Chapter 1

Introduction

Reinforcement learning (RL) is a field that has received a large amount of attention in recent years. A RL method tries to learn an optimal sequence of actions based on sparse rewards for it's actions. Advances in the field have resulted in super-human results in many well-known games. RL was the first algorithm to succesfully beat one of the worlds best Go players Silver et al. (2017). In addition RL methods have learned to fly helicoptersAbbeel et al. (2007), play at a super-human level in Atari games Mnih et al. (2015) and reduce server cluster cooling expenses by 40% at GoogleEvans and Gao (2016).

What makes RL especially interesting to research is it's generalization potential. StockFish, the defacto algorithm for playing and analyzing chess, is dependent on hundreds of hard-coded heuristics specific for Chess. Silver et al. (2017) uses the same method to learn go, chess and shogi, using no hardcoded heuristics. Mnih et al. (2015) learns to play over 40 completely different Atari games using the same exact algorithm.

Despite the recent success of RL there are a large variety of issues that remain. Sutton and Barto (2018) has an entire chapter dedicated future issues that need to be solved. A popular blog post that was passed around in the RL community (Irpan, 2018) brought up limitations with RL methods as they are today.

? introduced the ALE, a simple to use package that allows RL methods to play RL games. Due to it's simplicity and the wide array of different games, it is often used to evaluate new models, for example in Mnih et al. (2015). This suite of Atari games clearly shows some of these limits in RL.

One key point in Irpan (2018) was that the sample-efficiency of RL is low. For example, Mnih et al. (2015) required that the RL method played each game for 200 million frames. The current state of the art results still require 20 million frames per game (Hessel et al., 2017). Not only does this require a large amount of computation power and memory, it is also simply infeasible for most practical scenarios.

Another issue is that there are certain games in ALE which have proved difficult to solve using RL. One of the hardest to solve is called Montezuma's Revenge(MR). In this game most methods struggle to get out of the first room as there are many ways to lose the game and only a few complicated combinations of actions that lead to good results. This game is viewed as a problem that requires good exploration. This is a concept that will be discussed in depth in the theory section. The underlying idea is that for the RL method to learn this complicated combination of actions is a good idea, it must actually perform. This is unlikely to happen if the agent acts completely randomly so an informed exploration of possible actions is needed.

This project considers the paper O'Donoghue et al. (2017) which tries to combine statistical uncertainty with current RL methods to minimize the isses above. The paper attempts to increase the level of exploration in areas the method is uncertain about the correct action while decreasing exploration in areas with high certainty. Using this concept, O'Donoghue et al. (2017) managed to get out of the first room in MR without any heuristics. This was the highest performance achieved in MR when published.

This project will go in detail through the theory behind O'Donoghue et al. (2017) method. Then to test the method the classical RL environment known as cartpole introduced in Barto et al. (1983) will be used to compare and analyze the positive and negative sides to using the method described in O'Donoghue et al. (2017).

Before moving on to the theory section, it is important to note that RL related problems have not only been researched in the field of computer science. In fields such as control theory and operations research the same problem and approaches are often discussed. Another popular name for reinforcement learning in these fields is approximate dynamic programming and some of the sources used throughout this report refer to the topic under this name. (Powell, 2011, p. 16)

Chapter 2

Theory

2.1 Introduction to RL Terminology

2.1.1 A Guiding Example

Many of the concepts in RL are best explained through an example. Consider a version of the ’gridworld’ game discussed in p. 76 Barto et al. (1983) seen in figure 2.1. The goal of the game is to move from the starting position in the bottom left to the goal in the top right. The player can move any non-diagonal direction and trying to move off the grid will leave the player in it’s previous position.

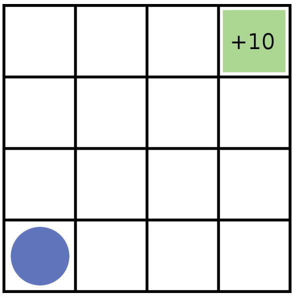


Figure 2.1: Left: Initial State of Gridworld. The blue circle represents the player, while the green square represents the goal.

In RL literature it is common to decompose this problem into an *agent* and *environment*. The agent is the object which can perform actions. In the gridworld example this is the player. The environment is what the agent interacts with and what returns information about the game.

In general the environment consists of two things. First is the state, which defines what is going on in the environment. In the gridworld example this could be the players position on the grid. Second is the reward, some signal that what the agent is doing is right or wrong. This could be -1 for every action taken and +10 for reaching the goal. Note that one does not always need a reward per action. For example in chess the reward can be purely +1 for a win, -1 for a loss and 0 for everything else. This agent environment decomposition is visualized in figure 2.2.

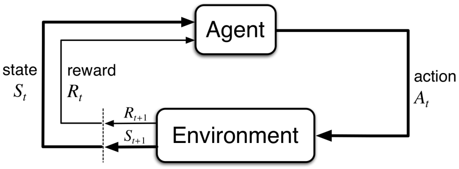


Figure 2.2: Visualization of the Agent and Enironment. The figure is from (Sutton and Barto, 2018, p. 48)

Finally upon reaching the goal the game is over, making the goal state a *temrinal state*. Environments with terminal states are called *episodic* as during training the game must be reset to keep playing. If instead the player is teleported back to the start and can keep playing it is a *continuous* environment. This is an important distinction for some of the methods used to learn to play the game.

2.1.2 Tabular Markov Decision Process

Though the applications of RL are broad, most of the theory behind RL has been developed around simpler problems that have the two following attributes. Firstly the problem should be tabular, meaning that the number of combinations of states and actions should be small enough to fit in memory. Secondly the problem should follow a Markov Decision Process (MDP). (Barto et al., 1983, p. 23)(Powell, 2011, p. 57)

MDPs build on the concept of Markov chains (MC). According to Ross (2014) a MC is a stochastic process consisting of a sequence of successive random variables S_t that can take values from a countable set $\mathbb{S}$. These are called states. The state at a time step is denoted S_t for $t \in T$ where T is a discrete or continuous set that often relates to the time step for the occurrence. For example, in gridworld S_2 would represent the players position after two actions.

In a MC the next state is dependent only on the current state. This is known as the Markov Property. Thus a transition matrix is defined consisting of probabilities

$$p_{s,s'} = P(S_t = s'|S_{t-1} = s) \quad \text{for } s', s \in \mathbb{S} \tag{2.1}$$

which denote the probability of transitioning from state s to state s'.

