing it to use batch normalization [**?**]. This combination allows it to achieve higher sample efficiency than REDQ **??** and DROQ **??** 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 [**?**].

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 **??** which is:

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

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

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

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 normlization works as expected.

These changes are applied to SAC **??** 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 **??**. 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, environemt steps S, neural network function approximator Q, neural network function approximator π and initial entropy coefficient α.

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
Initialize policy π_ϕ with random weights ϕ
Initialize both action-value function Q_{θ_i} with random weights θ_i
repeat
  for S steps **do**
    select action: $a_t \sim \pi_\phi(a_t|s_t)$
    Observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  end for
  for UTD times **do**
    Update Q-functions: $\theta_i \leftarrow \theta_i - \lambda_{Q_{BRN}} \hat{\nabla}_{\theta_i} J_{Q_{BRN}}(\theta_i)$ for $i \in \{1, 2\}$
    Update policy: $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$
    Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
  end for
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' [**?**]. 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 [**?**]. 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 in the given update. An understanding of confident can be acieved by using the variance of the ensemble of Q-functions. The actual weight is achieved using the equation.

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

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 multipling 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 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 initilisation is simply using a differnet initilised 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 paramters. This mask is made using the Bernoulli distribution.

Put togather 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

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

Chapter 3

Dynamic ensemble of Soft Actor Critics

Dynamic ensemble of soft actor critic is a algorithm that applies some of the ideas from Sunrise [**?**] and SAC [**?**] to create a new algorithm that explores quickly and stabilizes in the long run. The DSunrise algorithm is the built off the sunrise algorithm with the addition of a dynamic ensemble. That is on a regular interval (say 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 is 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 polices. 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.

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

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

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
- $R_i^{(\tau)}$: Return of learner i in its τ-th episode
- 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
- $\boldsymbol{a}_i(s)$: Action vector of learner i in state s

The removal decision combines both metrics:

$$\text{remove}_i^{(t)} = \begin{cases} 1 & \text{if } \exists\, j \neq i \text{ such that } \dfrac{\sum_{s \in \mathcal{S}} \left| \boldsymbol{a}_i(s) - \boldsymbol{a}_j(s) \right|_2}{|\mathcal{S}| \cdot (|\mathcal{E}| - 1) \mu_D^{(t)}} < d_{\text{thre}} \text{ and } \text{performance}_i^{(t)} < \text{performance}_j^{(t)} \\ 1 & \text{if } \dfrac{\text{performance}_i^{(t)}}{\mu_P^{(t)}} < p_{\text{crit}} \\ 0 & \text{otherwise} \end{cases}$$

With:

- $\mu_P^{(t)} = \frac{1}{|\mathcal{E}|} \sum_{k=1}^{|\mathcal{E}|} \text{performance}_k^{(t)}$
- $\mu_D^{(t)} = \frac{1}{|\mathcal{E}|} \sum_{k=1}^{|\mathcal{E}|} \text{diversity}_k^{(t)}$
- p_{crit}: Critical performance threshold (≈ 0.5)
- d_{thre}: Diversity threshold (≈ 0.6)

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 $\mathcal{D}$, Reward threshold r_{top}, Error threshold e_{top}, Random sample ratio ρ_{random}

1: Select top 50% transitions by reward:

$$\mathcal{D}_{\text{reward}} \leftarrow \text{Top } \lfloor 0.5 \cdot |\mathcal{D}| \rfloor \text{ transitions by reward}$$

2: Select top 30% transitions by error:

$$\mathcal{D}_{\text{error}} \leftarrow \text{Top } \lfloor 0.3 \cdot |\mathcal{D}| \rfloor \text{ transitions by error}$$

3: Randomly sample 20% transitions from $\mathcal{D}$:

$$\mathcal{D}_{\text{random}} \leftarrow \text{Randomly sample } \lfloor 0.2 \cdot |\mathcal{D}| \rfloor \text{ transitions}$$

4: Combine subsets:

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

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

Algorithm 14 Instantiate New Base Learner

Require: Replay buffer $\mathcal{D}$, Ensemble $\mathcal{E}$, Mutation noise σ, Diversity threshold p_{crit}

1: Select a random base learner base $\in \mathcal{E}$

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

3: Instantiate new base learner:

$$\boldsymbol{\theta}_{\text{new}} \leftarrow \boldsymbol{\theta}_{\text{base}} + \epsilon \cdot \boldsymbol{\delta}$$

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

$$\text{replay subset} \leftarrow \textbf{Generate Replay Subset}(\mathcal{D}, r_{\text{top}}, e_{\text{top}}, \rho_{\text{random}})$$

5: Train the new base learner on the replay subset

6: **if** $\text{diversity}_{\text{new}} < p_{\text{crit}}$ **then**

7: Discard new base learner

8: **else**

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

10: **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 [**?**] with the addition of the dynamic ensemble. The algorithm is outlined in Algorithm **??**. A implementation of the algorithm can be found at `https://github.com/1jamesthompson1/AIML440_code/tree/main/my-algorithm-implementation/DSUNRISE`.

