Table of Contents

List of Tables

List of Figures

Chapter 1

Theory

1.1 Introduction to RL Terminology

1.1.1 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 1.1. 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.

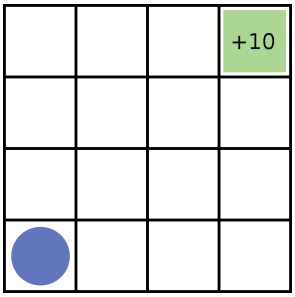


Figure 1.1: Initial State of Gridworld. The blue circle represents the player, while the green square represents the goal.

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 1.2.

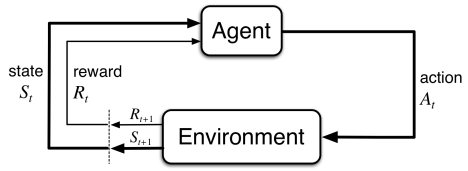


Figure 1.2: Visualization of the Agent and Enironment. The figure is from (Sutton and Barto, 2018, p. 48)

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.

1.1.2 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

$$p_{s,s'} = P(S_t = s' | S_{t-1} = s) \quad \text{for } s', s \in \mathbb{S} \tag{1.1}$$

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

$$P(s', r|s, a) = P(S_t = s', R = r|S_{t-1} = s, A = a) \tag{1.2}$$

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.

1.2 Dynamic Programming solutions to Tabular MDP's

1.2.1 Bellman Equation

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

$$G_t(s) = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}(A_t|S_t = s) \quad \text{where } 0 << \gamma < 1. \tag{1.3}$$

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

$$\pi = P(A = a|S_t = s). \tag{1.4}$$

The notation in equation 1.3 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

$$\max_{\pi} \mathbb{E}_{\pi}[G_t] \tag{1.5}$$

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 1.5 from a given state.

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|S_t = s_t] \tag{1.6}$$

Equation 1.6 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

$$v_\pi(s_t) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_{t+1} = s_{t+1}]. \tag{1.7}$$

Noting that

$$\mathbb{E}_\pi[R_t] = \mathbb{E}[R_t|A_t = a] = \mathbb{E}[r_t] = r_t \tag{1.8}$$

one can rewrite equation 1.7 as

$$\begin{aligned} &= r_{t+1} + \gamma\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s_{t+1}] \\ &= r_{t+1} + \gamma v_\pi(s_{t+1}). \end{aligned} \tag{1.9}$$

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.

$$\begin{aligned} Q(s,a) &= \mathbb{E}_\pi[G_t|S_t = s_t, A_t = a] \\ Q(s,a) &= r_t(a) + \gamma\mathbb{E}\big[v(S_{t+1})|S_t = s_t, A_t = a\big] \end{aligned} \tag{1.10}$$

In certain situations, which will be discussed at a later stage, equation 1.10 can be a more useful decomposition.

To conclude, a naive way to solve equation 1.5 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)

1.2.2 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

$$v_{k+1}(s) = \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] \tag{1.11}$$

and terminating the iteration when

$$\|v_{k+1} - v_k\| < \epsilon \quad for \quad \epsilon > 0. \tag{1.12}$$

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 $V_\pi(S_{t+1})$.

$$\pi'(s) = \arg\max_{a \in \mathbb{A}_t} \mathbb{E}\big[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a\big] \tag{1.13}$$

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 1.11 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:

$$v_{k+1}(s) = \max_{a \in \mathbb{A}_t} \mathbb{E}\big[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s, A_t = a)\big]. \tag{1.14}$$

The termination condition is given by equation 1.12 as in policy iteration. When the value function has converge a policy is given by

$$\pi(s) = \arg\max_{a \in \mathbb{A}_t} \mathbb{E}\big[R_{t+1} + \gamma v(S_{t+1}) | S_t = s, A_t = a\big] \tag{1.15}$$

The result of running value iteration on the gridworld example can be seen in figure 1.3

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 1.3: 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).

1.2.3 Limitations of Value and Policy iteration

There are many situations where value and policy iteration(equation 1.13 and 1.14) 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 $V(S_{t+1})$.

1.3 Reinforcement Learning solutions to Tabular MDP's

RL builds on a different approach than the methods discussed above. Consider the gridworld 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.

1.3.1 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 envi-

ronment. 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)

1.3.2 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 1.4 (Sutton and Barto, 2018, p. 86).

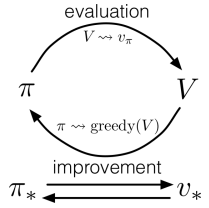


Figure 1.4: Visualization of GPI. Taken from p. 86 Sutton and Barto (2018)

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

$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[G_t - V_\pi(S_t)\big] \tag{1.16}$$

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.

$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[R_{t+1} + \gamma V_\pi(S_{t+1}) - V(S_t)\big]. \tag{1.17}$$

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 (1.10) and is called Q-learning.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{1.18}$$

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.

1.3.3 Exploration vs Exploitation

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

$$A_t = \arg\max_a Q_t(a) \tag{1.19}$$

The policy defined in equation 1.19 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

$$A_t = \begin{cases} \arg\max_a Q_t(a) & \text{with probability} \quad 1-\varepsilon \\ a \sim \text{Uniform}(\mathbb{A}) & \text{with probability} \quad \varepsilon. \end{cases} \tag{1.20}$$

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)

1.4 Deep Q-Learning

1.4.1 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 1.5A 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 1.5B shows that there are certain low price changes that have limited data. For these the same problem will occur.

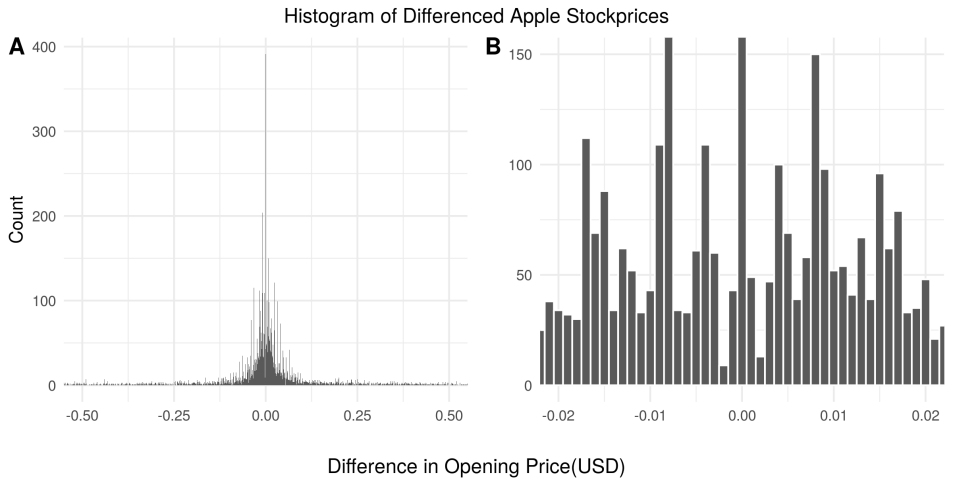


Figure 1.5: Differenced Apple Stock Opening Prices: A

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.

1.4.2 Nonlinear function approximation

Action value functions can be complicated functions so it can be desirable to have a non-linear 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

$$L_i(\theta_i) = \mathbb{E}_{s,a}\Big[\big(y_i - Q(s,a;\theta_i\big)^2\Big] \tag{1.21}$$

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 1.18).

$$y_i = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\right] \tag{1.22}$$

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

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a;s'}\Big(\Big[r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)\Big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big] \tag{1.23}$$

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

$$Z = \sigma(W^T X) \tag{1.24}$$

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

$$L(\boldsymbol{\theta}) = (y - \hat{y}(\boldsymbol{\theta}))^2 \tag{1.25}$$

while cross-entropy is

$$L(\boldsymbol{\theta}) = -y \log \hat{y}(\boldsymbol{\theta}). \tag{1.26}$$

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

$$w_k = w_k - \gamma \frac{\partial L}{\partial w_k} \tag{1.27}$$

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

$$\frac{\partial L}{\partial w_K} = \delta^K z_K \tag{1.28}$$

$$\frac{\delta L}{\delta w_k} = \delta^k z_{k-1} \tag{1.29}$$

$$\tag{1.30}$$

$$\text{where} \quad \delta^K = (y - z^K)^2 \tag{1.31}$$

$$\delta^k = \sigma'(w_k z_k) w_{k+1} \delta^{k+1} \tag{1.32}$$

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 1.22 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).

