Table of Contents

List of Tables

List of Figures

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 successfully beat one of the worlds best Go players (Silver et al., 2017). In addition RL methods have learned to fly helicopters(Abbeel 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 Google(Evans 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. In contrast Silver et al. (2017) uses one 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.

Bellemare et al. (2013) 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 methods still require 20 million frames per game (Hessel et al., 2017) to achieve convincing results. 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. A game especially known for it's difficulty is Montezuma's Revenge(MR). Most RL 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, a concept that will be discussed in depth in the theory section. The underlying idea is that for the RL method to learn the correct combination of actions it must first perform these action in a similar manner. This is unlikely to happen if the agent acts randomly so an informed exploration of possible actions is needed.

The specialization project that preceded this thesis considered the paper O'Donoghue et al. (2017) which tries to combine statistical uncertainty with current RL methods to minimize the issues 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 score achieved in MR when published.

O'Donoghue et al. (2017) builds on a bayesian perspective RL but does not directly use bayesian statistics to solve the environments. Instead a neural network is used to learn a variance term based on a developed formula. This motivated an investigation into how a more direct approach using bayesian statistics can be used to drive efficient exploration. This thesis starts by developing a bayesian method on a simple linear environment before generalizing the method to more complex models and environments.

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.

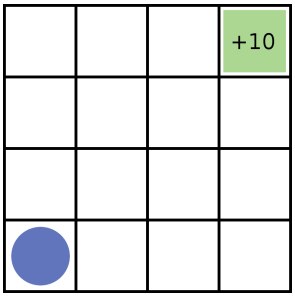


Figure 2.1: Initial State of Gridworld. The blue circle represents the player, while the green square represents the goal.

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.

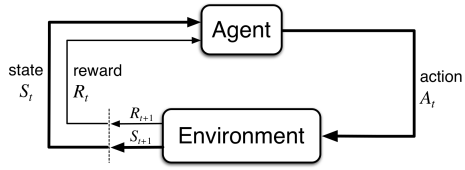


Figure 2.2: Visualization of the Agent and Environment. The figure is from (Sutton and Barto, 2018, p. 48)

Upon reaching the goal the game is over, making the goal state a *terminal 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.

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, an action A from a discrete action set $\mathbb{A}$ and a reward R from a real set $\mathbb{R}$. The transition probability is now 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 model the gridworld example as a 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 MDPs

2.2.1 Bellman Equation

Given a MDP the goal in RL is 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 when following policy π. 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.

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

TODO:Double check this source

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. This means one has access to all the transition probabilities for all possible state-action pairs. The following overview of the dynamic programming method is summarized from (Sutton and Barto, 2018, p. 74-84).

Policy Evaluation

The calculation of the value function of a given policy is known as policy evaluation. As an alternative to calculating the value function from the system of $|S|$ equations there is the iterative method

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

and terminating the iteration when

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

This converges to the value function given that $v_{\pi}(S)$ exists, which is true when $\gamma < 1$ or that an episode is guaranteed finite.

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 newly calculated $v_{\pi}(S_{t+1})$.

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

This process is called policy improvement. It can be proven that the new policy π' fulfills $v_{\pi'}(s) \geq v_{\pi}(s) \;\; \forall s \in \mathbb{S}$ meaning π' is as good or better than the policy π used for policy evaluation. (Sutton and Barto, 2018, p. 78-79)

Repeatedly running the policy evaluation and improvement steps above results in a monotonically improving policy and value function. For a finite MDP this will result in finding the policy that maximizes the discounted total reward.

Value Iteration

Policy iteration as described above requires that equation 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.

As in policy iteration it can be proved that value iteration 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 MDPs

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 is given access to an environment and tries to learn the optimal policy based on experiences from interacting with the environment.

2.3.1 Advantages of Learning From Experience

The experiences used to train the agent are often referred to as *samples* in RL literature as they can be viewed as samples from the underlying MDP. These samples are 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.

However this thesis also contains discussion about random samples and sampling from statistical models of value functions. To avoid confusion between these two concepts samples from the environment will be referred to as *experiences* throughout this thesis.

There are two main reasons why learning from experiences can be useful. In many cases one can generate experiences from 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 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 experiences 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 drawing experiences 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 MDPs 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).

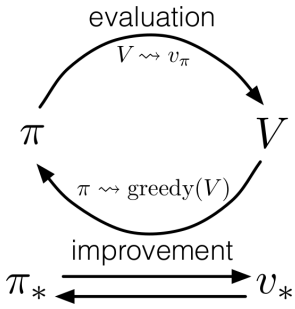


Figure 2.4: Visualization of GPI. Taken from p. 86 Sutton and Barto (2018)

Learning from experience

How to learn a value function from experiences is not obvious. Consider the case of trying to learn to play chess. One way to model this game as a 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. To do one can replace G_t by the one step return $G_{t:t+1} = R_{t+1} + \gamma V_{t+1}(S_{t+1})$ giving

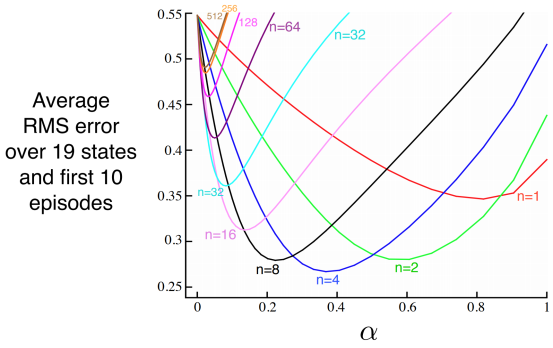
$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[R_{t+1} + \gamma V_\pi(S_{t+1}) - V(S_t)\big]. \tag{2.17}$$

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.