In a MDP two additional factors are added. This is an action A from a discrete action set $\mathbb{A}$ and a reward R from a real set $\mathbb{R}$. The transition probability is then defined as

$$P(s', r|s, a) = P(S_t = s', R = r|S_{t-1} = s, A = a) \tag{2.2}$$

Essentially a MDP allows for an action to be taken at each state which in turn effects the probability distribution of the next state. In addition the transition to a new state returns a numerical reward R. (Sutton and Barto, 2018, p. 38).

From this definition one can see that the gridworld example can be modeled as an MDP. Given a player state, an action can be chosen to move to North, East, West or South. As long as the action leads to a state that is on the grid, the transition probability is one to the desired square and zero for all other squares. If the action leads off-grid the transition probability is one to stay in the same square. This follows the Markov property as the transition probability is only dependent on the current state and action. Finally, performing an action the agent receives a reward of +10 if it reaches the goal and -1 otherwise.

2.2 Dynamic Programming solutions to Tabular MDP's

2.2.1 Bellman Equation

Given an MDP the goal is in general to maximize the total reward returned. The total discounted reward can be defined as

$$G_t(s) = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}(A_t|S_t = s) \quad \text{where } 0 << \gamma < 1. \tag{2.3}$$

The future rewards are discounted by a factor of γ to ensure the convergence of the methods that follow when used on endless environments. Episodic environments can have $\gamma = 1$, however in practice it is still common to use $\gamma < 1$ as it is the equivalent of weighting short-term rewards more than the possibly less reliable long-term rewards.

The total reward is influenced by the actions taken in the MDP so the aim is to estimate a function that takes in the current state and outputs the probability of performing an action. This is refereed to as the policy and is denoted

$$\pi = P(A = a|S_t = s). \tag{2.4}$$

The notation in equation 2.3 is often shortened for the sake of readability. For the rest of this project $G_t(s)$ will be denoted as G_t and $R_t(A_t|S_t = s)$ is denoted R_t. The optimization of the process can then be defined as

$$\max_{\pi} \mathbb{E}_{\pi}\big[G_t\big] \tag{2.5}$$

where G_t is the total discounted reward when following policy π.

To solve this maximization problem one must be able to calculate $\mathbb{E}_{\pi}\big[G_t\big]$ given a policy π. Consider equation 2.5 from a given state.

$$v_{\pi}(s_t) = \mathbb{E}_{\pi}[G_t|S_t = s_t] \tag{2.6}$$

Equation 2.6 is known as the *value function.* It is the expected reward to be gained from the state s_t and onward while following policy π. In other words this represents how good it is to be in state s_t. Expanding G_t gives

$$v_{\pi}(s_t) = \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1}|S_{t+1} = s_{t+1}]. \tag{2.7}$$

Noting that

$$\mathbb{E}_{\pi}[R_t] = \mathbb{E}[R_t|A_t = a] = \mathbb{E}[r_t] = r_t \tag{2.8}$$

one can rewrite equation 2.7 as

$$\begin{aligned} &= r_{t+1} + \gamma \mathbb{E}_{\pi}[G_{t+1}|S_{t+1} = s_{t+1}] \\ &= r_{t+1} + \gamma v_{\pi}(s_{t+1}). \end{aligned} \tag{2.9}$$

There is one such formula per state, so the whole environment is represented by a system of $|\mathbb{S}|$ simultaneous linear equations with $|\mathbb{S}|$ unknown values $v_{\pi}(s)$. This decomposition was first suggested by Richard Bellman(Bellman, 1957) and hence is called the Bellman Equation. The process of solving the above is known as *policy evaluation* in RL literature.

The value function decomposition can also be expanded to the action-value function which represents how 'good' each action a from state s is.

$$\begin{aligned} Q(s,a) &= \mathbb{E}_{\pi}[G_t|S_t = s_t, A_t = a] \\ Q(s,a) &= r_t(a) + \gamma \mathbb{E}\big[v(S_{t+1})|S_t = s_t, A_t = a\big] \end{aligned} \tag{2.10}$$

In certain situations, which will be discussed at a later stage, equation 2.10 can be a more useful decomposition.

To conclude, a naive way to solve equation 2.5 is then to calculate the value or action-value function for all policies and simply pick the policy that has the highest value for the initial state. This solution is guaranteed optimal but computationally inefficient and in practice unfeasible for many environments.

(Powell, 2011, p. 58-61)(Sutton and Barto, 2018, p. 59)

2.2.2 Value and Policy Iteration

If one can perfectly model the environment and the MDP has a finite number of states the Bellman Equation can be solved using dynamic programming. The following overview of the dynamic programming method is summarized from (Sutton and Barto, 2018, p. 74-84)

Policy Evaluation

As an alternative to calculating the value function from the system of $|S|$ equations there is the iterative method

$$v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \tag{2.11}$$

and terminating the iteration when

$$\|v_{k+1} - v_k\| < \epsilon \quad for \quad \epsilon > 0. \tag{2.12}$$

This converges given that $v(S)$ exists, which is when $\gamma < 1$ or that an episode is guaranteed finite. This method can be usefull for large state spaces.

Policy Iteration

This policy evaluation method can then be used to create a new policy π' by greedily picking actions in each state S_t based on $V_\pi(S_{t+1})$.

$$\pi'(s) = \underset{a \in \mathbb{A}_t}{\arg\max}\, \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\big] \tag{2.13}$$

It can be proven that the above leads to a policy such that $v_{\pi'}(s) \geq v_\pi(s) \;\; \forall s \in \mathbb{S}$. (Sutton and Barto, 2018, p. 78-79)

By repeatedly running policy evaluation and the policy improvement step above results in monotonically improvement in the policy and value function. For a finite MDP this results in the optimal policy and value function as this implies a finite number of policies.

Value Iteration

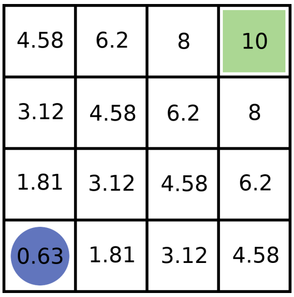
Policy iteration as described above requires that 2.11 converges before improving the policy. The basis for value iteration is that the policy evaluation does not need to converge first. Instead it combines both steps into one simple update rule:

$$v_{k+1}(s) = \max_{a \in \mathbb{A}_t} \mathbb{E}\big[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a)\big]. \tag{2.14}$$

The termination condition is given by equation 2.12 as in policy iteration. When the value function has converge a policy is given by

$$\pi(s) = \arg\max_{a \in \mathbb{A}_t} \mathbb{E}\big[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a\big] \tag{2.15}$$

The result of running value iteration on the gridworld example can be seen in figure 2.3



4.58	6.2	8	10
3.12	4.58	6.2	8
1.81	3.12	4.58	6.2
0.63	1.81	3.12	4.58

Figure 2.3: Value Iteration run on gridworld example. A discount rate of 0.9 was used. Note that the closer the player is to the green zone, the higher the point sum.