1.4.3 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 1.18. 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 1.33

$$L(y, \hat{y}) = \begin{cases} (y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq 1 \\ 2|y - \hat{y}| - 1 & \text{otherwise.} \end{cases} \tag{1.33}$$

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.

1.4.4 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

$$R_t + \gamma Q'(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a')) \tag{1.34}$$

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 1.35 to produce a Q-value per action that can be trained using the same method as a regular DQN.

$$Q(s, a) = v(s) + \left(A(s, a) - \frac{1}{|\mathbb{A}|} \sum_{a'} A(s, a') \right) \tag{1.35}$$

1.4.5 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.

1.5 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.

1.5.1 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

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}} \right]. \tag{1.36}$$

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 1.6. 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.

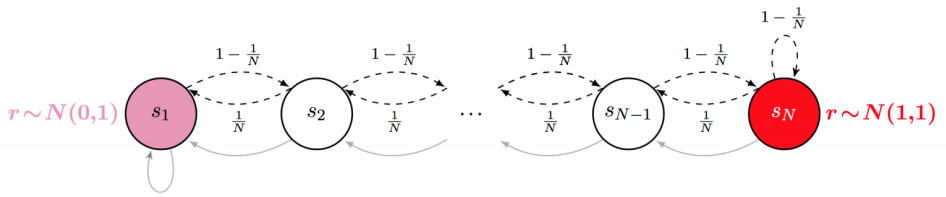


Figure 1.6: Chain environment. The figure is taken from Osband and Roy (2016)

The optimal policy is to always move right. However an agent following the policy in equation 1.36 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):

$$\max_a \mathbb{E}\Big[R_t + \gamma V(S_{t+1}) + \beta N(S, a)^{-\frac{1}{2}} \Big]. \tag{1.37}$$

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)

1.5.2 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.

$$Q^*(s_t, a_t) = \hat{R}^*(s,a) + \sum_{s_{t+1}, a_{t+1}} P^*(s_{t+1}|s_t, a_t) \max_{a'} Q^*(s_{t+1}, a_{t+1}) \tag{1.38}$$

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)

$$a_t = \arg\max_{a \in \mathcal{A}} Q^*(s_t, a) \tag{1.39}$$

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 1.7.

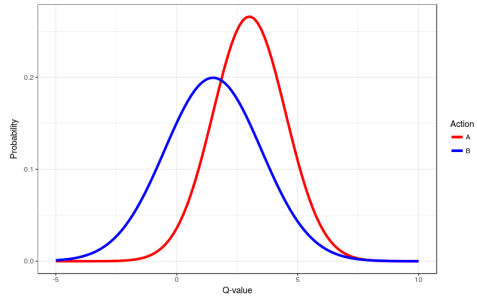


Figure 1.7: Posterior distribution of action-values: Despite action A having a higher expected value, the posteriors indicate that action B could potentially be the best action.

Figure 1.7 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 2

Conjugate Bayesian Linear 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 to investigate what models to use and what factors are important in a reinforcement learning setting.

2.1 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 for a single action can then be defined as

$$Q(X, a) = X\beta + \varepsilon \quad \text{where} \quad \varepsilon \sim N(0, \sigma^2) \tag{2.1}$$

with the response value defined as

$$Q(s, a) = r_t + \arg\max_{a'} Q(s', a'). \tag{2.2}$$

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 is purely an exploitation strategy, it is often coupled with the ε-greedy policy.

2.2 Bayesian Linear Q learning

To extend linear Q learning methods to Thompson sampling a bayesian perspective is required. To do this a prior distribution is placed over the regression parameters. Using bayes rule the posterior distribution of the parameters can be calculated and used to calculate the marginal distribution over Q.

$$p(\theta|Q, \mathcal{F}) \propto p(Q|\theta, \mathcal{F})p(\theta)$$
$$p(Q) = \int p(Q|\theta, \mathcal{F})p(\theta)d\theta$$

Q is a vector of all Q-values given the state X_t, θ denotes all parameters and $\mathcal{F}$ denotes all previous transitions. Since this is only used for Thompson sampling the value of the integral is not of interest. Instead it is the samples from $p(Q|\theta, \mathcal{F})$ that will be used to drive exploration.

2.3 Conjugate Bayesian Linear Q learning

TODO:Emphasize the assumptions in this chapter

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

2.3.1 Gaussian Prior with Known noise

To start consider the model used in Azizzadenesheli et al. (2019). As in frequintist regression the noise term ε is a zero mean gaussian distribution with variance σ_ε. In Azizzadenesheli et al. (2019) σ_ε is assumed known. This is rarely the case, but even if it is unknown it can be treated as a hyperparameter that has to be tuned to the environment.

A regression model is created per action, each with the same noise variance σ_ε and with a prior over every regression coefficient. The posterior Q-value for each action is expressed as

$$p(\beta_a|Q_a, \sigma_{\varepsilon_a}, \mathcal{F}) \propto p(Q_a|\beta_a, \sigma_{\varepsilon_a}, \mathcal{F})p(\beta_a)$$
$$p(Q_a|\sigma_{\varepsilon_a}, \mathcal{F}) = \int p(Q_a|\beta_a, \sigma_{\varepsilon_a}, \mathcal{F})p(\beta_a)d\beta_a.$$

A common conjugate prior for the coefficients is the gaussian distribution

$$p(\beta_a) = \mathrm{N}(\mu_a, \sigma_\varepsilon \Lambda_a^{-1})$$

where Λ_a is the precision matrix. With this choice of prior the posterior distribution of β_a is still gaussian with a closed form update for the distribution parameters. Given new state-action combinations X and target action-values Q^a the posterior can be updated using

$$\begin{aligned} \Lambda_{a_n} &= X^T X + \Lambda_{a_0} \\ \mu_{a_n} &= \Lambda_{a_n}^{-1}(\Lambda_{a_0}\mu_{a_0} + X^T Q_a). \end{aligned} \tag{2.3}$$

where the $_0$ denotes the prior and $_n$ denotes the posterior parameters. The development of these updates can be found in the appendix(TODO:reference). This model will be refered to as a bayesian normal model(BN).

With this setup actions can now be picked by Thompson sampling. For each action sample β values from its posterior distribution and a noise term from $N(0, \sigma_\varepsilon)$. These sample values are then used in the regression equation 2.1 to get a posterior sample from Q_a. Finally chose the action with the highest sampled Q-value.

When calculating the target Q-value Azizzadenesheli et al. (2019) does not use Q-value samples. Instead the MAP estimate of β is used which in this case is μ.

2.3.2 Propagating Variance

Using the MAP estimate means that the targets are calculated by

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

This does not correctly encorporate the target variance. To see why recall the definition of the Q-value

$$\begin{aligned} Q_t &= \mathbb{E}[G_t] = \mathbb{E}[r_t + r_{t+1} + \dots] \\ &= \mathbb{E}[r_t + Q_{t+1}] = \mathbb{E}[r_t + Q_{t+1}] \end{aligned}$$

This results in the regression problem $\mathbb{E}[r_t + Q_{t+1}] = X\beta_a$. However, since the expected reward is unknown this cannot be used. Instead one has access to the sample rewards from the environment. Asymptotically the mean of the samples approaches the expected value so this can be treated as a regression task with a noise term

$$\mathbb{E}[r_t + Q_{t+1}] = X\beta_a$$
$$r_t + Q_{t+1} = X\beta_a + \varepsilon$$

where ε accounts for the difference between the sample and the mean, $r_t - \mathbb{E}[r_t] + Q_{t+1} - \mathbb{E}[Q_{t+1}]$. This implies that the target used must be a sample from the posterior of Q_t not its expected value $X\mu_a$ as used in Azizzadenesheli et al. (2019).

The result of this is that the known noise model only includes 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."

Based on the above a better choice of target is

$$y = r_t + \max_a(X_{t+1}\beta + \varepsilon)$$

where β is sampled from its posterior and ε from the gaussian noise distribution. However the variance of β, as seen in equation 2.3, is independent of the target. One way to include a variance term that is dependent on the target is to include σ_ε as an unknown parameter.