Since equation 2.17 considers the reward after one step it is called the 1-step TD method. However one doesn't have to chose between the monte carlo return or the one-step bootstrap. G_t can be replaced by any n-step return. For example a 3-step TD method

$$V_\pi(S_t) = V_\pi(S_t) + \alpha\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V_\pi(S_{t+3}) - V(S_t)\big]. \tag{2.18}$$

As $n \to \infty$ the n-step update becomes a Monte Carlo update. The choice of n can be seen as a bias-variance trade-off. The variance in a target in RL comes from a combination of stochastic transitions in the environment and different or stochastic policies. The total return based on the Monte Carlo target has variance from both these sources over all steps in an episode. A 1-step episode only has variance from the single transition and action choice but uses a biased bootstraped target value. The minimal error will then be somewhere between the two as can be seen in in figure 2.5.



.

Figure 2.5: RMS error on a toy example given different n-step updates. A 4-step update is seen to achieve the best performance. The plot and more information about the toy example can be found on page 145 in Sutton and Barto (2018)

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 experiences following the policy. This means that if the policy is changed, a new set of transition experiences 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.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\big] \tag{2.19}$$

Note that this equation is completely independent of the policy followed when generating the experience. This means in contrast to an on-policy method, Q-learning can be trained on transition experiences 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.

$$A_t = \arg\max_a Q_t(a) \tag{2.20}$$

The policy defined in equation 2.20 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

$$A_t = \begin{cases} \arg\max_a Q_t(a) & \text{with probability} \quad 1-\varepsilon \\ a \sim \text{Uniform}(\mathbb{A}) & \text{with probability} \quad \varepsilon. \end{cases} \tag{2.21}$$

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 2.6A 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 2.6B shows that

there are certain low price changes that have limited data. For these the same problem will occur.

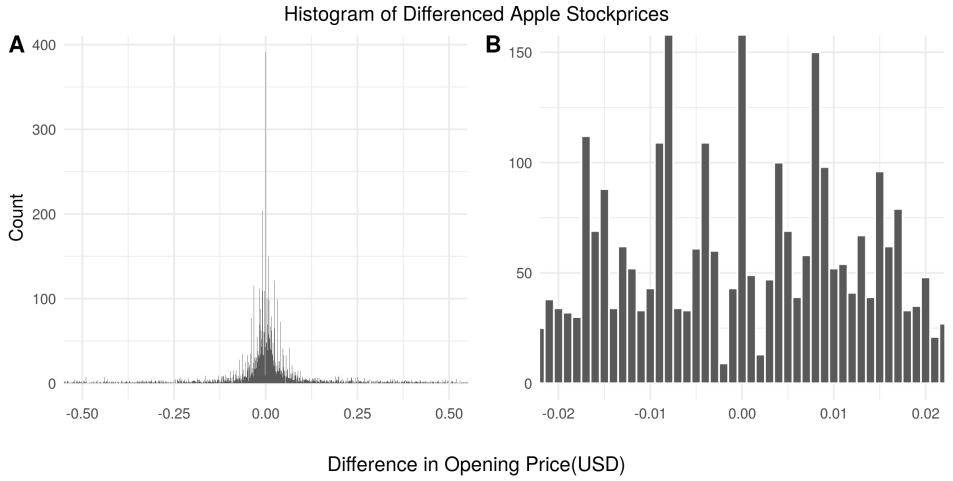


Figure 2.6: Differenced Apple Stock Opening Prices

Ideally the value estimate the method should be able to generalize to states with limited experiences 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 discussed earlier no longer hold. Some function approximation methods are still guaranteed to converge to a policy, albeit not necessarily the optimal policy while others are unstable and require special tweaks to reliably converge to a policy. Specifically off-policy methods with function approximation will sometimes diverge but in practice they tend to converge(Sutton and Barto, 2018, p .258).

2.4.2 Linear function approximation with 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

$$Q(X) = X\beta + \varepsilon \quad \text{where} \quad \varepsilon \sim N(0, \sigma^2) \tag{2.22}$$

with the response value defined as

$$Q(s, a) = r_t + \gamma \arg\max_{a'} Q(s', a'). \tag{2.23}$$

The ordinary least squares solution to the β coefficients can then be found using the normal equation (Fahrmeir et al., 2013, p .105-107) which in matrix form is

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

However this method is under the assumption that the target distribution is stationary and has to be trained on the entire dataset when given new data. This makes it ill-suited for reinforcement learning where one is constantly receiving new data from a non-stationary distribution.

Instead the common approach is to apply stochastic gradient descent over the parameters (**?**, p .201-203) With a monte carlo target this gives the equation

$$\beta = \beta + \alpha\big[G_t - Q(s, a|\beta)]\nabla_\beta Q(s, a|\beta).$$

The same equation for the temporal difference target gives

$$\beta = \beta + \alpha\big[r_t + \gamma \arg\max_{a'} Q(s', a'|\beta) - Q(s, a|\beta)]\nabla_\beta Q(s, a|\beta)$$

Note that in the temporal difference case the target is dependent on the coefficients and therefore does not fulfill all the criteria for stochastic gradient descent. This results in losing the guarantee that the method converges to a local optimum. However empirically in the case of linear models it tends to reliably converge.

