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 14, 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. In this report I will start with a introduction of reinforcement learning and explore some common modern algorithms. In the second part I will evaluate these common algorithms on a varied benchmark. Lastly I will propose extensions that improve the performance on introduced benchmarks.

Contents

Figures

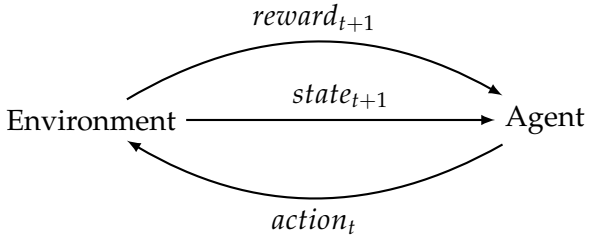
Chapter 1

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 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 situation is, the more reward the better, always. Lastly the action is sent to the environment and the environment will in turn return the next state that the action has taken the agent. 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).” [16]

1.0.2 Formalism

The reinforcement problem can be formalised as a Markov Decision Process. The MDP is a collection of states, actions and rewards along with a transition function which states that probability of the next reward and state given a state and reward.

$$Pr\left\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\right\}$$

This function completely characterises the dynamics of the environment. This 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* [@suttonReinforcementLearningSecond2018], which states including previous states in the conditional wont change the probability as the current state will have already captured that information. In interaction with the agent the MDP will generate a sequence that looks like:

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

In the case above we have an infinite sequence and these are continuing tasks as they have no end. Alternatively you could have episodic tasks that have a start and ending with a terminal state. "The cumulative sum of a received scalar signal" part of the reward hypothesis can be formalised to be the return G.

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

γ is called the discounting factor. It is usually added for two reasons. Firstly because it means that the return won't be infinite which simplifies it mathematically. 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 decision agent will use policy π which provides the probability of taking action a given state s. This policy could be deterministic or stochastic. To get the policy the agent needs an understanding of value of its current state. This is called the value function and it is a expectation of the future rewards.

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

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

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

These functions are closely related and have a large amount of connected definitions this one:

$$q_\pi(s, a) = \mathbb{E}\left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a\right]$$

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 $q_\star$.

This form explicitly learning a value function then implicitly getting a full policy or just needed 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 [16] and Temporal Difference learning [14] [15].

2.0.1 Traditional algorithms

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, therefore it has the advantage that we can force it to explore just the state space we are interested in, by starting it in states we want to know 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 this 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 one can see how you could easily make it just go on forever in an continuing environment.

Algorithm 2 TD(0)

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

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

The two methods of TD(n) and Monte carlo can be to get the best of both worlds using TD(λ) methods [15].

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. 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 both on-policy.

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 [17] [18] . 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 [4]. Therefore we can apply a neural network as an approximator $\hat{Q}$ for Q.

The algorithm we get is called Deep-Q-Learning [8][9]. 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 [8] [9].

Algorithm 3 Deep Q-Learning (DQN)

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

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

        Perform a gradient descent step on

$$\left(\gamma_j - Q(\phi_j, a_j; \theta)\right)^2$$

        Every C steps, reset $\hat{Q} \leftarrow Q$
    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 [6], 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.

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 [7].

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) [2].

First, similar to DQN [8, 9], 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 [2], the SAC algorithm was later modified to learn and adjust α throughout training [3]. This not only improves efficiency—since choosing an optimal temperature is task-dependent and non-trivial [3]—but also allows the policy to explore uncertain regions while being more exploitative in familiar areas.

The final SAC algorithm [3] is as follows:

Algorithm 4 Soft Actor-Critic Algorithm

Input: θ_1, θ_2, ϕ
$\bar{\theta}_1 \leftarrow \theta_1, \bar{\theta}_2 \leftarrow \theta_2$
$\mathcal{D} \leftarrow \emptyset$
for each iteration **do**
    for each environment step **do**
        Sample action: $a_t \sim \pi_\phi(a_t|s_t)$
        Sample next state: $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
        Store transition: $\mathcal{D} \leftarrow \mathcal{D} \cup (s_t, a_t, r(s_t, a_t), s_{t+1})$
    end for
    for each gradient step **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
end for
Output: θ_1, θ_2, ϕ

One noteworthy difference from conventional actor-critic methods is that SAC uses two Q-functions. While not strictly necessary, this significantly speeds up training [3].

Between 2018 and 2019, when SAC was first introduced, it outperformed other popular methods such as TD3 [1], DDPG [5], and PPO [13] 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 [11]. Yet the most popular method is that of PPO [13], as it is computationally simpler. The method used in PPO is to use a clipped objective which results in a loss function like so:

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

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 [12], as it better balances bias and variance. GAE is defined as:

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

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 2.0.2. This leads to this objective:

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

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

Here is a reference version of the algorithm from Open AI's spinning up documentation [10]:

Algorithm 5 PPO Algorithm

1: **Input:** initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards to go $\hat{R}_t$ (This is an estimate of the return R_t for each state in the trajectories)
5: Compute advantage estimates $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}
6: Update the policy by maximizing the PPO-CLIP objective:

$$\theta_{k+1} = \arg\max_\theta \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \text{clip}\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}} \right)$$

7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_\phi \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} (V_\phi(s_t) - \hat{R}_t)^2$$

8: **end for**

Chapter 3

Conclusions

The conclusions are presented in this Chapter.

Bibliography

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

[10] OPENAI. Proximal Policy Optimization. https://spinningup.openai.com/en/latest/algorithms/ppo.ht
Jan. 2020.

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

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

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

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

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

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

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

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