2.3.3 Normal Prior with Unknown noise

Including σ_ε as an unknown parameter resuts in the new posterior

$$p(\beta_a, \sigma_{\varepsilon_a}|Q_a, \mathcal{F}) \propto p(Q_a|\beta_a, \sigma_{\varepsilon_a}, \mathcal{F})p(\theta)$$
$$p(Q_a|\mathcal{F}) = \int p(Q_a|\beta_a, \sigma_{\varepsilon_a}, \mathcal{F})p(\beta_a, \sigma_{\varepsilon_a})d\beta_a d\sigma_{\varepsilon_a}.$$

The conjugate priors for this setup are

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

with the posterior update

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

The development of these formulas can be found in the appendix (TODO:reference). This model will be refered to as the Bayesian Normal Inverse Gamma model(BNIG).

2.3.4 Testing Variance Propagation

To ensure that this method is propagating uncertainty consider a simple example from Osband et al. (2018). Consider a MPD with two states. The initial state allows only one action that deterministicly leads to state 2 with no reward. State 2 is a terminal state with a known posterior distribution. If a RL method properly propagates uncertainty the posterior distribution of state 1 should match state 2 as long as $\gamma = 1$.

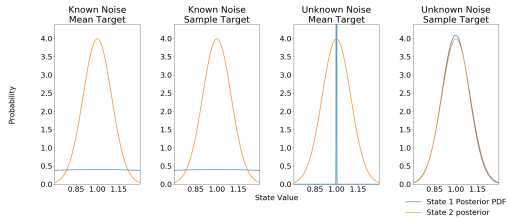
Both models were tested with both MAP and sample targets. The priors for the BN models were set to

$$
\begin{aligned}
\beta &\sim N(0, 10^3) \\
\sigma^2 &= 1
\end{aligned}
$$

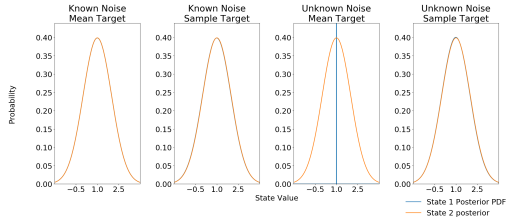
and the priors for the BNIG models were set to

$$
\begin{aligned}
\beta &\sim N(0, 10^3) \\
\sigma^2 &\sim InvGamma(1, 1).
\end{aligned}
$$

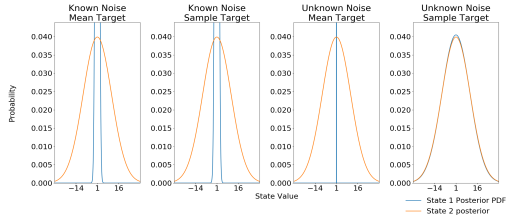
Three MDP's were set up with a known posterior of $N(1, 0.1), N(1, 1)$ and $N(1, 10)$ respectively. The results are seen in figure 2.1.



(a) $N(1, 0.1^2)$



(b) $N(1, 1^2)$



(c) $N(1, 10^2)$

Figure 2.1: Variance Propagation On 2 State Toy Example: The blue lines show the models Q-value posterior distribution while the orange lines show the target posterior distribution. Only the BNIG model with sample targets is able to correctly estimate the target in all cases.

The results summarized in figure 2.1 showed that all models were able to correctly estimate the mean. However the target variance was only correctly estimated by the the BNIG model with sample targets. The BNIG model with the MAP target leads to the correct mean but dramatically understimates the variance. This would be an expected result if the model is approximating $\mathbb{E}[Q]$ instead of Q. The BN model is only correct for both the mean and sample target if the hyperparameter ε is set to the correct variance. In an unknown and more complex environment this is unlikely to be possible. However with enough hyperparameter tuning one could argue that this can lead to good results which might explain the results achieved in Azizzadenesheli et al. (2019).

Based on these results further developments are focused on the BNIG model with sampled targets.

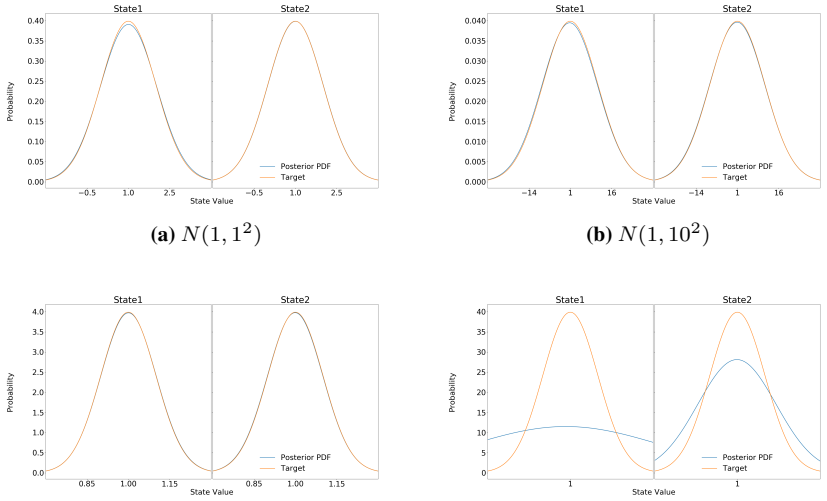
2.4 Variance Propagation

2.4.1 Over Multiple States

The setting above is equivalent to regular regression. In a RL setting the variance needs to be propagated to further states. To test that this is still the case with the BNIG model consider a modification to the environment where an extra state is placed between the initial and terminal state. This state has the same dynamics as earlier. It deterministically transitions to the next state with zero reward over the transition. The correct posterior for each state is then the target posterior in the terminal state.

The same priors as earlier are used on 4 different target posteriors. The results are summarised in figure 2.2.

TODO:Add details around these experiments in the appendix



(a) $N(1, 1^2)$ **(b)** $N(1, 10^2)$

(c) $N(1, 0.1^2)$ **(d)** $N(1, 0.01^2)$

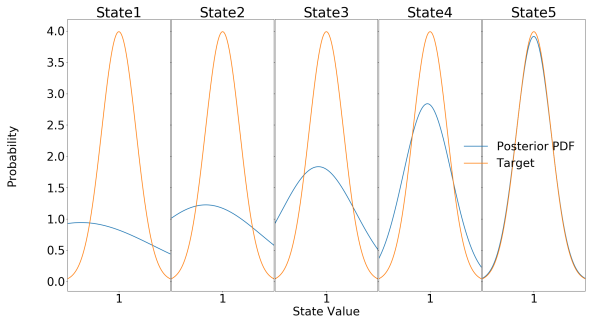
Figure 2.2: Variance Propagation On 3 State Toy Example: The models posterior estimate for the two first states are shown. The third state is the terminal state that returns a sample from the target distribution.

Figure 2.2 shows that for larger variance targets the propagation correctly updates the state 1 variance. However the low variance targets results in an overestimation over the variance in state 1. Running this experiment for more iterations does lead to a better approximation implying that the problem lies in the the convergence rate for different posteriors.

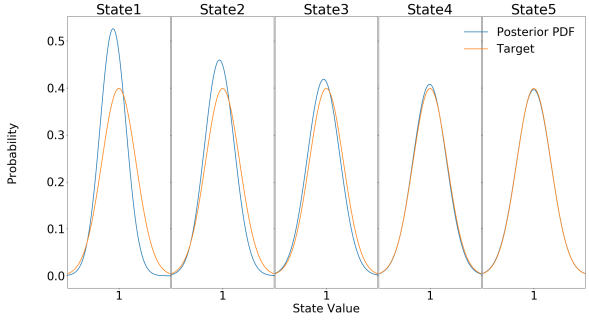
One possible reason for this is that the posterior representing the 0.01 standard deviation posterior is more sensitive to small changes in in its parameters than the larger variance posteriors. In other words small changes in the parameters for a distribution with low

variance leads to large changes in the variance of the posterior. Since the learning is happening in an online fashion the first estimates of the posterior will likely have large error. This effect is amplified for state 1 since it is training on the large error state 2 posterior. (TODO:: Confirm/source that this is actually the case)

By extending the toy example to even more states as in figure 2.3 one can see that this problem increases the further the variance needs to be propagated. Even large variance targets will fail to correctly propagate given enough states.



(a) 6 States with $N(0.1, 1^2)$



(b) 6 States with $N(1, 1^2)$ target

Figure 2.3: Failure of variance propagation over many states: (**a**) shows that the error in estimation close to the terminal state leads to failure in the estimation of the posterior of the initial states. In (**b**) the seemingly correct estimation close to the terminal state still does not prevent errors closer to the initial state.

This issue will be refered to as the speed of propagation. It encompasses the problem of quickly propagating variance estimates from downstream states back to states which are far from the environments reward.