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.

2.4.3 Nonlinear function approximation

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

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 with the MSE loss function it can be shown that

$$
\begin{aligned}
&\frac{\partial L}{\partial w_K} = \delta_K z_K \\
&\frac{\delta L}{\delta w_k} = \delta_k z_{k-1} \\
\text{where} \quad &\delta_K = \nabla L(w_K) \circ \sigma'(w_K z_{K-1}) \\
&\delta_k = \sigma'(w_k z_{k-1}) \sum_{i=1}^{S_k} \left(w_{k+1}\delta_{k+1}\right)
\end{aligned}
\tag{2.28}
$$

where S is the size of the layer, K denotes the final layer and σ' is the differentiated activation function. The above allows the gradients to be calculated based on the 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)

Q-learning with Neural Networks

To use neural networks as a nonlinear function approximator in RL the only step left is defining the loss in the RL setting. Consider the case of trying to minimize the mean square error of the action-value estimate

$$L_i(\theta_i) = \mathbb{E}_{s,a}\Big[\big(y_i - Q(s,a;\theta_i)\big)^2\Big] \tag{2.29}$$

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 2.19).

$$y_i = \mathbb{E}_{s'}\big[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a\big] \tag{2.30}$$

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

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a;s'}\left[\left(r + \gamma \max_{a'}\left(Q(s', a'; \theta_i)\right) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{2.31}$$

which can then replace $\nabla L(w_K)$ in equation 2.28.

Convolutional Neural Networks

Certain methods in this thesis will also use convolutional neural networks(CNN) in combination with fully connected NNs. This is not vital to the understanding of the work

done in this thesis so only a brief intuition of CNNs will be provided. The following is summarized from chapter 9 in Goodfellow et al. (2013).

Images are high dimensional so running fully connected NN that maps from every input to some final output requires a lot of parameters and is computationally heavy. In addition many of these parameters are wasted the same feature in different parts of the image should lead to similar results. In a fully connected NN this feature would have to be relearned for each location it can appear in an image. In addition if a feature ends up in a new position that the NN has not be trained on the method will fail.

CNNs are a different NN structure designed towards use cases such as images. A convolution operator is essentially a weighted sum of some input. In an image the weighted sum is over a local neighborhood based on the inputs position. CNNs learn a small set of weights and repeatedly takes the convolution between these a subset of the input across the entire input. Usually a CNN consists of many sets of these weights which learn different features. The result of this is that a CNN can recognize features regardless of their placement, known as translation invariant, and so does not need to relearn the same encodings. This means a CNN can recognize features with far less features and better generalizability than a regular NN.

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).

2.4.4 Deep Q Networks

The recent rise in interest in NN led to research 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, experiences are saved as the tuple $e_t = (s_t, a_t, r_t, s_{t+1})$ creating a data set of experiences $\mathcal{D} = e_1, ..., e_n$. 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 2.19. 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 2.32

$$L(y, \hat{y}) = \begin{cases} (y - \hat{y})^2 & \text{for } |y - \hat{y}| \leq 1 \\ 2|y - \hat{y}| - 1 & \text{otherwise.} \end{cases} \tag{2.32}$$

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.

2.4.5 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 one of the changes mentioned in Hessel et al. (2017), namely the double DQN. This change is not fundamental to this project so it will only be briefly discussed.

Hasselt (2010) showed that tabular Q-learning can have large positive bias in certain environments. This was due to an overestimation as a result of the max operator. To fix this they suggested creating two separate action-value functions and replacing the Q-learning update with

$$\begin{aligned} Q_2(s,a) &= Q_2(s,a) + \alpha\Big(r + \gamma Q_1\big(s', \arg\max_{a'}[Q_2(s',a')]\big) - Q_2(s,a)\Big) \\ Q_1(s,a) &= Q_1(s,a) + \alpha\Big(r + \gamma Q_2\big(s', \arg\max_{a'}[Q_1(s',a')]\big) - Q_1(s,a)\Big). \end{aligned} \tag{2.33}$$

During training one randomly picks one update to run every step. The underlying idea is that one decouples the targets action selection from its value estimation which results in a less biased estimator. In van Hasselt et al. (2015) this is extended to the DQN by simply using the target Q-network Q' as the second action-value function. The target Q-value was then calculated by

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

This is not equivalent to a complete decoupling of two action-value functions but empirically this reduced the bias and thus improved performance.

2.4.6 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 required 20 million frames per game to achieve its results. 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

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{ln(t)}{N_t(a)}} \right]. \tag{2.35}$$

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 2.7. 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.

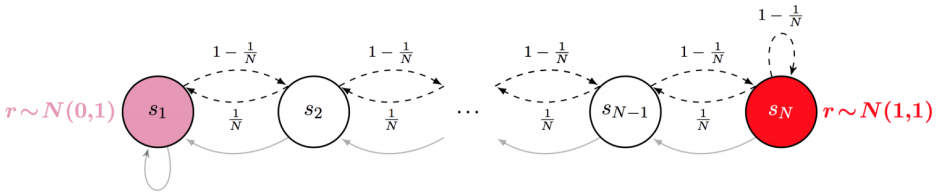


Figure 2.7: Chain environment. The figure is taken from Osband and Roy (2016)

The optimal policy is to always move right. However an agent following the policy in equation 2.35 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):

$$\max_a \mathbb{E}\Big[R_t + \gamma V(S_{t+1}) + \beta N(S,a)^{-\frac{1}{2}}\Big]. \tag{2.36}$$

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). In addition the method used both temporal difference and monte carlo updates, which the paper itself states lead to major improvements.