Finally it can be proved that value iteration also converges to the optimal value and policy function given a finite MDP. (Powell, 2011, p. 89-93).

2.2.3 Limitations of Value and Policy iteration

There are many situations where value and policy iteration(equation 2.13 and 2.14) cannot be used to solve MDPs. Powell (2011) (p. 5-6) discusses the curse of dimensionality. When there are too many states or actions it is computationally infeasible to run the above algorithms. Sutton and Barto (2018) (p. 91, 119) points out that a suitable environment model isn't always available, which makes it impossible to directly calculate $V(S_{t+1})$.

2.3 Reinforcement Learning solutions to Tabular MDP’s

RL builds on a different approach than the methods discussed above. Consider the grid-world example. In value and policy iteration the game is never played during training. The entire policy is learned based on a model of how the game works. In a RL approach the agent plays the game and based on the experiences of what happens tries to learn the optimal policy.

2.3.1 Advantages of Learning From Samples

The experiences used to train the agent are known as *samples* in RL literature. A sample is simply a set of states, actions and rewards that resulted from interacting with the environment. The specific size of a sample varies based on the method and can be anything from one state transition to all the transitions within an episode.

There are two main reasons why learning from samples can be useful. In many cases one can generate samples of the environment without having a model of all the dynamics of the system. Consider for example trying to maximize profit by buying and selling a stock. There are simply far too many factors and unknowns to be able to perfectly predict the future price of a stock. However for the correct stock there can exist decades of pricing history. This history can be used as samples to train a RL model. This is a common problem, where creating an accurate model of the environment can be a lot more challenging then sampling from the environment.

Secondly reinforcement learning algorithms can focus on modeling promising states and neglect states that clearly lead to sub-optimal results. In contrast the dynamic programming methods run the same number of calculation for all states. This allows reinforcement learning to solve larger MDP’s than DP methods. (Sutton and Barto, 2018, p. 115)

2.3.2 Temporal Difference learning

Generalized Policy Iteration

There are two major paradigms in reinforcement learning: generalized policy iteration (GPI) and policy gradient (PG) methods. The focus of this project will be on generalized policy iteration. These are methods that follow the same structure as value and policy iteration. Essentially a GPI methods are characterized by the fact that they try to model the value function of the MDP. This value function is then used to improve the policy. The improved policy then leads to a new value function, and the cycle repeats. A visualization of this can be seen in figure 2.4 (Sutton and Barto, 2018, p. 86).

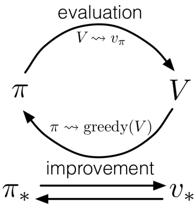

evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement
π_*
v_*

Figure 2.4: Visualization of GPI. Taken from p. 86 Sutton and Barto (2018)

Learning from samples

How to learn a value function from samples is not obvious. Consider the case of trying to learn to play chess. One way to model this game as an MDP is to give a reward upon winning the game. Given one game, how does one propagate the final reward to the states that lead up to the result?

One set of methods are Monte Carlo methods. These simply update the values of the states that were visited before the reward. Mathematically

$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[G_t - V_\pi(S_t)\big] \tag{2.16}$$

where α is the step size of the update. This can be viewed as using G_t as the target value that is trying to be modeled. Note that using this method one must wait until the end of the game before updating values.

Temporal difference methods instead update the value function every time a step is taken in the MDP.

$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[R_{t+1} + \gamma V_\pi(S_{t+1}) - V(S_t)\big]. \tag{2.17}$$

In this case the target is based on the estimated value of the next state and the reward gained in the step taken. In reinforcement learning literature this is referred to as bootstrapping the target value.

Q-Learning

One issue that remains with the aforementioned methods is that they are *on-policy*. This means that in order to update the value function of a policy one has to use transition samples following the policy. This means that if the policy is changed, a new set of transition samples are required to keep training. This means that an agent must be given direct access to the environment to learn new policies and decreases the amount of data that can be used for training.

Watkins and Dayan (1992) marked a large step forward in reinforcement learning through the development of an off-policy temporal difference method. The method is based on the action-value function (2.10) and is called Q-learning.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.18}$$

Note that this equation is completely independent of the policy followed when generating the sample. This means in contrast to an on-policy method, Q-learning can be trained on transition samples from any policy regardless of what the Q-learning policy is.

2.3.3 Exploration vs Exploitation

Given the correct action-value function, the optimal policy will be to pick the action with the highest Q-value.

$$A_t = \arg\max_a Q_t(a) \tag{2.19}$$

The policy defined in equation 2.19 is known as a greedy policy and following this policy is called *exploiting* the policy. Note that in contrast to a greedy method in computer science, this greedy policy does take into account future events through the reward propagated through the bellman equation.

The issue with this policy is that one does not have the correct action-value function. It will always pick what is the estimated best action without picking new actions to test if it will lead to an even better reward. In other words, the greedy policy will never *explore* the environment and therefore might miss a better policy.

A balance is needed between exploiting the policy to maximize reward and exploring to find a better policy. A simple solution to this problem is the ε-greedy policy

$$A_t = \begin{cases} \arg\max_a Q_t(a) & \text{with probability} \quad 1-\varepsilon \\ a \sim \text{Uniform}(\mathbb{A}) & \text{with probability} \quad \varepsilon. \end{cases} \tag{2.20}$$

where $0 < \varepsilon < 1$ and $\mathbb{A}$ is the set of all legal actions. Asymptotically this policy is guaranteed to visit every state an infinite amount of times. This generally works quite well in practice but can be inefficient for complex environments. (Sutton and Barto, 2018, p. 27-28)

2.4 Deep Q-Learning

2.4.1 Function Approximation

The reinforcement methods discussed in the previous section are called tabular methods as they consist of saving a value for each state or an action-value for each state action pair. These methods have two major shortcomings (Sutton and Barto, 2018, p. 195-196). Firstly when either the action or state space becomes sufficiently large, this representation becomes impractical due to memory constraints. For example, the game of chess has a state space of magnitude 10^{43}(Shannon, 1950) so creating a dictionary mapping from state to value is impossible with current technology.

