
**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

James Thompson

Supervisor: Dr Aaron Chen

March 18, 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 reports will have 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

# Figures

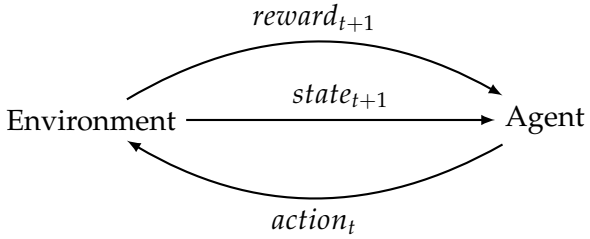
# Chapter 1

# Introduction

### 1.0.1 The reinforcement learning problem

Reinforcement learning is a framework of learning 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 will use 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)." [19]

### 1.0.2 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* [19]. 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.

The agent interaction 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

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

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.

"The cumulative sum of a received scalar signal" part of the reward hypothesis 1.0.1 can be formalised to be the return $G$.

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

$\gamma$ 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 $\pi$ 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 $\gamma$ 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 [19]. 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 $v_\pi$ 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 [1].

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 [19] and Temporal Difference learning [17] [18].

### 2.0.1 Traditional algorithms

#### 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 $\pi$
    $G \leftarrow 0$
    **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 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.

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

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($\infty$) 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($\lambda$) methods [18].

There are two issues with the Naive methods of TD and Monte Carlo methods 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 appropriator $\hat{Q}$.

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. $\epsilon$-greedy is a good example of this as it will take a random action with probability $\epsilon$ 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 both on-policy.

#### 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 [20]. To use policy gradient methods you need to parametize the policy $pi$ in any way as long as $pi(a|s, \theta)$ is differentiable with respect to $\theta$.

The first of these policy gradient methods was the Reinforce algorithm [24]. 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 [20].

**Algorithm 3** REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for $\pi_*$

**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**

### 2.0.2 Modern Deep learning algorithms

#### 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 [22] [23] . 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. To resolve this we can use a function approximator, which means we no longer need to store a mapping for every single state. The most powerful function approximators we have are neural networks [5]. Therefore we can apply a neural network as an approximator $\hat{Q}$ for $Q$. The algorithm we get is called Deep-Q-Learning [10][11]. 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 [10] [11], 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 $\gamma$, learning rate $\alpha$, update frequency $C$, exploration probability $\epsilon$. A neural network function approximiator$Q$

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity $N$
Initialize action-value function $Q_\theta$ with random weights $\theta$
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 $\epsilon$, 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 [8], 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 $\gamma$. 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 $\gamma_j$ but then updates $Q$ every $C$ updates.

This gives the network a reasonable opportunity to tend towards the target $\gamma_j$ 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.

**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, $\pi$, 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 [9].

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) [3].

First, similar to DQN [10, 11], 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 $\alpha$ determining the strength of this encouragement. Initially a hyperparameter in [3], the SAC algorithm was later modified to learn and adjust $\alpha$ throughout training [4]. This not only improves efficiency—since choosing an optimal temperature is task-dependent and non-trivial [4]—but also allows the policy to explore uncertain regions while being more exploitative in familiar areas.

The final SAC algorithm [4] modified for clarity is shown below:

**Algorithm 5** Soft Actor-Critic Algorithm

**Require:** Replay buffer size $N$, Minibatch size $K$, discount factor $\gamma$, learning rates $\lambda$, $\lambda_\pi$ and $\lambda_Q$, update frequency $C$, target smoothing coefficient $\tau$, Updates to Data ratio UTD, environemt steps $S$, neural network function approximator $Q$, neural network function approximator $\pi$ and initial entropy coefficient $\alpha$.

Initialize replay memory $\mathcal{D} \leftarrow \emptyset$ with capacity $N$
Initialize policy $\pi_\phi$ with random weights $\phi$
Initialize both action-value function $Q_{\theta_i}$ with random weights $\theta_i$
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 [4].

Between 2018 and 2019, when SAC was first introduced, it outperformed other popular methods such as TD3 [2], DDPG [7], and PPO [15] in continuous control benchmarks.

### Proximal Polixy Optimizatin

One of the problems that can happen with policy changes is that a small change which results in a different actions being taken can have large consequences. A natural idea is to try and limit the changes that can happen to 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 guranteed to improve) and resulted in the TRPO algorithm [13]. Yet the most popular method is that of PPO [15], 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 [14], 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 $\delta_t^V = r_t + \gamma V(s_t + 1) - V(s_t)$. $\gamma$ is the usual discounting factor but this is expanded on with $\lambda$ 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 2.0.2. 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 [15] with some modifications to make it clearer.:

**Algorithm 6** PPO Algorithm, Actor Critic style

**Require:** Clipping amount $\epsilon$, Minibatch size $K$, epoch $M$, discount factor $\gamma$, learning rates $\alpha_\theta$ and $\alpha_\phi$, number of actors $N$, environemt steps $T$, neural network function approximator $V$, neural network function approximator $\pi$ and value function coefficient $c_1$ and entropy coefficient $c_2$

Initialize policy $\pi_\theta$ and value function $V_\phi$ with random weights $\theta$ and $\phi$
$\theta_{old} \leftarrow \theta$
**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 $\theta$ using SGD or Adam optimizer: $\theta \leftarrow \theta + \alpha_\theta \nabla_\theta L(\theta, \phi)$
        Update $\phi$ 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

### Twin Delayed Deep Deterministic Policy Gradient

Twin Delayed Deep Deterministic Policy Gradient [2] is a variant of DDPG [6] that builds on Double Q-Learning [21]. It is a actor critic method that uses experience replay, target networks and policy noise to stabilise learning. The main idea 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 to reduce the overestimation bias that can happen in Q-learning methods.