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.

$$Q^*(s_t,a_t) = \hat{R}^*(s,a) + \sum_{s_{t+1},a_{t+1}} P^*(s_{t+1}|s_t,a_t) \max_{a'} Q^*(s_{t+1},a_{t+1}) \tag{2.37}$$

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)

$$a_t = \arg\max_{a \in \mathcal{A}} Q^*(s_t,a) \tag{2.38}$$

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 2.8.

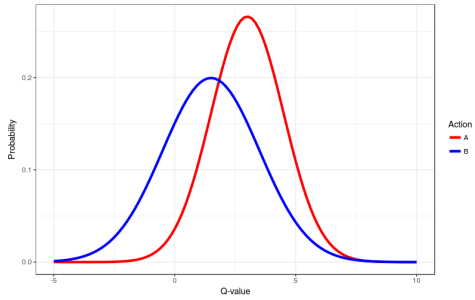


Figure 2.8: Posterior distribution of action-values: Despite action A having a higher expected value, the posteriors indicate that action B could potentially be the best action.

Figure 2.8 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.

2.6 Recent Research within Deep PSRL

Computationally tractable methods for bayesian inference in neural networks is especially challenging. Despite this there are multiple recent papers within that have been released the past year that are able to generalize PSRL to deep RL. Most of these use some form of approximation to simplify sampling from the posterior Q-value. A summary of three recent deep PSRL papers that are well cited and have helped form this thesis are provided.

2.6.1 Randomized Prior Functions for Deep Reinforcement Learning

Osband et al. (2018) finds an approximation to the posterior Q-value distribution through a mixture of statistical bootstrapping and ensemble methods. First the method assumes the posterior to be a gaussian distribution. It then presents the following, slightly summarized, lemma:

Let $f_\theta(x) = x^T\theta$, $\tilde{y}_i \sim N(y_i, \sigma^2)$ and $\tilde{\theta} \sim N(\bar{\theta}, \lambda I)$. Then the following optimization problem generates a sample $\theta|\mathcal{D}$

$$\tilde{\theta} + \arg\min_{\theta} \sum_{i=1}^{n} \|\tilde{y}_i - (f_{\tilde{\theta}} + f_\theta)(x_i)\|_2^2 + \frac{\sigma^2}{\lambda}\|\theta\|_2^2. \tag{2.39}$$

Breaking down this statement the method starts by sampling coefficients $\tilde{\theta}$ from the prior to create a prior function $f_{\tilde{\theta}}$. Then it fits an additive model $f_\theta(x)$ with a regularization term $\frac{\sigma^2}{\lambda}\|\theta\|_2^2$ to a noisy version of the dataset $\tilde{y} \sim N(y, \sigma^2)$. The resulting outputs from $(f_{\tilde{\theta}} - f_\theta)(x_i)$ will be a single sample from the posterior.

This means that in order to model the posterior based on n samples one needs n models. Though it can be computationally heavy to create many models it is also noted that these models can be trained in parallel, meaning this method is scalable given the right computational resources.

To generalize to neural networks the method approximates further by replacing the prior model with a fixed randomly initialized neural network p multiplied by a hyperparameter β that regulates the magnitude of the prior. Instead of applying noise to the data a new dataset is created by statistically bootstraping the original dataset. A separate DQN is then trained on the bootstraped data and the addition of the fixed prior network. The resulting Q-value is then

$$Q(s,a) = f_\theta(s,a) + \beta p(s,a) \tag{2.40}$$

The paper then shows that on hard exploration toy examples such as a variant on the chain environment(figure 2.7) the method far outperforms an ε-greedy exploration policy. In addition the method manages to escape the first room in Montezuma's Revenge, receiving an average score of approximately 2500. However the paper also notes that this performance is sensitive to adjustments to the prior scale β.

2.6.2 The Uncertainty Bellman Equation and Exploration

The specialization project that preceded this work was an investigation into O'Donoghue et al. (2017). They introduce an upper-bound on the variance of the Q-value based on the Bellman equation

$$\text{var } Q(s,a) \leq \nu(s,a) + \sum_{s',a'} \mathbb{E}\big[P(s'|s,a)\big] \text{ var } Q(s',a') \tag{2.41}$$

where ν is given by

$$\nu(s,a) = \text{var } \mu_r(s,a) + Q^2_{max} \sum_{s'} \text{var } \frac{P(s'|s,a)}{\mathbb{E}\big[P(s'|s,a)\big]}. \tag{2.42}$$

ν is called the local uncertainty since it only depends on the current state-action pair and not the variance of future action-state pairs.

By assuming the variance is the upper-bound value O'Donoghue et al. (2017) one gets a bellman style equation

$$u(s,a) = \nu(s,a) + \sum_{s',a'} \mathbb{E}\big[P(s'|s,a)\big] \text{ var } Q(s',a') \tag{2.43}$$

which, when based on experiences rather than model dynamics gives

$$u(s,a) = \nu(s,a) + \gamma^2 u(s',a'). \tag{2.44}$$

Given a method to calculate ν this allows one to learn the $u(s,a)$ values using the same methods used to learn $Q(s,a)$.