The second issue is generalization. The tabular methods discussed require many visits to each state and action of interest to have an accurate action value estimate. Given an unvisited state there will be no good estimate of the action-values for the policy to be based upon. The tabular method does not generalize to new or even rarely visited states.

To illustrate this consider the environment of buying and selling stock while maximizing profit. Take for example the Apple stock prices and define the state as the differenced opening prices rounded to the nearest cent. Figure 2.5A shows that small changes in price are most common. Therefore one expects good value approximations at these price changes. However for larger changes in prices there is less or no data. If in evaluation the environment results in a large price drop that hasn't been seen before the tabular methods will have no action-value estimates leading to no policy to follow. This is not only a problem for large price changes. Zooming in on the price data as in figure 2.5B shows that there are certain low price changes that have limited data. For these the same problem will occur.

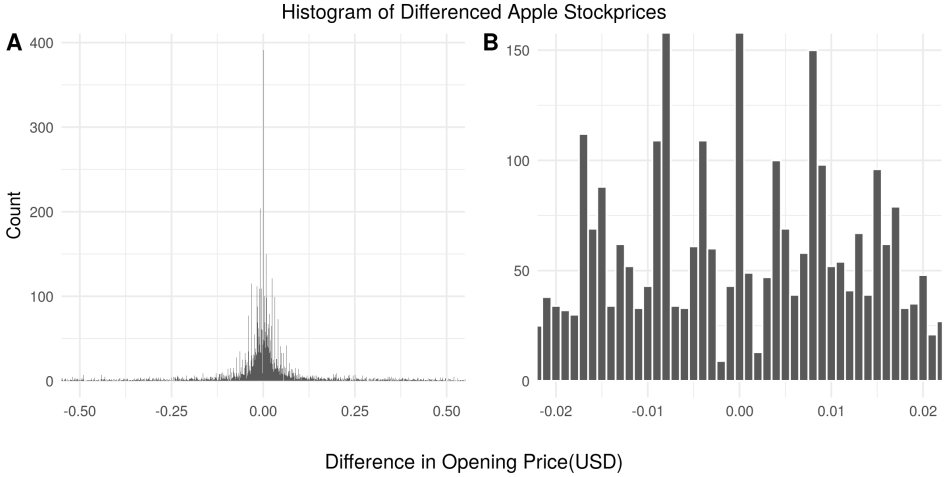


Figure 2.5: Differenced Apple Stock Opening Prices: A

Ideally the value estimate the method should be able to generalize to limited samples or completely unseen states. In the stock price example the value and policy at similar prices should give information about how the agent should act given an unseen price.

The approach to solving this is using function approximation. When estimating the value or action-value function one is trying to estimate a continuous value given a set of input values. This is a regression problem. So instead of using a table to map states to values one can use regression methods to do the same.

Since the value function is dependent on the policy and the policy is changing during training, the target is non-stationary. It is therefore important that the regression method chosen must be able to deal with non-stationary targets.(Sutton and Barto, 2018, p. 198-199)

It is important to note that using function approximation means that the convergence guarantees no longer hold. However linear approximation methods generally converge in practice and methods can often be tweaked to increase the stability of convergence.

2.4.2 Nonlinear function approximation

Action value functions can be complicated functions so it can be desirable to have a nonlinear function approximator. To do this standard numerical optimization methods, like gradient descent, are used. For this a loss function must be defined. Consider the case of trying to minimize the mean square error of the action-value estimate

$$L_i(\theta_i) = \mathbb{E}_{s,a}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \tag{2.21}$$

where θ are the parameters of the model being used. As this project focuses on temporal difference methods the target is set to the same as in Q-learning (equation 2.18).

$$y_i = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1})|s,a\right] \tag{2.22}$$

The loss function can then be differentiated with respect to the model parameters resulting in

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a;s'}\Big(\Big[r + \gamma \max_{a'} Q(s',a';\theta_i) - Q(s,a;\theta_i)\Big)\nabla_{\theta_i} Q(s,a;\theta_i)\Big] \tag{2.23}$$

In general this Jacobian matrix is used in stochastic gradient descent.

Fully Connected Neural Networks

One popular class of nonlinear models are neural networks. This class covers a large variety of models. The simplest and perhaps most used model is the fully connected(F neural network(NN). A fully connected NN consists of sets of neurons, called layers, where each neuron receives an input from every neuron in the previous layer.

A neuron is simply either a regression or classification model where the output is passed through a function named the activation function. This function is usually non-linear to allow the NN to model non-linear functions. Mathematically a neuron is expressed as

$$Z = \sigma(W^T X) \tag{2.24}$$

where X is the vector of inputs from the previous layer, W are the coefficients of the regression or classification and σ is the activation function.

The coefficients of regression/classification are called weights. These are the unknown parameters in the model that must be estimated. To do this a loss function must be defined. There any many choices of loss function. Two common choices are mean square error(MSE) for regression problems and cross-entropy for classification problems.

MSE is defined as

$$L(\boldsymbol{\theta}) = (y - \hat{y}(\boldsymbol{\theta}))^2 \tag{2.25}$$

while cross-entropy is

$$L(\boldsymbol{\theta}) = -y \log \hat{y}(\boldsymbol{\theta}). \tag{2.26}$$

The weights that minimize these losses around found through gradient descent. The weights in layer k are updated by the formula

$$w_k = w_k - \gamma \frac{\partial L}{\partial w_k} \tag{2.27}$$

where γ is known as the learning rate and controls the step size of the optimization.

The implementation of this can be simplified through the use of the back propagation equations. Using chain differentiation it can be shown that the

$$\begin{aligned} \frac{\partial L}{\partial w_K} &= \delta^K z_K && (2.28)\\ \frac{\delta L}{\delta w_k} &= \delta^k z_{k-1} && (2.29)\\ & && (2.30)\\ \text{where} \quad \delta^K &= (y - z^K)^2 && (2.31)\\ \delta^k &= \sigma'(w_k z_k) w_{k+1} \delta^{k+1} && (2.32) \end{aligned}$$

where K is the final layer and σ' is the differentiated activation function. The above allows the gradients to be calculated based on gradient calculation for next layer. To train the network the prediction is calculated by a forward pass through the network and then the weights are updated by calculating the above for each layer in backward order. (Hastie et al., 2009, p. 392-396)

By simply setting the target to be the Q-learning target in equation 2.22 one can use neural networks as a nonlinear function approximator in RL.