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.

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, Critical performance threshold p_{crit}, Diversity threshold d_{thre}, Critical diversity threshold d_{crit}, max number of removals r, removal check frequency R, warmup steps W and removal function $\text{remove}_i^{(t)}$

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity N
for $i \in \{1, 2, \ldots, E\}$ **do**
    Initialize critic $Q_{\theta_{i,j}}$ with random weights $\theta_{i,j}$ for $j \in \{1, 2\}$
    Initialize target critic $\bar{Q}_{\bar{\theta}_{i,j}} \leftarrow Q_{\theta_{i,j}}$ for $j \in \{1, 2\}$
    Initialize policy π_{ϕ_i} with random weights ϕ_i
end for
repeat
    for S steps **do**
        Collect E action samples $\boldsymbol{a}_t^{(i)} = \pi_\phi(s_t)$ for $i \in \{1, 2, \ldots, E\}$
        select action: $a_t \sim \pi_\phi(a_t|s_t)$
        // If not using UCB then simply randomly select an actor to complete an entire episode.
        Choose the action that maximizes the UCB exploration term:
        $a_t = \arg\max_a \left(Q_{\text{mean}}(s_t, a) + \alpha \cdot \sigma(Q_{\text{std}}(s_t, a))\right)$
        Observe reward r_t and next state s_{t+1}
        Sample bootstrap masks $m_t = \{m_{t,i} \sim \text{Bernoulli}(0.5) \text{ for } i \in \{1, ..., E\}\}$
        Store transition $(s_t, a_t, r_t, s_{t+1}, m_t)$ in $\mathcal{D}$
    end for
    if Size of $\mathcal{D}$ is less than W **then**
        Continue to next iteration without training of removal checks
    end if
    for UTD times **do**
        Sample a batch B of size K from $\mathcal{D}$
        for each agent i **do**
            Use Mask $m_{t,i}$ to select transitions from B for agent i
            Update Q-functions to minimize $(\text{sigmoid}(-\bar{Q}_{\text{std}}(s_{t+1}, a_{t+1}) * T) + 0.5)(Q_{\theta_i}(s_t, a_t) - r_t - \gamma \bar{V}(s_{t+1}))$
            Update policy to minimize $\mathbb{E}_{a_t \sim \pi_\phi}\left[\alpha \log \pi_\phi(a_t|s_t) - Q_\theta(s_t, a_t)\right]$
            Update entropy coefficient: $\alpha \leftarrow \alpha - \lambda \hat{\nabla}_\alpha J(\alpha)$
            Update target Q-networks: $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$
        end for
    end for
    if the current episode is a multiple of R **then**
        Calculate $\text{remove}_i^{(t)}$ for each base learner $i \in \{1, 2, \ldots, E\}$ using the removal check
        for worst r performing base learners **do**
            Instantiate new base learner using the instantiation algorithm
            Update T to reflect the new ensemble size
        end for
    end if
until stopping criteria is met

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 [**?**] library which provides a framework to use among other things some Mujoco [**?**] 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 theses 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 improvment to a model it must be better at either/both of these metrics will not 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.