It is then assumed that there are enough experiences to use the central bayesian limit theorem. This implies that the posterior distribution of the Q^*-value is approximately distributed by a normal distribution(Berger, 1985, p. 224). As such, given a $\mathbb{E}[Q]$ value from any regular Q-learning method and a variance calculated from the UBE one can sample a posterior Q-value from the distribution $N\Big(\mathbb{E}\big[Q(s,a)\big], u(s,a)\Big)$.

The issue that remains is how to calculate the local uncertainty ν. O'Donoghue et al. (2017) argues that ν can be approximated by action-value models variance. Given a target y_i with i.i.d. variance it can be shown that for a linear regression with weights w_a, a new input $\phi(s)$ and design matrix $\mathbf{\Phi}$ the model variance is $\phi(s)^T \mathbf{\Phi}^T \mathbf{\Phi} \phi(s)$. The local uncertainty is the estimated by

$$\nu(s, a) = \beta^2 \phi(s)^T \mathbf{\Phi}^T \mathbf{\Phi} \phi(s) \tag{2.45}$$

where β is a hyperparameter that controls the scale of the local variance and represents the actual variance of the target.

By ignoring the uncertainty in the neural network encoding the same logic can be applied to the final layer in a DQN. The final deep RL implementation creates a DQN with two heads. Head one is trained as a regular DQN and outputs Q-values. Head 2 is trained using the UBE with local uncertainties calculated from the final linear layer from head 1.

Actions are then chosen by sampling the normal distribution with the mean estimate from head 1 and variance estimate from head 2. With this method O'Donoghue et al. (2017) managed state of the art results on Montezuma's Revenge when published. In addition it allowed, under heavy assumptions that are broken, sampling from an approximated posterior using a DQN model that only took 10% more computation to run, rather than the n times heavier computation for ensemble methods.

2.6.3 Efficient Exploration through Bayesian Deep Q-Networks

Azizzadenesheli et al. (2019) will be discussed in detail in chapter 3 so only a brief overview of the paper is provided here. The paper consists of two major results of which the first is relevant to this thesis. This part introduces a deep RL PSRL method they denote the Bayesian Deep Q-Network (BDQN).

The BDQN assumes a gaussian prior and posterior distribution. It then replaces the final linear layer in the DQN with a bayesian linear regression. This allows a DQN that can sample Q-values from a posterior, albeit one that ignores uncertainty in the network encoding. Note that this method does not explicitly propagate uncertainty and chapter 3 will argue that the method has no uncertainty propagation. In addition, as in O'Donoghue et al. (2017) and Osband et al. (2018) the variance scale is set as a hyperparameter.

Since this thesis will be heavily discussing the BDQN it is also worth noting that Azizzadenesheli et al. (2019) has not been accepted to any conference at the time of writing. Despite this the paper is well known in the RL field with over 30 citations and is often brought up in new PSRL papers. An open review for ICLR 2019 shows that the concerns about this paper revolve mainly around the second part of the paper, a theoretical proof of performance, not the BDQN(OpenReview, 2018).

2.6.4 Dropout

On a short note, if the reader is well-versed in neural networks one might wonder why none of these paper use dropout to estimate the posterior. In this case it is recommended to view the discussion in Osband et al. (2018) about the shortcomings of dropout in RL. In addition Azizzadenesheli et al. (2019) empirically shows that dropout does not perform well in the RL setting.

Chapter 3

Conjugate Bayesian Linear Q learning

The goal of this thesis is to try to develop a method that implicitly propagates uncertainty without the need of as many assumptions and extra components as the UBE. First this chapter will show that the bayesian model in Azizzadenesheli et al. (2019) does not propagate variance. It will then go on to suggest a new model that does propagate variance, discuss the properties of this propagation and test the models performance in a linear setting.

3.1 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.

3.2 Conjugate Bayesian Linear Q learning

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.

3.2.1 Gaussian Prior with Known noise

To start consider the model used in Azizzadenesheli et al. (2019). As in frequentist 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

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

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

$$
\begin{aligned}
\Lambda_{a_n} &= \sigma^{-2} X^T X + \Lambda_{a_0} \\
\mu_{a_n} &= \sigma^{-2} \Lambda_{a_n}^{-1} (\Lambda_{a_0} \mu_{a_0} + X^T Q_a).
\end{aligned}
\tag{3.1}
$$

where $_0$ denotes the prior and $_n$ denotes the posterior parameters. The development of these updates can be found in A.1. This model will be referred 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 2.22 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.

3.2.2 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 incorporate 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

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

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

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

where β is sampled from its posterior and ε from the gaussian noise distribution. However the variance of β, as seen in equation 3.1, 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.

3.2.3 Normal Prior with Unknown noise

Including σ_ε as an unknown parameter results 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

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

The development of these formulas can be found in A.2. This model will be referred to as the Bayesian Normal Inverse Gamma model(BNIG).

3.2.4 Testing Variance Propagation

To show that the BNIG model with sample targets is the only method that propagates uncertainty consider a simple example from Osband et al. (2018). The environment consists of a MPD with two states(figure 3.1). The initial state allows only one action that deterministically 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 $\gamma = 1$.