The Deadly Triad

There is an issue that arises with function approximation, which is known as the deadly triad. When combining function approximation, bootstrapping and off-policy training one often finds that the estimate becomes unstable and can diverge. This is especially a problem with nonlinear function approximators like neural networks (Mnih et al., 2013). Dropping one of these factors essentially negates this problem, however using all of these is desirable due to their contribution to an increase in performance. (Sutton and Barto, 2018, p. 264-265).

2.4.3 Deep Q Networks

The recent rise in interest in NN led to interest in using these as a nonlinear approximator for Q-learning. In Mnih et al. (2013) a method called the Deep Q Network (DQN) was introduced achieving state of the art results on a select few Atari games. They used a multilayer NN Q-value function approximator that takes in a state S as input and outputs a Q-value per action. For exploration they follow a ε-greedy policy starting with $\varepsilon = 1$ that decreases towards to $\varepsilon = 0$ as training progresses.

Due to the deadly triad issues, some modifications had to be made to Q-learning to handle the divergence issues. To deal with this Mnih et al. (2013) reintroduced a concept called *experience replay*, originally introduced in Lin (1993). Instead of training the network after every step taken, samples are saved as the tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ creating a data set of samples $\mathcal{D} = e_1, ..., e_n$. The neural network can then be trained using samples drawn randomly from $\mathcal{D}$. In practice this is done every few steps taken by the agent.

Mnih et al. (2015) further increased stability by using two neural networks instead of one. The second neural network, called the target network, is used to calculate the target Q value, the $Q(S_{t+1}, a)$ term in equation 2.18. The weights of the target network are copied from the original network, called the online network, after a large number of steps. This creates a more stable optimization target.

In addition, instead of using the MSE Mnih et al. (2015) suggests clipping the gradient of the loss function to be between -1 and 1 as they observed it lead to more stable learning. Since one only uses the derivative of the loss function in this application this is the equivalent of using the Huber loss function 2.33

$$L(y, \hat{y}) = \begin{cases} (y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq 1 \\ 2|y - \hat{y}| - 1 & \text{otherwise.} \end{cases} \tag{2.33}$$

as it is defined in (Hastie et al., 2009, p. 349).

The DQN version from Mnih et al. (2015) achieved new state of the art results on a much larger set of Atari games and has become a standard baseline for RL methods.

2.4.4 Further developments on DQN

Many additional tweaks to DQN have been introduced since Mnih et al. (2015). Some of the more promising changes were combined and tested in Hessel et al. (2017). To decrease the experiment run-time this project implements two of changes mentioned in Hessel et al. (2017), namely dueling DQN and double DQN, which gives a method named Dueling Double DQN (Dueling DDQN). However, to keep notation short, the abbreviation DQN will refer to the Deuling DDQN for the rest of this project. These changes are not fundamental to this project so they will only be briefly discussed below.

Double DQN

In Hasselt (2010) and van Hasselt et al. (2015) it was shown that Q-learning overestimated Q-values. The following change in the Q-learning target reduced this bias

$$R_t + \gamma Q'(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a')) \tag{2.34}$$

where Q is estimated by the online network and Q' is estimated by the target network.

Dueling DQN

The Dueling DQN builds upon the idea that a Q-value $Q_\pi(s, a)$ can be viewed as a combination of state value $v_\pi(s)$ and the improvement by taking an action called the advantage function $A_\pi(s, a)$. Wang et al. (2015) suggested that representing this in the network architecture could simplify learning. Dueling DQN consists of splitting the final layer into two streams. One stream is used to estimate the state value. The other stream creates an advantage value for each action. These are finally added together as in equation 2.35 to produce a Q-value per action that can be trained using the same method as a regular DQN.

$$Q(s, a) = v(s) + \left(A(s, a) - \frac{1}{|\mathbb{A}|}\sum_{a'} A(s, a')\right) \tag{2.35}$$

2.4.5 Limits of DQN

Despite the human-level performance of DQN methods in many Atari games (Mnih et al., 2015) there are still some games DQN fails to complete successfully. These games have proved to be difficult to despite developments in RL (Mnih et al., 2016; Schulman et al., 2017; Hessel et al., 2017). One game of particular interest in the RL research community is Montezuma's revenge. This environment has sparse rewards with many policies leading to a quick loss. In this case modern RL methods fail to explore efficiently to reach any successful policy.

2.5 Exploration through uncertainty

Despite the guarantee that ε-greedy will asymptotically explore all states this might not always be computationally feasible. Even if it is computationally feasible, the sample efficiency of RL is known to be quite bad. One of the most sample efficient methods within Atari games is Hessel et al. (2017) but this still requires 20 million frames per game. Since ε-greedy uses no information about the environment or agent a focus of research has been to perform more informed exploration.

2.5.1 Uncertainty in Reinforcement Learning

Knowing the variance of the Q-value estimate gives an insight into how certain the model is about the Q-value. This can be used to pick actions that are estimated to be sub-optimal but could have higher (or lower) Q-values due to uncertainty. However calculating this variance isn't as simple as a regular regression setting.

Propagation of Uncertainty

To understand the challenge of variance in RL, first consider a naive attempt at incorporating variance in action selection. Assuming that the variance of an estimate is proportional to the inverse visit count to a state one can define the policy

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}} \right]. \tag{2.36}$$

This can be viewed as setting the Q-value to be the upper confidence bound of the Q-value (Strehl and Littman, 2008)(Sutton and Barto, 2018, p. 35-36).

This assumes that future returns come from a stationary distribution. In reality this assumption is often wrong. As the agent's policy changes, the future returns change, which implies an non-stationary distribution. This means that the variance of a Q-value is dependent on the variance of Q-value estimate along with the variance of future Q-values due to the uncertainty in the agent's policy. Therefore, in the same way Q-values must be propagated from future Q-values, the variance of the Q-value must be propagated from the variance of future Q-values. (Moerland et al., 2017)

To illustrate the issue, consider the chain example from Osband and Roy (2016). Consider N states connected in a chain as in figure 2.6. The agent starts in S_1 and has two actions; move left or right. Transitioning to S_1 gives a reward sampled from $\mathcal{N}(0,1)$, S_N gives a reward from $\mathcal{N}(1,1)$ and the rest of the states result in no reward.