The policy noise 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 [2] with slight modifications for clarity:

**Algorithm 7** Twin Delayed Deep Deterministic Policy Gradient (TD3)

**Require:** Replay buffer size $N$, Minibatch size $K$, discount factor $\gamma$, learning rates $\alpha$, $\alpha_\pi$ and $\alpha_Q$, target smoothing coefficient $\tau$, target policy smoothing noise $\sigma$, target policy noise clip $c$, target policy noise $\tilde{\sigma}$, target policy noise clip $c$, target policy noise $\tilde{\sigma}$, target policy noise clip $c$, update frequency $d$

Initialize critic networks $Q_{\theta_1}$, $Q_{\theta_2}$, and actor network $\pi_\phi$ with random parameters $\theta_1$, $\theta_2$, $\phi$
Initialize target networks $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{D}$
**repeat**
    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 $\phi$ 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

# Chapter 3

# Conclusions

Reinforcement Learning is a powerful framework for learning how to act. With the recent advancements with deep learning and the resultant Deep reinforcement learning algorithms have shown that the potential of the framework is vast. In this report I have covered the basics of reinforcement learning and the algorithms that are used to solve the problem. I have also evaluated the state of the art algorithms and shown that they are capable of solving complex problems.

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.

| 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** | $\epsilon$-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 3.1: Differences between modern deep reinforcement learning algorithms.

The presented models have been used to successfully solve many problems. For example SAC derivatives have been used to teach a 4 legged robot to walk in just 20 minutes [16]. Or a scaled up PPO algorithm beating world champions at Dota 2 [12]. These are just two examples of the power of reinforcement learning, which gives credit and support to Richards Sutton's reward hypothesis [19].

The possibilities of future work are vast and varied. There is unknown potential in applying the current algorithms to new problems. There is also the potential of creating new algorithms that improve on the current state of the art. 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 generalisation and how different on an environment can agent act well without retraining.

# Bibliography


[1] BELLMAN, R. *Dynamic Programming*. Princeton University Press, 1957.

[2] FUJIMOTO, S., VAN HOOF, H., AND MEGER, D. Addressing Function Approximation Error in Actor-Critic Methods, Oct. 2018.

[3] 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.

[4] 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.

[5] HORNIK, K., STINCHCOMBE, M., AND WHITE, H. Multilayer feedforward networks are universal approximators. *Neural Networks 2*, 5 (Jan. 1989), 359–366.

[6] 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.

[7] 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.

[8] LIN, L.-J. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, USA, 1992.

[9] MNIH, 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.

[10] MNIH, V., KAVUKCUOGLU, K., SILVER, D., GRAVES, A., ANTONOGLOU, I., WIERSTRA, D., AND RIEDMILLER, M. Playing Atari with Deep Reinforcement Learning, Dec. 2013.

[11] MNIH, 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.

[12] OPENAI, BERNER, C., BROCKMAN, G., CHAN, B., CHEUNG, V., DEBIAK, P., DENNISON, C., FARHI, D., FISCHER, Q., HASHME, S., HESSE, C., JÓZEFOWICZ, R., GRAY, S., OLSSON, C., PACHOCKI, J., PETROV, M., PINTO, H. P. D. O., RAIMAN, J., SALIMANS, T., SCHLATTER, J., SCHNEIDER, J., SIDOR, S., SUTSKEVER, I., TANG, J., WOLSKI, F., AND ZHANG, S. Dota 2 with Large Scale Deep Reinforcement Learning, Dec. 2019.

[13] SCHULMAN, J., LEVINE, S., MORITZ, P., JORDAN, M. I., AND ABBEEL, P. Trust Region Policy Optimization, Apr. 2017.

[14] SCHULMAN, J., MORITZ, P., LEVINE, S., JORDAN, M., AND ABBEEL, P. High-Dimensional Continuous Control Using Generalized Advantage Estimation, June 2015.

[15] SCHULMAN, J., WOLSKI, F., DHARIWAL, P., RADFORD, A., AND KLIMOV, O. Proximal Policy Optimization Algorithms, Aug. 2017.

[16] SMITH, L., KOSTRIKOV, I., AND LEVINE, S. A Walk in the Park: Learning to Walk in 20 Minutes With Model-Free Reinforcement Learning, Aug. 2022.

[17] SUTTON, R. S. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst, 1984.

[18] SUTTON, R. S. Learning to predict by the methods of temporal differences. *Machine Learning 3*, 1 (Aug. 1988), 9–44.

[19] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning, Second Edition: An Introduction*, 2nd edition ed. Bradford Books, Nov. 2018.

[20] 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.

[21] VAN HASSELT, H., GUEZ, A., AND SILVER, D. Deep Reinforcement Learning with Double Q-learning, Dec. 2015.

[22] WATKINS, C. *Learning from Delayed Reward*. PhD thesis, King's College London, England, May 1989.

[23] WATKINS, C. J. C. H., AND DAYAN, P. Q-learning. *Machine Learning 8*, 3 (May 1992), 279–292.

[24] WILLIAMS, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning 8*, 3 (May 1992), 229–256.