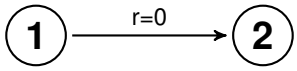
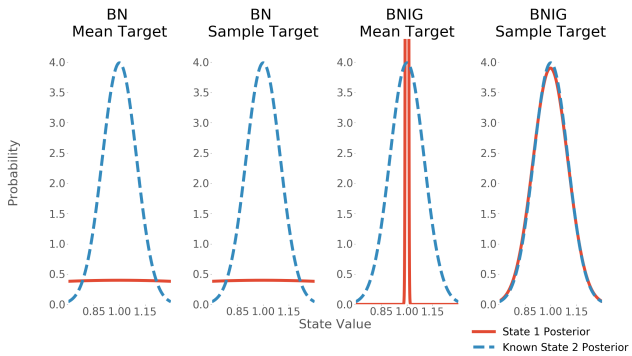


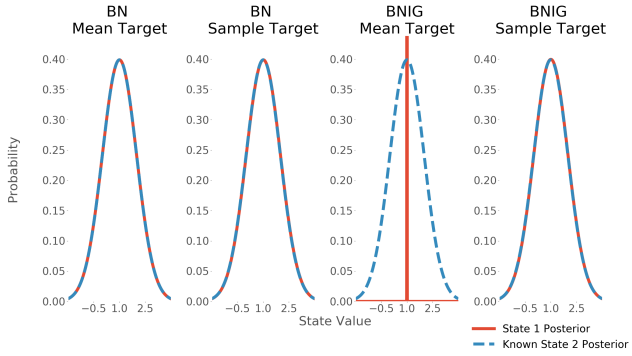
Figure 3.1: 2 State Toy Environment

Both models were tested with both MAP and sample targets. The priors for the BN models were set to $\beta \sim N(0, 10^3)$ and $\sigma^2 = 1$. The priors for the BNIG models were set to

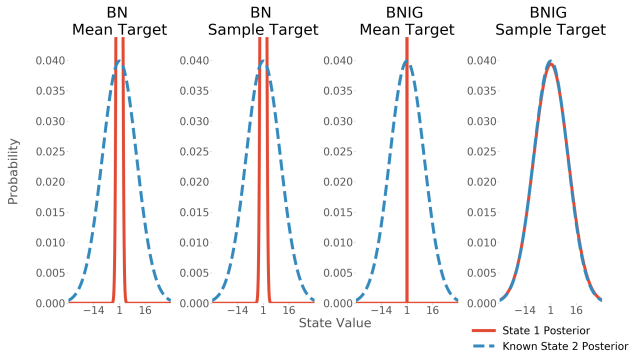
$\beta \sim N(0, 10^3)$ and $\sigma^2 \sim InvGamma(1, 0.01)$. Three MDPs were set up with a known posterior of $N(1, 0.1), N(1, 1)$ and $N(1, 10)$ respectively. Implementation details can be found in **??**.



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



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



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

Figure 3.2: 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.

The results summarized in figure 3.2 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 underestimates the variance. This would be the 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.

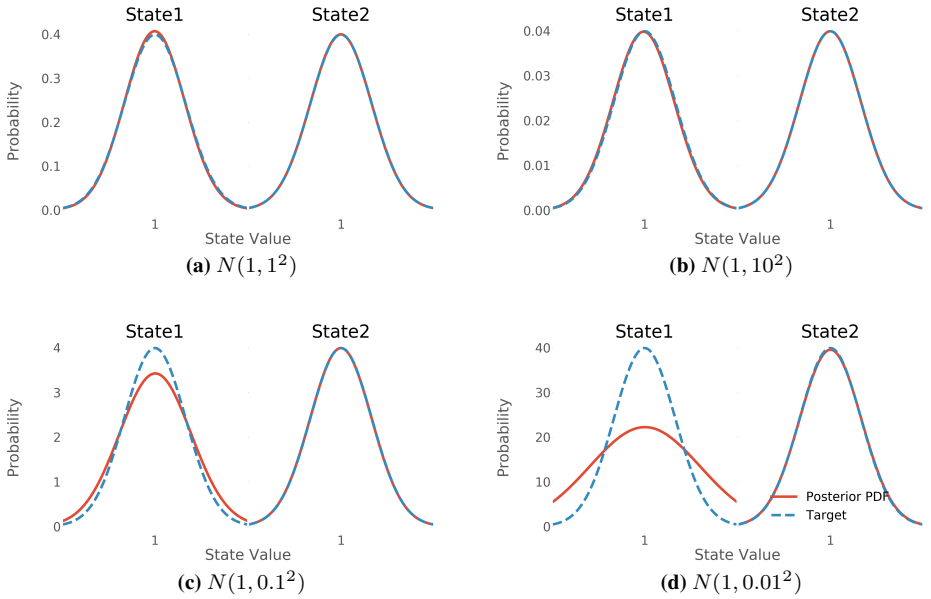
3.3 Variance Propagation

3.3.1 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(figure 3.3). 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.

Figure 3.3: 3 State Toy Environment

The same priors as earlier are used on 4 different target posteriors. The results are summarised in figure 3.4. Implementation details are provided in **??**.