TODO:: Define speed of propagation. Maybe a seperate subsection that plots how the error changes with length of chain. How do we measure the error between two distributions?

KL Divergence?

2.4.2 Speed of propagation

Inorder to improve the speed of propagation insight is required into what causes the issue. In this section two causes and methods for counteracting them are discussed. However these are not necessarily the only two causes of slow propagation.

TODO:These argumentations are weak and require sources or ”proof”.

The first cause is best explained using an example. Consider the 3 state toy example. State 1 can only converge to the correct distribution once State 2 has converged. If more states are added, State 1 will only converge once all the states between it and the terminal state have converged. The longer the chain is, the more iterations are required for state 1 to converge. This issue is the same issue faced with the expected Q value in regular RL which is dealt with through a bias-variance trade-off (TODO:Include n-step in theory). Similarly extending the 1-step update to an n-step update one will increase the speed of variance propagtation with the downside of introducing more variance to the σ_ε estimate.

The second cause is a result of no longer using a step size when updating the Q-values using bayesian updates. In regular Q-learning the step size can be viewed as the weighting of new data relative to the current model. This effect is also found in the bayesian setting in the posterior update that combines the prior and the new data. However, in regular Q-learning the step size also leads to the model forgetting old data(Sutton and Barto (2018)TODO:find this page). This is not the case for the bayesian models previously described.

In the bayesian regression setting described the prior is always the previous posterior. In simple terms the model assumes all data to be equally important. Recalling that the prior used is the previous posterior, a prior based on many datapoints will have a bigger Mˆ L based upon the data it has gathered up until episode L. To offer a representative approaimpact on the posterior than a single new datapoint. This is a problem since a reinforcement learning problem is almost always non-stationary due to changes in policy. With this weighting scheme the new data points which are more relevant to the current target are weighted the same as a datapoint collected based on the initial priors.

Counteracting this effect while retaining the bayesian regression model is not trivial. The solution used in Azizzadenesheli et al. (2019) is to define a hyperparameter T_{bayes} and train a new bayesian regression model from scratch using targets from the old model every T_{bayes} steps. However this is a computationally heavy step and can greatly increase the run time of the algorithm for problems with many predictors.

To avoid this a concept called exponential forgetting is used. This is first mentioned in Dearden et al. (1998) but with no reference and no explanation to what it is. (TODO:write down best source). The method reduces the impact of previous data by exponentially decaying old data.

Rather than keeping track of the entire dataset used one can consider the terms used to update the posterior seen in equation 2.4. The terms $X^T X$, $X^T y$, $y^T y$ and n are all

proportional to the number of predictors rather than the number of datapoints. The terms can then be exponentially decayed by using the updates

$$(X^T X)^{(k+1)} = \alpha X^T X^{(k)} + x^T x \tag{2.4}$$

$$(X^T y)^{(k+1)} = \alpha X^T y^{(k)} + x^T y \tag{2.5}$$

$$(y^T y)^{(k+1)} = \alpha y^T y^{(k)} + y^T y \tag{2.6}$$

$$n^{(k+1)} = \alpha n^{(k)} + b \tag{2.7}$$

where α is the decay rate and the b is the batch size. The result of this will be a regression that is based on the $\frac{1}{1-\alpha}$ previous datapoints with more recent datapoints being weighted more than older ones. α should be set close to 1 to avoid instantly forgetting previous observations.

2.5 Deep Exploration

TODO:When I started reading and writing this I realized that propagating the variance is already performing deep exploration. Why does sampling new parameters every episode work? It could be related to this environment so I tested on acrobot and cartpole. Sampling per episode works better on acrobot. Sampling per episode leads to no difference in evaluation performance on cartpole, but reduces performance during training. I think this indicates it is not only related to the corridor environment. I think it might be better because it indirectly decreases our certainty about the policy. If the model is pretty sure moving left is best it is very unlikely that the model will move right multiple steps in a row. Sampling per episode increases this probability. We can discuss this next meeting.

Osband et al. (2018) introduces a term called deep exploration, the idea that a good exploration strategy has to consider how an action effects the future information gain.

2.6 Performance on Linear RL Problem

Up to now the model has only been tested on simple problems with known posteriors and no active decision making. To test how well this generalizes to a more realistic RL environment consider a variant on the corridor environment introduced in the theory section in figure 1.6.

Consider a chain of length N where the goal is to move from the initial state on the left side of the corridor to the final state on the right side of the corridor in N steps. A reward of 1 is given in a final state, moving left gives a reward of $\frac{1}{10N}$ and otherwise the agent recieves no reward.

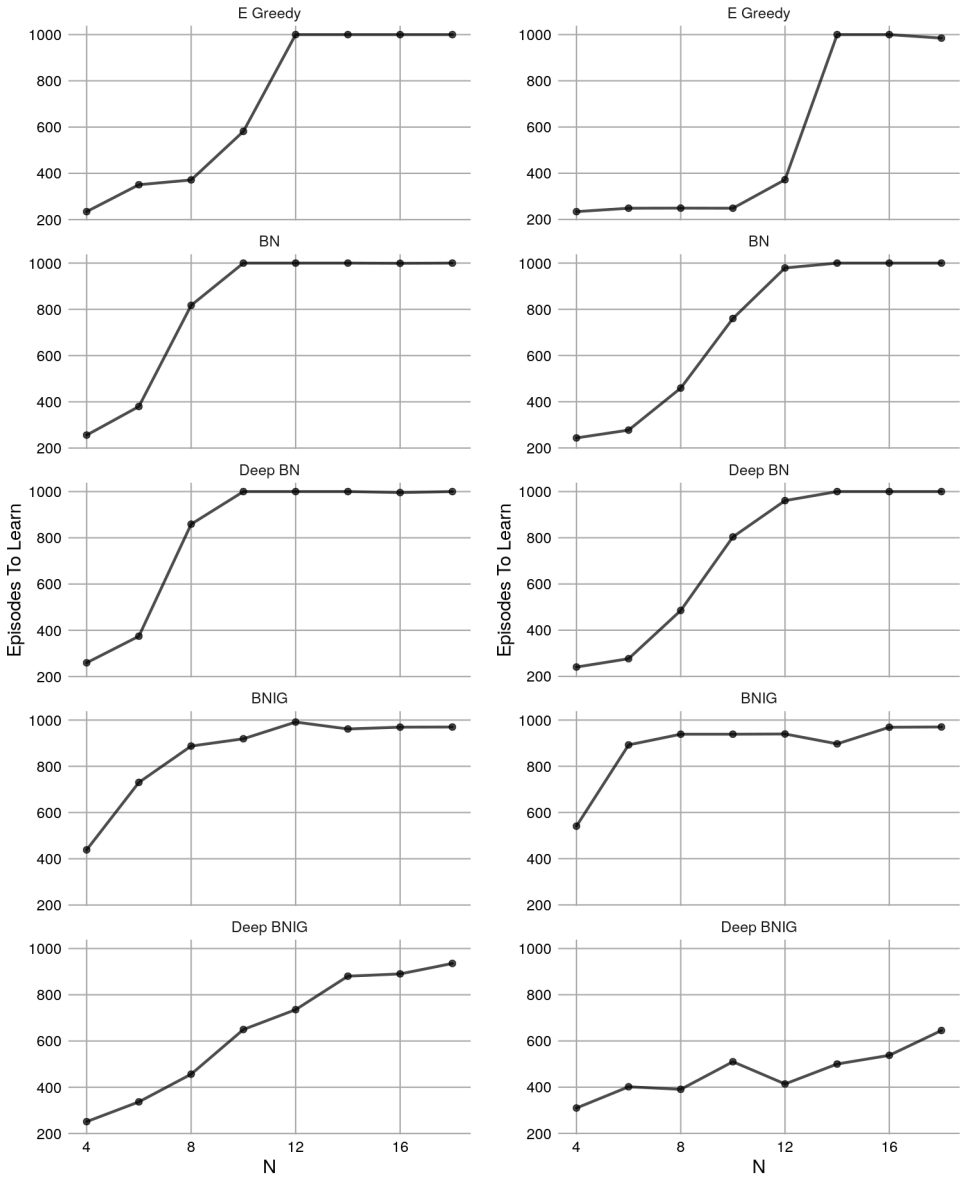
TODO:Add regret to theory section

All the methods discussed so far were tested on this enviroment with different chain lengths. This includes the BN and BNIG model with both regular and deep exploration.