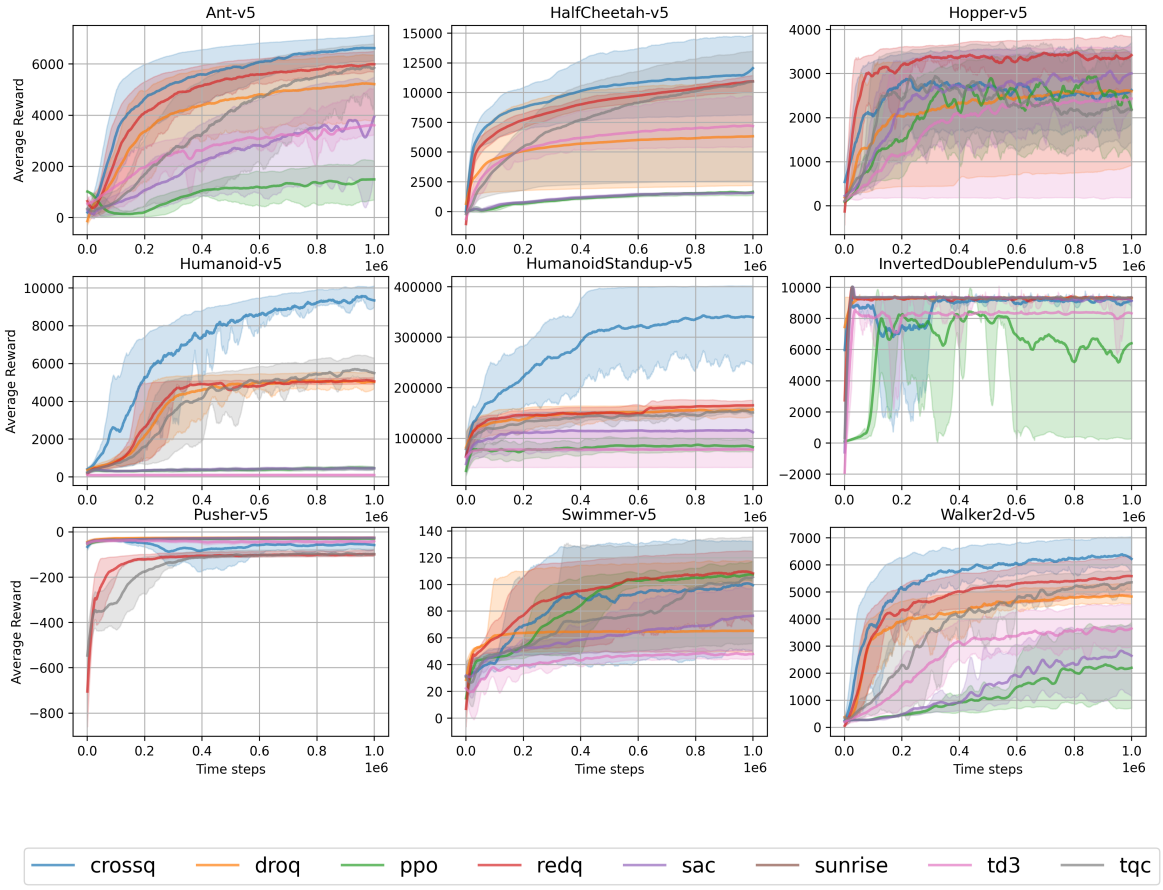


Figure 4.1: Sample Efficiency. The line is a smoothed average return across the 100 episodes made by each seed every 1000 times steps. The shaded area is a smoothed 70% confidence interval highlighted

Information on the implementations of the algorithms and the hyperparameters can be found in appendix **??**. 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 be seen that REDQ and TQC both outperform SAC even though they have the same foundations.

Simple analysis of the results here. Point out that CrossQ, DroQ and RedQ are noticably betters

...

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 mos general method is to measure the wall clock time it takes the algorithm to complete a certain number of environment steps.