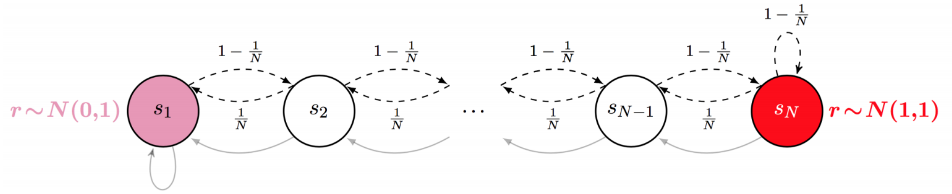


Figure 2.6: Chain environment. The figure is taken from Osband and Roy (2016)

The optimal policy is to always move right. However an agent following the policy in equation 2.36 will quickly end up underestimating the value of states too the right as multiple visits to S_2 would lead to a low exploration bonus despite the fact that states further to the right have not been properly explored. If this occurs before the agent reaches S_N it will never find the higher reward to the right and end up with a suboptimal policy.

Optimism in the Face of Uncertainty

One method to propagate the uncertainty from future value estimates is to include the exploration bonus in the Bellman equation. This is done with the value function in Strehl and Littman (2008):

$$\max_a \mathbb{E}\Big[R_t + \gamma V(S_{t+1}) + \beta N(S,a)^{-\frac{1}{2}}\Big]. \tag{2.37}$$

If one considers the uncertainty around the value to be an interval of statistically plausible values, this method optimizes the Bellman equation over the highest statistically plausible value. This has given the method the name optimism in the face of uncertainty(OFU). This method is only applicable to tabular environments and fails for large state spaces where visit counts tend to be low.

Bellemare et al. (2016) generalized this equation away from relying directly on visit counts by estimating a visit count from a linear approximation model. This achieved state of the art results in multiple environments when published. However, an issue with this method is that it changes the loss function which no longer directly optimizes the Bellman equation. This can lead to inefficient exploration at times or sub-optimal behavior. (Moerland et al., 2017)

2.5.2 Posterior sampling for reinforcement learning

A second paradigm in uncertainty based exploration is posterior sampling for reinforcement learning (PSRL). This method builds on a bayesian view of reinforcement learning. Considering the task of maximizing reward from an MDP. Bayesian reinforcement learning treats the unknown MDP as a random variable. To do this one considers the expected one-step reward $\hat{R}^*(s,a)$ and transition probabilities $P^*(s,a)$ to be random variables. Denoting a sample from these distributions as $r*$ and $p*$ one can create a posterior sample of the action-value $Q*$ conditioned on the history of transitions by using the following equation.

$$Q^*(s_t,a_t) = \hat{R}^*(s,a) + \sum_{s_{t+1},a_{t+1}} P^*(s_{t+1}|s_t,a_t) \max_{a'} Q^*(s_{t+1},a_{t+1}) \tag{2.38}$$

The PSRL method then defines the policy by greedily picking the best action over a posterior sample of each available action-value. This is known as Thompson sampling. (Strens, 2000)(Osband and Roy, 2016)

$$a_t = \arg\max_{a\in\mathcal{A}} Q^*(s_t,a) \tag{2.39}$$

To grasp the intuition to why the above leads to exploration consider an environment with only two actions. Assume that the action-value posterior is gaussian and that there are two actions to choose from as shown in figure 2.7.

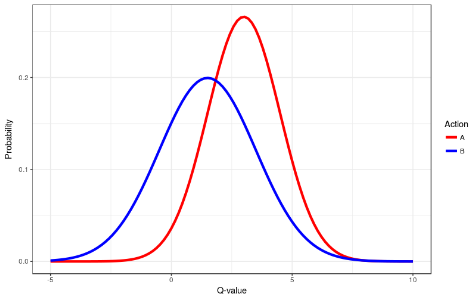


Figure 2.7: Posterior distribution of action-values: Despite action A having a higher expected value, the posteriors indicate that action B could potentially be the best action.

Figure 2.7 shows that the expected Q-value of action A is higher than B. However, the posterior distribution of Q-values can be viewed as the distribution of plausible values for Q (Osband and Roy, 2016). The overlap between the two distributions indicates that there is a certain probability that action B is actually better than action A. By sampling these posteriors when chosing action one gives a probability of choosing action B over action A that is related to the amount of overlap between these distributions.

Osband and Roy (2016) shows that the sample-efficiency scales better with respect to the number of states and actions for posterior sampling than optimism in the face of uncertainty. However, the challenge remains in finding a good posterior that is not so computationally heavy that it cancels out the sample-efficiency.

Chapter 3

Bayesian Q Learning

In an attempt to find a better balance between exploration and exploitation this thesis investigates the use of bayesian methods to allow for Thompson sampling. This chapter builds and compares bayesian methods in a linear model context before attempting to extend the most successful methods to neural networks.

3.1 Linear Q learning

In linear Q learning the goal is to create a regression model that maps the state and action to a Q-value, $Q(s, a)$. Let x_t denote the state and action at timestep t. X then denotes the design matrix containing these features and Q the vector of corresponding Q-values. The regression model can then be defined as

$$Q = X\beta + \varepsilon \quad \text{where} \quad \varepsilon \sim N(0, \sigma^2)$$

with the response value defined as

$$Q(s, a) = r_t + \arg\max_{a'} Q(s', a'). \tag{3.1}$$

The ordinary least squares solution to the β coefficients can then be found using the normal equation which in matrix form is

$$\beta = [X^T X]^{-1} X^T Q$$

Given this model the agent can take an action by acting greedily over the models $Q(s, a)$ values in a given state. Since this purely an exploitation strategy, it is often coupled with the ε-greedy policy.

3.2 Bayesian Linear Q learning

To extend linear Q learning methods to follow a Thompson sampling policy a bayesian perspective is required. In a RL perspective the goal is to model the posterior

$$p(Q|\theta, X_t) \propto p(\theta|X_t, A_t, R, X_{t+1})p(\theta)$$

where Q is a vector of all Q-values given the state X_t and θ denotes all parameters.

The calculation of an arbitrary posterior is computationally heavy which is ill-suited to the already long running reinforcement learning methods. To keep computation costs low this thesis will only consider conjugate priors which have an analytical solution.

3.2.1 Normal Prior with Known noise

There are multiple ways to setup a bayesian regression model using conjugate priors. First consider the case used in Azizzadenesheli et al. (2019) which creates one model per action and assumes the noise variance is known. The known noise variance is then treated as a hyperparameter. In this case the posterior can be expressed as

$$p(\beta_a|\sigma_{\varepsilon_a}, X_t, A_t, R, X_{t+1}) \propto p(Q_a|\beta_a, \sigma_{\varepsilon_a}, X_t)p(\beta_a)$$

In literature it is common to use a gaussian prior for β

$$p(\beta) = \mathrm{N}(\mu, \sigma_\varepsilon \Lambda^{-1})$$

where Λ is the precision matrix. This results in the following posterior update

$$\begin{aligned}
\Lambda_n &= X^T X + \Lambda_0 \\
\mu_n &= \Lambda_n^{-1}(\Lambda_0 \mu_0 + X^T Q_a)
\end{aligned}$$

To calculate the target Q-value the MAP estimate of β is used which in this case is μ.