In addition a linear Q-learning method following an ε-greedy exploration scheme was tested. The hyperparameters for each model were found by manually testning different combinations. The priors that gave good propagation for the toy examples were used as a starting point. A table of the hyperparameters can be found in the appendinx in table TODO:.

As in Osband et al. (2018) models were compared by the average number of episodes required to ”learn” the enivorment. In this case the enivorment is considered learned when the running average regret of the last 100 episodes drops below 0.1.

TODO:These results are based on 10 attempts per length, this should be run for 50-100 attempts for nicer plots.



(a) 1 step update

(b) 3 step update

Figure 2.4: Linear Model Performance on Corridor Environment: (**a**) shows that the only method to out perform is the Deep BNIG model. Increasing to a 3-step update in (**b**) improves the performance of all models but the largest increase in performance seems to be Deep BNIG.

The key result from figure 2.4 is that the ε-greedy method outperforms all methods apart

from the BNIG model with deep exploration. However there are a few interesting differences in the results between the bayesian models and their exploration methods.

The regular BNIG model performs worse than the regular BN model. One reason for this could be the constant noise term allowing for exploration even when the model is certain about it's current estimate. If the BNIG model quickly learns that a left moving policy gives a reward, the variance for the left action might approach zero before it gets enough data showing that a right moving policy gives a higher reward. A BN model will result in the same issue, but the small additional variance might provide enough exploration to manage a slightly longer corridor. This effect could be counteracted in the BNIG model by making the variance term σ_a^2 dependent on the state.

Adding deep exploration has seemingly no effect on the BN model but leads to far better results on the BNIG model. This could be due to the BN model not propagating variance which quickly leads to a low variance model. Once the variance is low the difference between sampled action-value functions is also small so sampling each step versus sampling each episode will lead to similar actions.

Finally, increasing from a 1-step to a 3-step update leads to better performance across all models. However for all models apart from deep BNIG the improvement is increasing the longest learnable corridor by around 2-4 states. In deep BNIG's case the improvment is an increase of atleast 8 states and possibly more. Once again this is probably due to the BNIG model being the only one actually propagating uncertainty. By increasing the step update in the BNIG model one is not only decreasing the bias of the expected value but also decreasing the bias of the variance. In other words the n-state update not only improves the value estimation but also improves the exploration, resulting in a larger improvment than in other methods.

2.7 Temp? State Dependent Inverse Gamma Distribution

TODO:This might not be worth including but I'm writing it down so we can discuss if it's worth working on.

TODO:I originally worked on this to stabalize beta (which no longer is a problem) but realized it could be used with a model. I've tested a bit on corridor with a linear model with OK results, but need to test more to actually conclude that this works. But I wont work more on this until we have discussed this and after I've come further on my thesis.

The variance of a state should be dependent on the state, which is not the case with in BNIG model. To fix this the inverse gamma distribution parameters should be dependent on the state. With some linear algebra it is possible to rewrite the b term in terms of the next state variance and temporal difference. First recall the parameter updates

$$a_n = a_0 + \frac{n}{2} \tag{2.8}$$

$$b_n = b_0 + \frac{1}{2}(Q^T Q - \mu_n^T X^T X \mu_n) \tag{2.9}$$

where some terms are removed as a prior mean of 0 is used.

Denote y as the target, $\hat{Q}$ as the action-value of the next state and Q the action-value of the current state. The target is then distributed by $N(r + \hat{Q}, \sigma_{a'}^2)$. Using the moments of a normal distribution one finds

$$\begin{aligned}
\mathbb{E}[Q^T Q] &= n\sigma_{a'}^2 + (r + \hat{Q})^T (r + \hat{Q}) \\
\mathbb{E}[b_n] &= b_0 + \frac{1}{2}\left(n\sigma^2 + (r + \hat{Q})^T (r + \hat{Q}) - Q^T Q\right) \\
&= b_0 + \frac{1}{2}\left(n\sigma_{a'}^2 + (r + \hat{Q} - Q)^T (r + \hat{Q} + Q)\right) \\
&= b_0 + \frac{1}{2}\left(n\sigma_{a'}^2 + \delta^T (r + \hat{Q} + Q)\right) \\
&= b_0 + \frac{1}{2}\left(n\sigma_{a'}^2 + 2\delta^T y - \delta^T \delta\right)
\end{aligned} \tag{2.10}$$

where δ is the temporal difference and the last line reduces the number arguments required for the calculation. This leaves an equation that is dependent on the variance of the next state and the temporal difference error. With this setup σ_a^2 can be replaced by a model $\sigma_a^2 = X\beta_{\sigma_a}$ that trains on targets $\sigma_{a'}^2 + 2\delta^T y - \delta^T \delta$.

Chapter 3

Neural Linear Bayesian Regression Model

A major drawback of the methods discussed in chapter 2 is that they are all linear models. These cannot perform well in environments with complex non-linear relationships between state-action pairs and Q-values without significant feature engineering. Recent developments in the RL field focus on deep RL(Mnih et al., 2015, 2016; Silver et al., 2017) where neural networks are used as encode these relationships have allowed successfull results on complex games. As such it it would be beneficial to be able to combine the methods from chapter 2 with more complex models. This chapter attempts this and tests the new model on multiple complex environments with comparisons to other popular deep RL methods.

3.1 Combining Bayesian Q-learning with Neural Networks

3.1.1 Neural Linear Model

Without significant feature engineering a linear model cannot generalize to more complex non-linear relationships between state-action pairs and their Q-values. However using bayesian methods with non-linear models can be difficult and computationally heavy. Riquelme et al. (2018) compared a large array of bayesian models on a set of bandit environments. They found that accurate complex models often performed worse than simpler approximate methods. The suggested reason for this is that complex models require more data and training to acheive reasonable variance estimates. Since RL is an online task this can lead to miscalibrated variance early on in the training process that leads to worse results.

Emperically Riquelme et al. (2018) finds what they coin as a neural linear model to work

best. The model consists of using a neural network as a basis function that is used as the covariates to a linear bayesian regression model. 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. The error in the bayesian regressions point estimates is backpropagated through the neural network to learn a useful basis function. Note that this means the bayesian regression no longer incorporates all the uncertainty since the above assumes no uncertainty in the $\phi(X)$ encoding. Riquelme et al. (2018) suggests that error that comes with this assumption is counteracted by the models stable uncerainty estimates.

This setup allows the application of the methods discussed in chapter 2 in more complex environments. It is also this method Azizzadenesheli et al. (2019) follows in their application of the BN model to more complex models.

3.1.2 Bayesian DQN Models

Based on the results of Riquelme et al. (2018) this thesis attempts to combine neural networks and the BNIG model through a neural linear setup. To start off a summary of the archicture used in Azizzadenesheli et al. (2019), called the BDQN, is provided. This is used as a base which will modified to fit with the BNIG model and thus allow for better variance propagation.

The BDQN architecture starts with the same archicture as the standard DQN architecture(Mnih et al., 2015). The final layer of a DQN is a linear layer which means it can be replaced by any linear model. Azizzadenesheli et al. (2019) replaces this with a BN model. This model is trained using the posterior updates described in chapter 2 in equation 2.3. The neural network is trained using the loss function

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

The only difference tzo a regular neural network is that the networks output estimate is replaced by the MAP estimate of β. One could replace the MAP estimate by samples from the posterior Q. However as shown in the linear case this should have no effect on the final estimate but does have a higher computational cost than simply using that MAP estimate.

These two training processes do not have to happen sequentially. In Azizzadenesheli et al. (2019) the neural network is updated as frequently as in the original DQN implementation, while the bayesian regression trained from scratch every 10,000 steps on either 100,000 datapoints or the entire replay buffer if it contains less. This is done to handle the non-stationarity of the task.

3.1.3 From BDQN to BNIG DQN

The downside retraining the bayesian regression is that this is computational heavy, especially considering that the final layer in the neural network consists of 512 neurons, meaning the update requires matrix arithmetic with a 512 by 512 matrix. On top of this using a BNIG model requires the target Q-values to be sampled from the posterior. This means every 10,000 steps 100,000 new samples must be drawn which requires a new set of matrix arithmetic of the same magnitude. Instead this thesis considers the exponential forgetting method. However implementing this requires some extra considerations to ensure that the model remains stable.

Recall that the classic DQN has one online network that is updated each training step and one target network that is updated occaisonally to match the online network. Mnih et al. (2013) found that using this target network to calculate the regression target helped stabalize the algorithm.