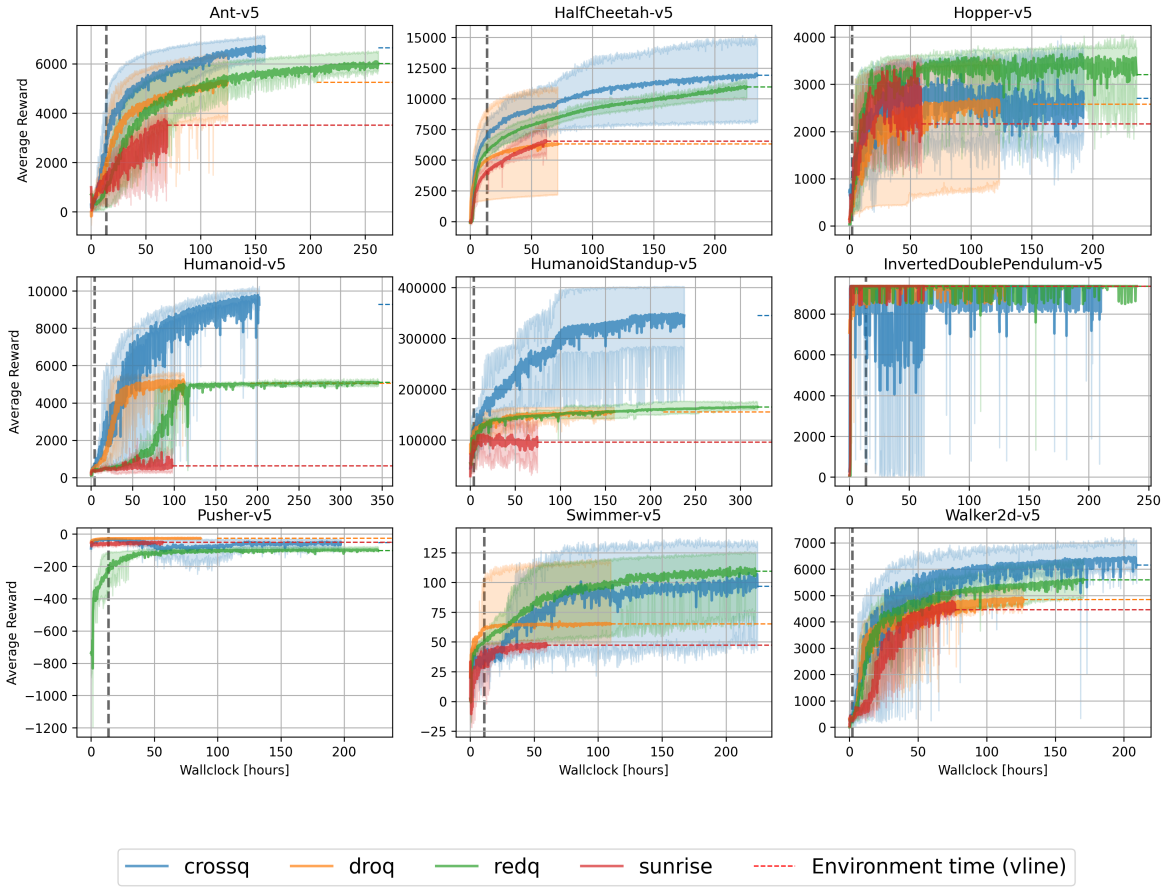


Figure 4.2: Wall clock time to get to 1 million environment time steps. Vertical lines is how long a million time-steps is in the environment. Horizontal line is the mean average reward at the end of the million steps of learning to help comparisons.

All of experiments were run on similar CPU only machines. See the appendix **??** 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.

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 alot 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.

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. 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 detangled the affect of simulation time and the action/state dimension in the wall clock time.

...

Chapter 5

Conclusions

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 and a novel extension to on other algorithms. In this exploration I have run some experiments that investigate the difference in sample efficiency and computational efficiency of the algorithms presented.

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 summarise 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	Yes	6	Ensemble-based exploration
DSunrise	Yes	No	Yes	Dynamic (start 20)	Dynamic ensemble with removal and instantiation of new actors

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

These algorithms represent some of the best performing algorithms in the field that excel in either their sample efficiency, computational efficiency or stability.

Dsunrise represents a new algorithm that takes a stable algorithm and extends it to ...

There is unknown potential in the applicability of current algorithms as well as unknown power in new algorithms. Some of the current limitations of the algorithms are the sample efficiency and the stability of the algorithms. Furthermore there is the larger problem of generalization, which is getting an agent to act intelligently on varied environments that it has never seen before with minimal to no new training.

The field of reinforcement learning is still in its infancy and there is much to be discovered. The combination of deep learning and reinforcement learning has opened up many possibilities and the future is bright and exciting.

Appendix A

A.1 Experiment details

All of the experiements 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 [**?**]
TQC	My own updated copy of the authors implementation [**?**] [**?**]
REDQ	My own copy of the authors implementation [**?**] [**?**]
DroQ and CrossQ	Using SBX which is part of Stable Baselines 3 [**?**]
Sunrise	My own updated copy of the authors implementation [**?**] [**?**]

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 [**?**]. 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.

The code used to run the experiments can be found at: `https://github.com/1jamesthompson1/AIML440_code`.

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 **??** and the second table shows the hyperparameters for the state of the art deep learning algorithms **??** 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-$\lambda = 0.95$ evironment 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 DESAC.

Parameter	Sunrise	DESAC
Discount Factor (γ)	0.99	
Learning Rate	0.0003	
Replay Buffer Size	10^6	
Batch Size	256	
Activation Function	`relu`	
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	
Update-To-Data ratio	1	1
Policy Delay	1	1
Ensemble start size	3	??
Actor temperature	0	??
Critic temperature	0	??
Bernoulli mean	0.5	??
UCB exploration	False	??
Algorithm Specific	N/A	??