3.2.2 Normal Prior with Unknown noise

To avoid the noise variance as a hyperparameter it can be included as an unknown parameter.

$$p(\beta_a, \sigma_{\varepsilon_a} | X_t, A_t, R, X_{t+1}) \propto p(Q_a | \beta_a, \sigma_{\varepsilon_a}, X_t) p(\beta_a | \sigma^2_{\varepsilon_a}) p(\sigma^2_{\varepsilon_a})$$

The conjugate priors for this setup are

$$\begin{aligned} p(\sigma^2) &= \text{InvGamma}(\alpha, b) \\ p(\beta|\sigma^2) &= \text{N}(\mu, \sigma^2\Sigma) \end{aligned}$$

with the posterior update

$$\begin{aligned} \Lambda_n &= (X^T X + \Lambda_0) \\ \mu_n &= \Lambda_n^{-1}(\Lambda_0\mu_0 + X^T Q) \\ \alpha_n &= \alpha_0 + \frac{n}{2} \\ b_n &= b_0 + (Q^T Q + \mu^T \Lambda_0 \mu_0 - \mu_n^T \Lambda_n \mu_n) \end{aligned}$$

Once again μ, the MAP estimate of β is used to calculate the target Q-value.

TODO:: Insert results for linear methods on chain environment?

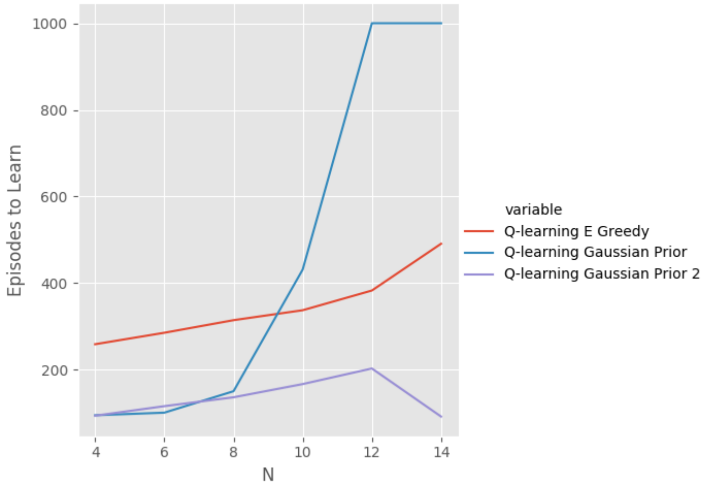


Figure 3.1: WIP: **Performance on chain environment** Guassian Prior 2 includes the noise as a parameter

3.2.3 Propagating Uncertainty

One possible issue with the two above methods is training using the MAP estimate of β. Using the MAP estimate means that the targets come from the following process:

$$y = R + \max_a X_{t+1}\mu_a.$$

Though this does incorporate the variance in the reward process through R it does not convey the variance in the Q-value estimate of the next state. Even in a deterministic environment the policy shifts during training mean that there is an uncertainty in the Q-value of the next state. Quoting Moerland et al. (2017), ”…repeatedly visiting a state-action pair should not makes us certain about its value if we are still uncertain about what to do next.”

One possible method to include this uncertainty is to sample the β posterior when calculating the target value.

TODO:Here is where I want stronger argumentation of why this should work (other than intuition).

TODO:Insert results for linear methods on chain environment

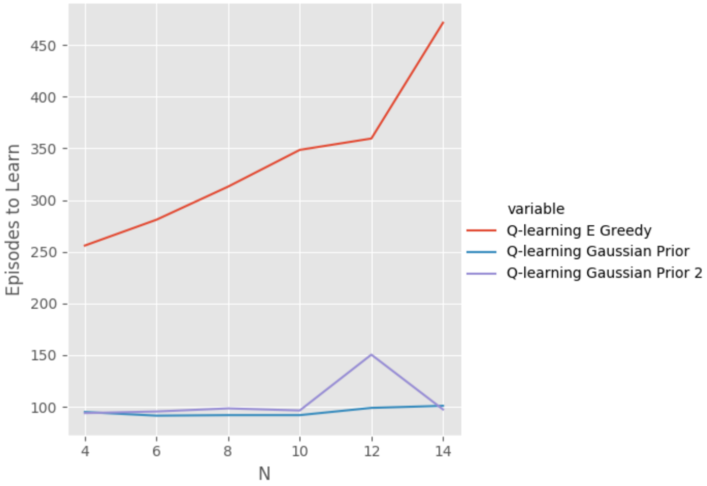


Figure 3.2: WIP: **Performance on chain environment** Guassian Prior 2 includes the noise as a parameter

3.2.4 Temporary Why sampling might not be good enough

I’ve tried to get down my concerns with sampling the posterior for the target.

Sensitivy of the Q-value

I said at an earlier meeting that adding weak priors worked well on the chain environment. However this was a few weeks after I ran these tests, so I'd forgotten that it worked well for known noise case, but crashes for the unknown noise case. The crash is caused by samples from the weak inverse gamma prior InvGamma$(0.001, 0.001)$. This distribution can generate samples that are extremely large which causes the Q-value to explode and eventually overflow.

This can be countered by using a less weak prior but I do think it highlights the issue of using samples from the posterior when there is little data.

I think the issue is best illustrated if we consider a normal distribution. A weak normal distribution will have a large variance and spit out a wide range of values. The intuition is that this doesn't cause problems because we are as likely to sample an extremely large positive or negative number, so the magnitude cancels out.

However in a reinforcement learning setting the sample effects the next target. This means if the first sampled value is a large positive value we move the prior towards this value reducing the probability that the next target is a large negative number. To make matters worse the target is chosen to be the max value among the sampled Q-values so this effect scales with the number of actions. These factors push the Q-values to be large and leads to stability issues.

Summary: Posterior samples based on little data and weak priors lead to instability.

Propagating Uncertainty

This section was supposed to be about the hunch I had that sampling the target isn't actually propagating the uncertainty properly.