With exponential forgetting the bayesian regression is trained continuosly. However, using the online bayesian regression method to calculate targets based on the output from the target network will lead to instability. The regression model can train on thosands of datapoints from the online network between network syncs. Using different network encodings for the target and prediction with the same bayesian regression will lead to a different results which will artificially increase the loss. An increased loss leads to larger network changes which amplifies the effect leading to instability.

To deal with this issue the same setup that is used for the networks is used for the regression models. Two bayesian regression models are created, one online and one target. The target model is used to calculate the target action-values while the online model is used for decision making. The target model is then updated to the parameters of the online model when the target network is updated. This decreases the loss and resulted in a stable network.

With this approach the only required change to transform this to a BNIG setup is to swap out the bayesian model used in the BDQN with the BNIG model and sample action-values from the posterior for both the online and target models. The resulting algorithm is sum-

marized in the pseudocode below.

Algorithm 1: BNIG DQN

```
Initialize variables according to algorithm 2
for episode = 1, M do
    Initialize environment
    s_1 = initial environment state
    Sample β_a, σ_a for a ∈ A
    for t=1, T do
        a_t = arg max_a ( φ(s_t)β_a + ε_a )
        Execute action a_t and observe reward r_t and state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Train online network using algorithm 3
        Train online model using algorithm 4
        if t mod T_sync == 0 then
            Set target network weights to online network weights
            Set target BNIG parameters to online BNIG parameters.
        end
    end
end
```

Here ε_a represents a sample from $N(0, \sigma_a^2)$ and T_{sync} is a hyperparameter that defines how often to sync the online and target network and model. The initialization process in algorithm 2 sets up the models, networks and experience buffer required.

Algorithm 2: BNIG DQN Initialization

Initialize experience buffer $\mathcal{D}$ with capacity N
Initialize neural network ϕ with random weights θ
Initialize target neural network $\phi*$ with weights $\theta^* = \theta$
Initialize $|\mathcal{A}|$ BNIG models with priors $\mu_a = zeros(p, 1)$, $\Sigma_a = diag(p, p)$, $\alpha_a = 1$, $\beta_a = 0.01$ Initialize target BNIG models with priors $\mu_a^* = \mu_a^*$, $\Sigma_a^* = \Sigma_a^*$, $\alpha_a^* = \alpha_a^*$, $\beta_a^* = \beta_a^*$.

Finally the training procedure for the network is defined in algorithm 3 and algorithm 4.

Note that the * notation denotes values from the target model and network.

Algorithm 3: BNIG training

if t ***mod*** $T_{train} == 0$ **then**
  Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
  Sample β_a, σ_a for $a \in \mathcal{A}$ for each transition
  $a' = \arg\max_a \left(\phi(s_t)\beta_a + \varepsilon_a \right)$ for each transition
  Sample $\beta^*_{a'}, \sigma^*_{a'}$ for a' in each transition
  Set $y_j = \begin{cases} r_j & \text{for terminal} \quad s_{j+1} \\ r_j + \gamma\left(\phi^*(s_t, a')\beta^*_{a'} + \varepsilon^*_{a'} \right) & \text{else} \end{cases}$
  Update BNIG model parameters using posterior update equation 2.4
end

Algorithm 4: BNIG training

if t ***mod*** $T_{train} == 0$ **then**
  Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
  $a' = \arg\max_a \left(\phi(s_t)\mu_a \right)$ for each transition
  Set $y_j = \begin{cases} r_j & \text{for terminal} \quad s_{j+1} \\ r_j + \gamma\phi^*(s_t, a')\mu^*_{a'} & \text{else} \end{cases}$
  Optimize the network weights over the Huber loss (equation 1.33)
end

3.2 BNIG DQN Results

In this section the BNIG DQN is tested on a variety of environments and compared to a regular DQN.

To ensure a correct implementation of the DQN and a fair comparison to the BNIG DQN the python package Dopamine(Castro et al., 2018) was used. This package contains an implementation of a DQN that matches the baselines of the Mnih et al. (2015). The BNIG DQN was implemented ontop of this DQN implementation.

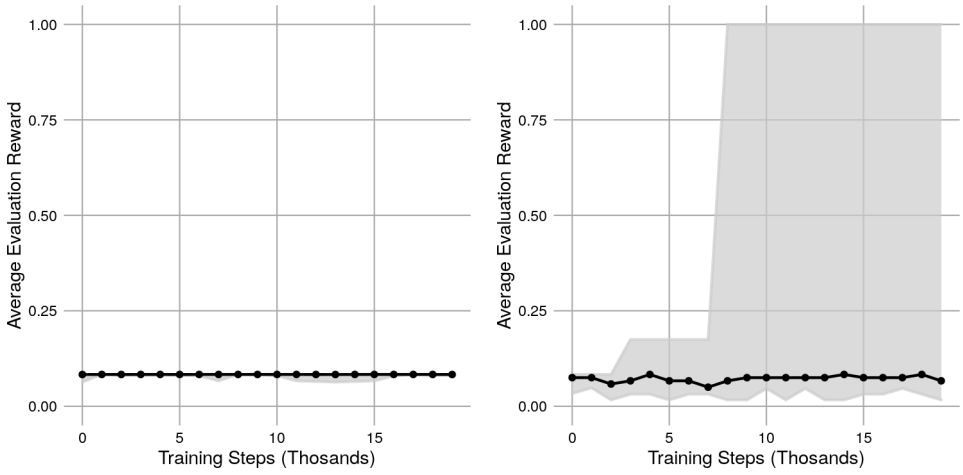
The Dopamine package has hyperparmeters for the DQN on many of the tested environments. In some of the environments no hyperparemeters were provided or the ones given performed badly. In these cases, to avoid bias towards BNIG DQN, the hyperparemeters were found by testing different combinations with the DQN. Only after good results were acheived using the DQN the BNIG DQN was run with the same hyperparameters. This means no tuning was done on the hyperparameters based on the BNIG DQN results. An overview of the hyperparmeters per environment is provided in the appendix.

The comparison of the two methods is based on their performance on the environment relative to the number of samples it has trained on. To fairly compare the methods during training the experiments the results are gathered from an evaluation stage. The evaluation

stage is run every time one wants to measure the performance of the method and consists of calculating the average total reward over a few episodes.

In this evaluation phase the model and network are not trained. The DQN agent acts greedily without any probability of doing an action. The BNIG DQN acts greedily with respect to it's MAP estimate. This means it acts based on the mean β values and no noise.

3.2.1 Corridor



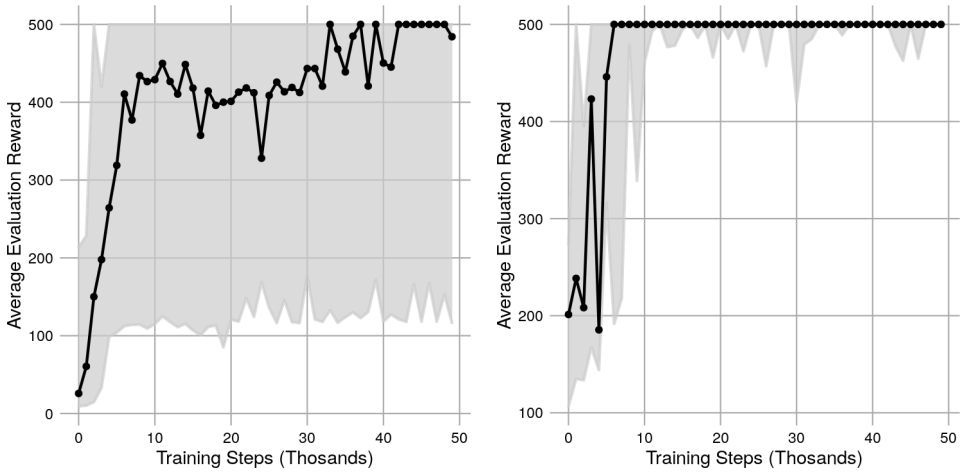
(a) DQN **(b)** BNIG DQN

Figure 3.1: DQN and BNIG DQN Performance on Corridor: Plots show the median performance over 10 different attempts. The shaded area covers 80% of the total reward over all attempts.

3.2.2 Cartpole

The environment considered is a classic toy example called cartpole. The environment was first proposed in Barto et al. (1983) and is easily available through the Open AI gym environment (Brockman et al., 2016a). TODO:figure to illustrate The task consists of balancing a pole on top of a cart. If the pole falls over 15 degrees from the upright position or if the cart goes too far (2.4 units from the center) to the left or right the game is terminated. The game is also terminated after 500 timesteps. Every timestep a reward of $+1$ is received. There are two actions, the cart can be pushed to the left or to the right at every timestep. There are 4 input variables that define the state: the pole angle from vertical, the position of the cart and the derivative of both variables.