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

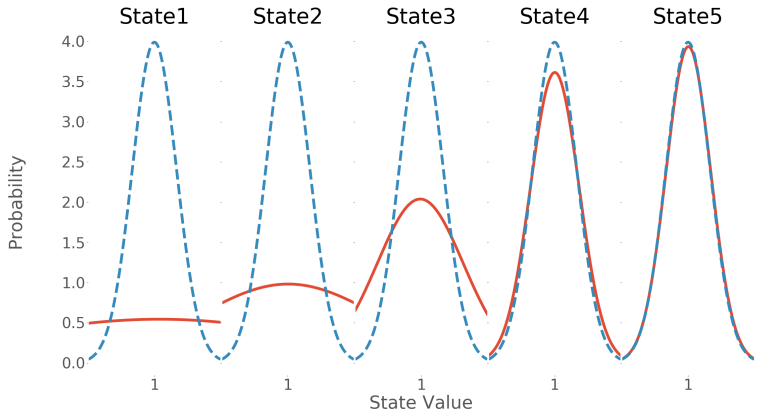
(c) $N(1, 0.1^2)$ **(d)** $N(1, 0.01^2)$

Figure 3.4: 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.

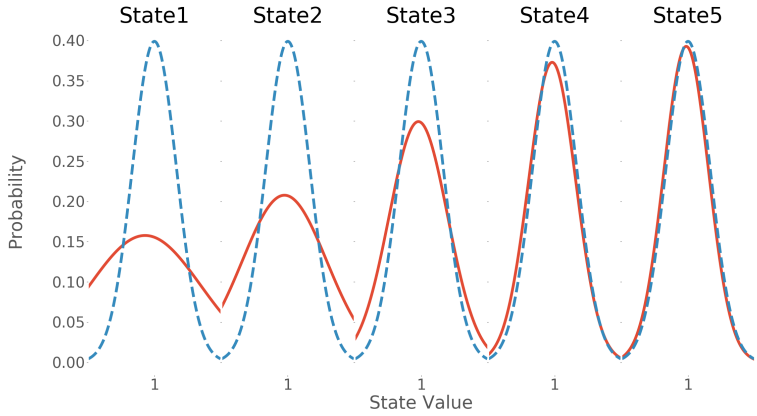
Figure 3.4 shows that for larger variance targets the propagation correctly updates the state 1 variance. However the low variance targets results in an overestimation over the variance in state 1. Running this experiment for more iterations does lead to a better approximation implying that the problem 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.

By extending the toy example to even more states as in figure 3.5 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 $N(1, 0.1^2)$



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

Figure 3.5: 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 referred 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.

3.3.2 Speed of propagation

In order to improve the speed of propagation insight is required into what causes the issue. In this section two possible causes and methods for counteracting them are discussed.

However these are not necessarily the only two causes of slow propagation.

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. Similarly extending the 1-step update to an n-step update will increase the speed of variance propagation 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, p .229)). 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 impact 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.

3.3.3 Forgetting Old Data

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.

This is a problem faced in one of the earliest bayesian RL papers, Dearden et al. (1998). They deal with the problem by using exponential forgetting, a term they don't explain or give a reference too. However a similar issue is faced in Ting et al. (2007) where a method that reduces the impact of previous data by exponentially decaying old data is used. It seems likely that this is the exponential forgetting mentioned in Dearden et al. (1998).

One exponentially decay the data by keeping all the previous datapoints in memory and discounting all datapoints every timestep. However a more space and computationally efficient method used in Ting et al. (2007) is to only keep track of the terms used to update the posterior. In equation 3.2 the terms X^TX, X^Ty, y^Ty and n are used. All of these are proportional to the number of predictors rather than the number of datapoints. The terms can easily be exponentially decayed by using the following updates

$$(X^T X)^{(k+1)} = \alpha X^T X^{(k)} + x^T x \tag{3.2}$$

$$(X^T y)^{(k+1)} = \alpha X^T y^{(k)} + x^T y \tag{3.3}$$

$$(y^T y)^{(k+1)} = \alpha y^T y^{(k)} + y^T y \tag{3.4}$$

$$n^{(k+1)} = \alpha n^{(k)} + b \tag{3.5}$$

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.

3.4 Sampling Frequency

A final consideration to be made is how often to sample the coefficients for the Q-value regression during decision making. From a statistical background it might seem most natural to sample new coefficients every step. This is what is done in O'Donoghue et al. (2017). Another possibility is presented in Osband et al. (2018) which samples new coefficients every episode. This is the equivalent to sampling one action-value function per episode and Osband et al. (2018) argues that this increases the probability for optimistic action-value functions leading to increased exploration. Azizzadenesheli et al. (2019) takes this even further and samples new coefficients at large intervals spanning thousands of steps. However this is done to stabilize their neural network and mention that ideally the sampling interval should match the average episode length.

Following the 3 mentioned papers the choice stands between sampling every step versus every episode. None of the papers provide mathematical proofs behind their sampling frequencies so both will be tested in the next section.