However writing this I think I found a simple reason it doesn't. Maybe I'm overlooking something obvious, but consider the posterior update for the known noise case.

$$\Lambda_n = X^T X + \Lambda_0 \tag{3.2}$$

$$\mu_n = \Lambda_n^{-1}(\Lambda_0 \mu_0 + X^T Q_a) \tag{3.3}$$

Note that $X^T X$ is always positive meaning Λ increases regardless of the data provided. Even more importantly Λ is completely independent of the target. This means that visiting a state decreases our variance even if the variance of the next state is large.

I think the issue is that regular bayesian linear regression assumes that the problem is non-stationary, so no matter what data you get in it should always decrease variance.

How can we fix it? Three possibilities

- **Filter**: The difficulty is how this should be setup.

- **Include the policy**: Given the distributions we can calculate the probability of doing different actions. Maybe this can be included in the model to make the problem stationary?
- **Include the Q-value dependencies in the bayesian model**: Maybe it's possible to setup a bayesian regression that includes the information that the Q-value target is dependent on the β values. Note that this will most likely require multivariate bayesian regression.

I want to attempt one of these before stopping any further developments on the masters. This is something we can discuss next meeting in more detail.

3.3 Bayesian Deep Q Network

TODO:this section assumes DQN has been explained

The predominant issue with bayesian methods in deep reinforcement learning is using bayesian methods with neural networks. This thesis will address the linear layer method (TODO:Actual name for method), a simple and computationally efficient method that comes at the cost of accuracy.

The final layer in a DQN is a linear layer. Since bayesian regression is also a linear combination one can replace the final layer with a bayesian regression model per action. This is equivalent to rewriting the regression task to

$$Q = \phi(X)\beta + \varepsilon \quad \text{where} \quad \varepsilon \sim N(0, \sigma^2)$$

where $\phi(X)$ is the neural networks output given an input X. Note that this means the bayesian regression no longer incorporates all the uncertainty since the above assumes no uncertainty in the $\phi(X)$ encoding.

Training the model now needs to be split into two processes. Firstly the bayesian regression is trained using the posterior update shown above. The neural network is trained using a similar loss function as the DQN. However the networks Q-value estimate is replaced by the MAP estimate of β resulting in

$$\theta = \theta - \alpha \nabla_\theta \big(Q_t - [\mu_n^T \phi_\theta(x_t)]\big)^2.$$

Note that these do not have to happen sequentially. In Azizzadenesheli et al. (2019) and this implementation the bayesian regression is updated less often than the neural network.

Finally to deal with the fact that reinforcement learning is a non-stationary problem the the bayesian regression is trained for scratch each time it is updated.

Bibliography

Abbeel, P., Coates, A., Quigley, M., Ng, A. Y., 2007. An application of reinforcement learning to aerobatic helicopter flight. Advances in Neural Information Processing Systems 19.

Azizzadenesheli, K., Brunskill, E., Anandkumar, A., 2019. Efficient exploration through bayesian deep q-networks. CoRR abs/1802.04412.
URL `http://arxiv.org/abs/1802.04412`

Barto, A. G., Sutton, R. S., Anderson, C. W., 1983. Neuronlike adaptive elements that can solve difficult learning control problems. IEEE Transactions on Systems, Man, and Cybernetics SMC-13 (5), 834846.

Bellemare, M. G., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., Munos, R., 2016. Unifying count-based exploration and intrinsic motivation. CoRR abs/1606.01868.
URL `http://arxiv.org/abs/1606.01868`

Bellman, R., 1957. Dynamic Programming. Dover Publications.

Evans, R., Gao, J., 2016. Deepmind ai reduces google data centre cooling bill by 40
URL `https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-c`

Hasselt, H. V., 2010. Double q-learning, 2613–2621.
URL `http://papers.nips.cc/paper/3964-double-q-learning.pdf`

Hastie, T., Tibshirani, R., Friedman, J., 2009. The elements of statistical learning: data mining, inference and prediction, 2nd Edition. Springer.

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M. G., Silver, D., 2017. Rainbow: Combining improvements in deep reinforcement learning. CoRR abs/1710.02298.
URL `http://arxiv.org/abs/1710.02298`

Irpan, A., 2018. Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

Lin, L.-J., 1993. Reinforcement learning for robots using neural networks. Tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. CoRR abs/1602.01783.
URL `http://arxiv.org/abs/1602.01783`

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M. A., 2013. Playing atari with deep reinforcement learning. CoRR abs/1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al., 2015. Human-level control through deep reinforcement learning. Nature 518 (7540), 529533.

Moerland, T. M., Broekens, J., Jonker, C. M., 2017. Efficient exploration with double uncertain value networks. CoRR abs/1711.10789.
URL `http://arxiv.org/abs/1711.10789`

O'Donoghue, B., Osband, I., Munos, R., Mnih, V., 2017. The uncertainty bellman equation and exploration. CoRR abs/1709.05380.
URL `http://arxiv.org/abs/1709.05380`

Osband, I., Roy, B. V., 2016. Why is posterior sampling better than optimism for reinforcement learning?

Powell, W. B., 2011. Approximate dynamic programming: solving the curses of dimensionality. Wiley.

Ross, S. M., 2014. Introduction To Probability Models. Elsevier Academic Press.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O., 2017. Proximal policy optimization algorithms. CoRR abs/1707.06347.
URL `http://arxiv.org/abs/1707.06347`

Shannon, C. E., March 1950. Programming a computer for playing chess. Philosophical Magazine 41 (314).

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., Dec. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al., 2017. Mastering the game of go without human knowledge. Nature 550 (7676), 354359.

Strehl, A. L., Littman, M. L., 2008. An analysis of model-based interval estimation for markov decision processes. Journal of Computer and System Sciences 74 (8), 1309 – 1331.
URL http://www.sciencedirect.com/science/article/pii/S0022000008000767

Strens, M., 2000. A bayesian framework for reinforcement learning, 943–950.

Sutton, R. S., Barto, A., 2018. Reinforcement learning: an introduction. The MIT Press.

van Hasselt, H., Guez, A., Silver, D., 2015. Deep reinforcement learning with double q-learning. CoRR abs/1509.06461.
URL http://arxiv.org/abs/1509.06461

Wang, Z., de Freitas, N., Lanctot, M., 2015. Dueling network architectures for deep reinforcement learning. CoRR abs/1511.06581.
URL http://arxiv.org/abs/1511.06581

Watkins, C. J. C. H., Dayan, P., 1992. Q-learning. Machine Learning 8 (3-4), 279292.