The environment implementation from OpenAI Gym was used (Brockman et al., 2016b). Each method was run for 50,000 steps with 10 different seeds and every 1000 steps an evaluation phase is run over 1000 steps. The results are summarized in figure 3.2.



(a) DQN **(b)** BNIG DQN

Figure 3.2: DQN and BNIG DQN Performance on Cartpole: Plots show the median performance over 10 different attempts. The shaded area covers 80% of the total reward over all attempts.

A policy that does not balance the pole lasts about 10 to 20 frames which is equivalent to a reward of 10 to 20. Figure 3.2 shows both methods are able to find successfull balancing policy. However, while the max performance of the DQN matches the BNIG DQN, the latter acheives much more stable results. Breaking down the results per seeds (figure A.2) one can see that the spread is caused by the instability of DQN after having found an optimal policy in some attempts and failing to find the optimal policy in others. The BNIG BDQN however always finds the optimal policy however can face some detoriation in performance towards the end of the experiment.

3.2.3 Acrobot

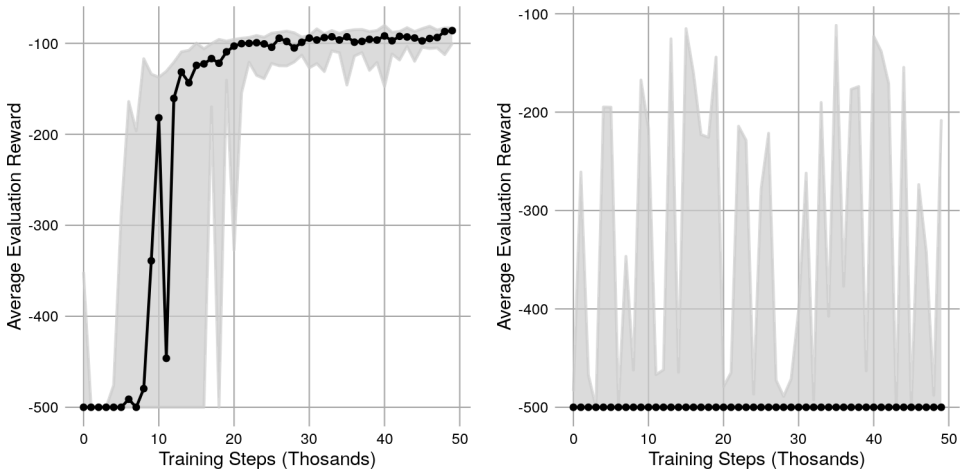
Another common toy example is called acrobot. The experiment is first described in Hauser and Murray (1990) and put into an RL context in Sutton (1996). The environment consists of a robot arm with two joints, one fixed to the origin and one with an actuator. The arm starts hanging downwards under the origin. The goal is to get the end of the robot arm over a line above the origin which terminates the episode. A reward of -1 is given each timestep. There are three actions, apply 1 torque left, 1 torque right or no operation. The state input is

$$[\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \dot{\theta}_1, \dot{\theta}_2]$$

where θ_1 is the angle of the first joint and θ_2 is the angle of the second joint relative to the first.

TODO:figure to illustrate

The environment implementation from OpenAI Gym was used (Brockman et al., 2016b). Each method was run for 100,000 steps with 10 different seeds and every 2000 steps an evaluation phase is run over 2000 steps.



(a) DQN **(b)** BNIG DQN

Figure 3.3: DQN and BNIG DQN Performance on Acrobot: Plots show the median performance over 10 different attempts. The shaded area covers 80% of the total reward over all attempts.

A score above -500 indicates the agent was able to lift the robot arm over the required threshold. As seen in figure 3.3 both agents manage to do this. However, while the DQN model find a stable policy leading to an average reward of -100, the BDQN struggles to stay above -500. A deeper look into the per seed results (figure 3.3) shows that the BDQN acheives an average reward of around -100 every seed. However, it is unable to keep this policy over more than two iterations before falling to a -500 reward again. There are many possible reasons. I can be caused by a too high learning rate, a miscalibrated exponential memory or a miscalibrated target update time period to a name a few.

TODO:Try some hyperparameter tuning to get BNIG to work on acrobot. I have not been successfull with this yet. Maybe there is another reason this is failing? Note that this worked well when the bayesian regression was trained from scratch.

3.2.4 Atari

TODO:In progress, looks promising

Bibliography

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, arXiv:1802.04412v2.

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, 834–846. doi:10.1109/tsmc.1983.6313077.

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, arXiv:1606.01868.

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

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W., 2016a. Openai gym. CoRR abs/1606.01540. URL: http://arxiv.org/abs/1606.01540, arXiv:1606.01540.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W., 2016b. Openai gym. arXiv:arXiv:1606.01540.

Castro, P.S., Moitra, S., Gelada, C., Kumar, S., Bellemare, M.G., 2018. Dopamine: A Research Framework for Deep Reinforcement Learning URL: http://arxiv.org/abs/1812.06110.

Dearden, R., Friedman, N., Russell, S., 1998. Bayesian q-learning, in: Aaai/iaai, pp. 761–768.

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, arXiv:1509.06461.

Hasselt, H.V., 2010. Double q-learning , 2613–2621URL: 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. 2 ed., Springer.

Hauser, J., Murray, R.M., 1990. Nonlinear controllers for non-integrable systems: the acrobot example. 1990 American Control Conference doi:10.23919/acc.1990.4790817.

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, arXiv:1710.02298.

Lin, L.J., 1993. Reinforcement learning for robots using neural networks. Technical Report. 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, arXiv: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. arXiv: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, 529–533. doi:10.1038/nature14236.

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, arXiv:1711.10789.

Osband, I., Aslanides, J., Cassirer, A., 2018. Randomized prior functions for deep reinforcement learning , 8617–8629URL: http://papers.nips.cc/paper/8080-randomized-prior-functions-for-deep-reinforcement-learning.pdf.

Osband, I., Roy, B.V., 2016. Why is posterior sampling better than optimism for reinforcement learning? arXiv:arXiv:1607.00215.

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

Riquelme, C., Tucker, G., Snoek, J., 2018. Deep Bayesian Bandits Showdown: An Empirical Comparison of Bayesian Deep Networks for Thompson Sampling. arXiv e-prints .

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, arXiv:1707.06347.

Shannon, C.E., 1950. Programming a computer for playing chess. Philosophical Magazine 41. doi:10.1109/tsmc.1983.6313077.

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, 354–359. doi:10.1038/nature24270.

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, 1309 – 1331. URL: http://www.sciencedirect.com/science/article/pii/S0022000008000767, doi:https://doi.org/10.1016/j.jcss.2007.08.009.

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

Sutton, R.S., 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding, in: Advances in neural information processing systems, pp. 1038–1044. URL: https://www.ece.uvic.ca/~bctill/papers/learning/Sutton_1996.pdf.

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

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, arXiv:1511.06581.

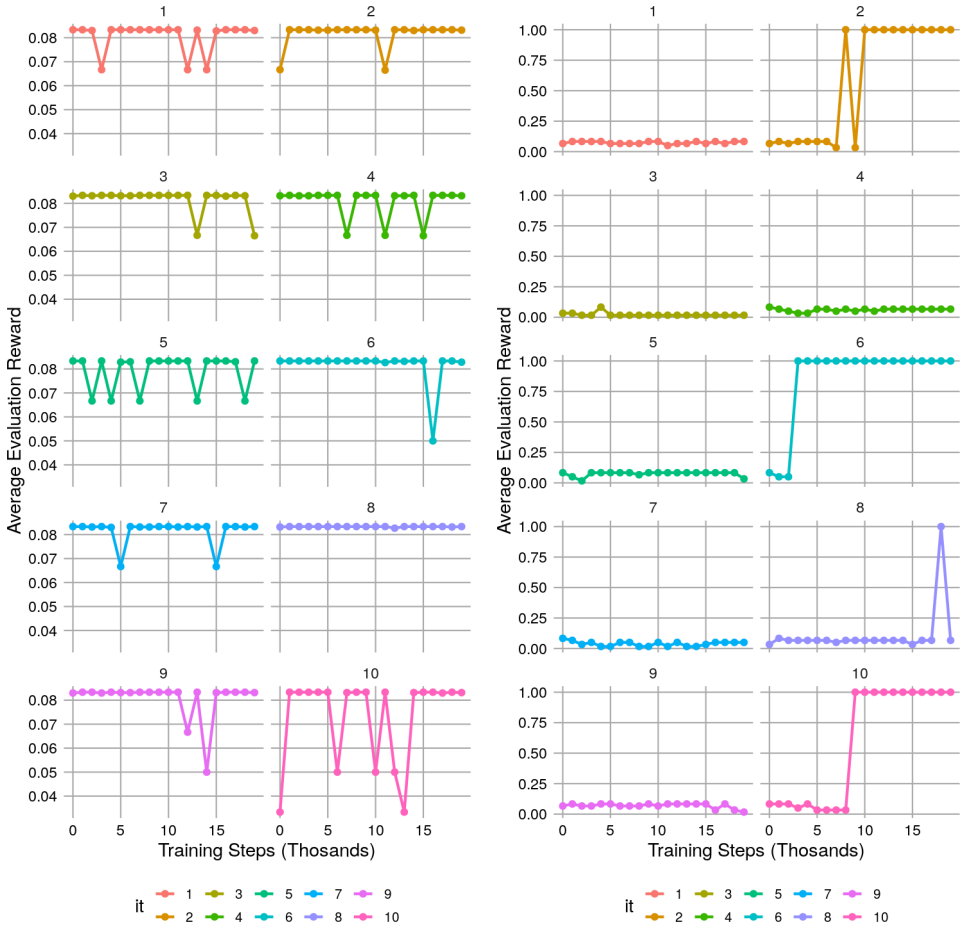
Watkins, C.J.C.H., Dayan, P., 1992. Q-learning. Machine Learning 8, 279–292. doi:10.1007/bf00992698.

Appendix A

Plots

A.1 Per seed plots of experiments

A.1.1 Corridor

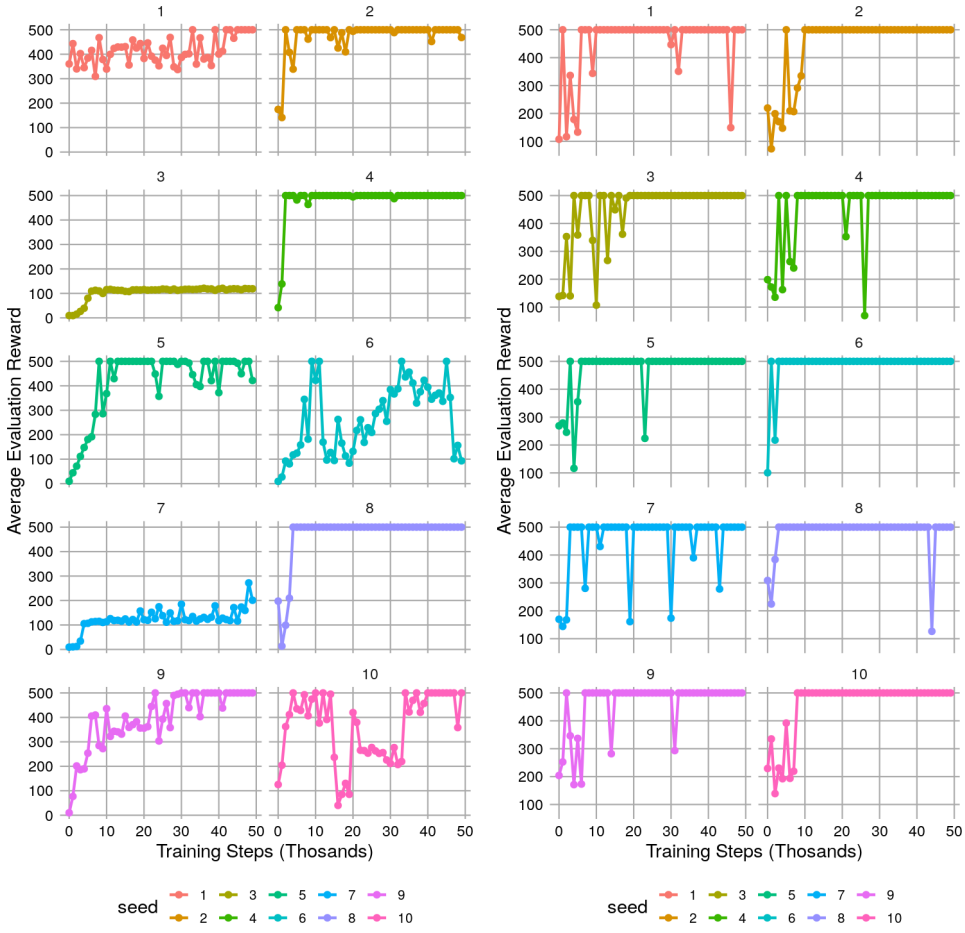


(a) DQN

(b) BNIG DQN

Figure A.1: Per Seed DQN and BNIG DQN Performance on Corridor

A.1.2 Cartpole

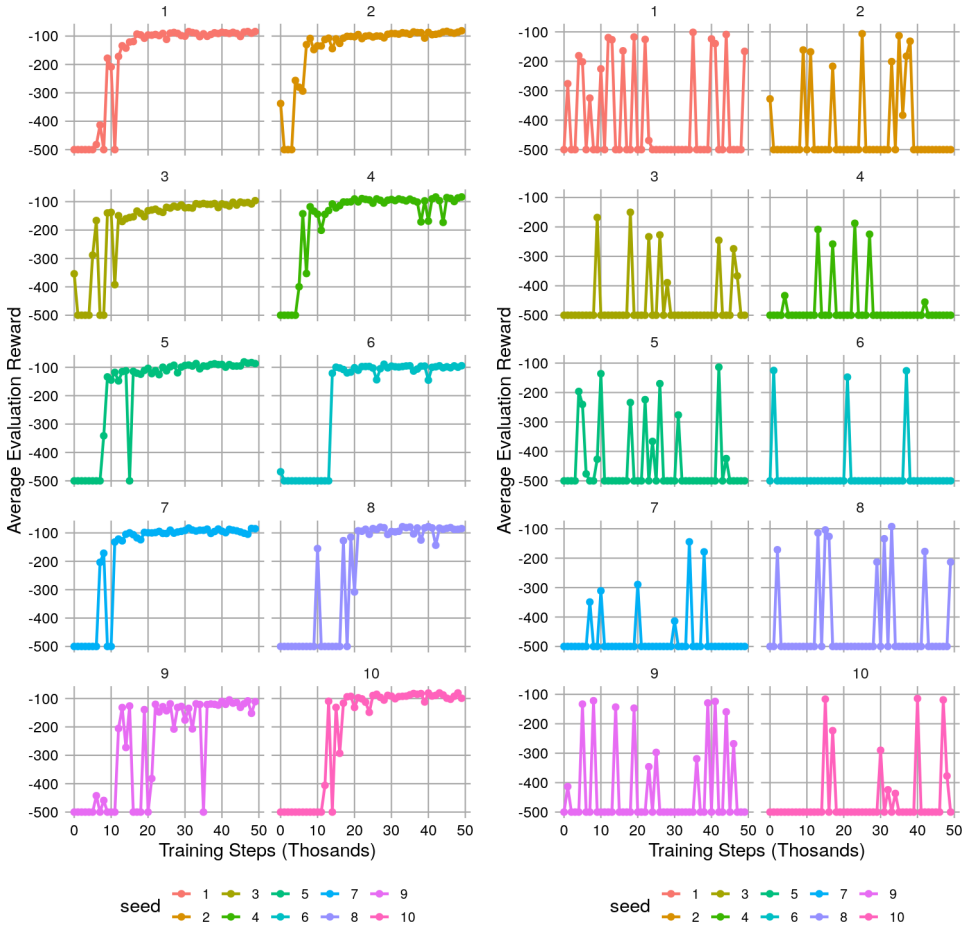


(a) DQN

(b) BNIG DQN

Figure A.2: Per Seed DQN and BNIG DQN Performance on Cartpole: Each color represents a new attempt. Note the unstable performance of DQN when it has found a good policy and that some attempts it never finds an optimal policy.

A.1.3 Acrobot



(a) DQN

(b) BNIG DQN

Figure A.3: Per Seed DQN and BNIG DQN Performance on Acrobot: Each color represents a new attempt. The BNIG DQN occaisonaly finds a good policy, but is not able keep a stable policy.