3.5 Performance on Linear RL Problem

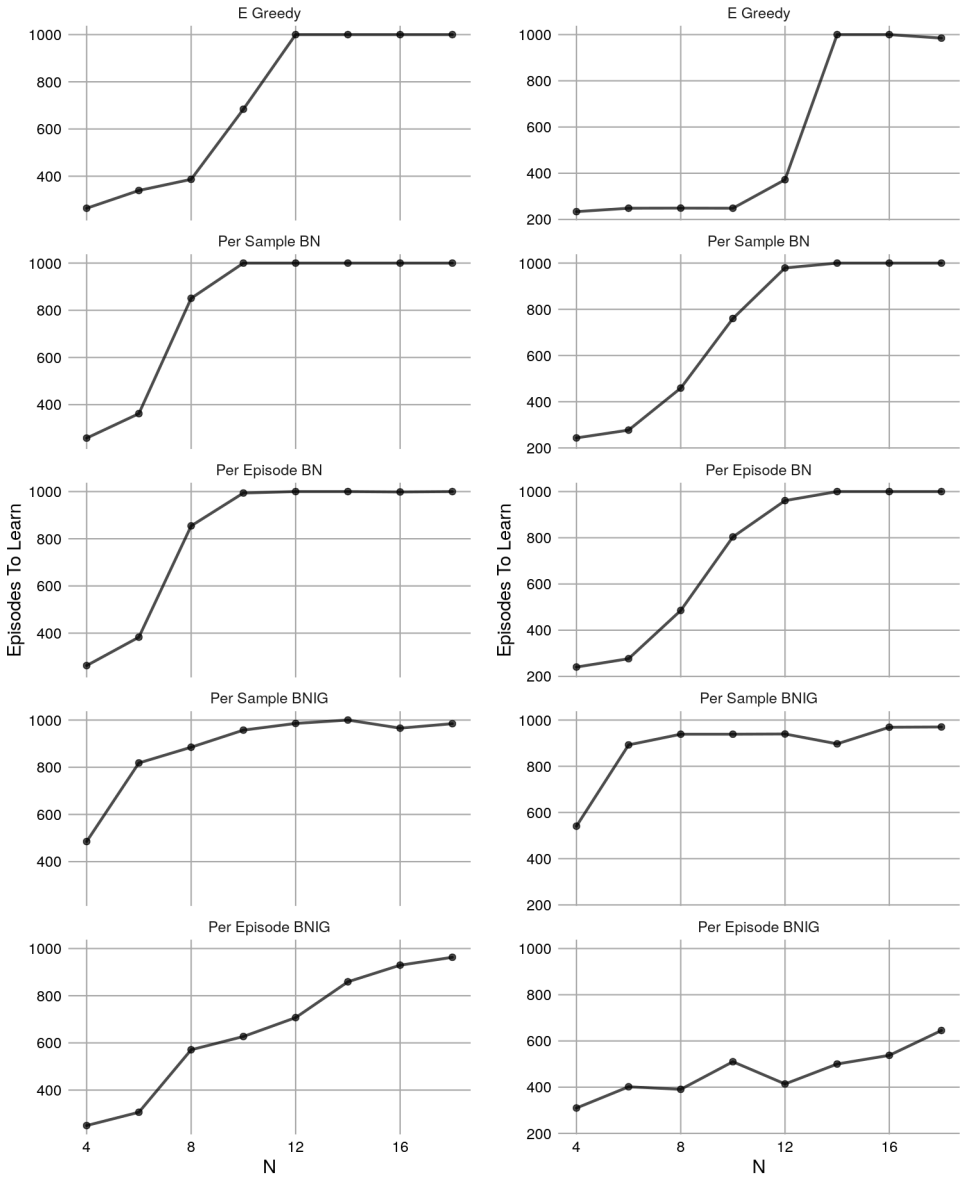
Up to now the model has only been tested on simple problems with known posteriors and no active decision making. To test how well this generalizes to a more realistic RL environment consider a variant on the corridor environment introduced in the theory section in figure 2.7.

The environment is 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 receives no reward.

All the methods discussed so far were tested on this environment with different chain lengths. This includes the BN and BNIG model with both per step and episode sampling. In addition a linear Q-learning method following an ε-greedy exploration scheme was

tested. The hyperparameters for each model were found by manually testing 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 appendix(B.3).

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



(a) 1 step update

(b) 3 step update

Figure 3.6: Linear Model Performance on Corridor Environment: (**a**) shows that the only method that outperforms the ε-greedy policy is the per episode 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 3.6 is that the ε-greedy method outperforms all methods apart from the BNIG model with per episode sampling. However there are a few interesting differences in the results between the bayesian models and their exploration methods.

The per step BNIG model performs worse than the per step 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.

Using per episode sampling 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 per episode BNIG the improvement is increasing the longest learnable corridor by around 2-4 states. In the per episode BNIG's case the improvement is an increase of at least 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 improvement than in other methods.

Chapter 4

Neural Linear Bayesian Regression Model

A major drawback of the methods discussed in chapter 3 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 3 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.

4.1 Combining Bayesian Q-learning with Neural Networks

4.1.1 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 3 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.

4.1.2 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 and uses the MAP target for both without considering the possibility of sampling the target. 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 to 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 the network does not consider the variance of the target it is less computationally expensive and more stabalizing to use the MAP. The pseudocode

for this update is

Algorithm 1: BDQN Network 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 2.32)
end

The BN model is trained using the posterior updates described in chapter 3 in equation 3.1. Since the BN model does keep track of variance one could consider using a sample targert rather than the MAP target in this case. However it has already been shown that in the linear case using a sampled target over a MAP target has no effect on the final estimate as it does not propagate uncerainty.

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.

4.1.3 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. The pseudocode for this update is seen in algorithm 2. Note that the * notation denotes values from the target parameters and ε_a is a sample from $N(0, \sigma_a^2)$.

Algorithm 2: BNIG training

if t ***mod*** *batch size* $== 0$ **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 3.2

end

With this approach the only required change to transform the BDQN setup to a BNIG method is to swap out the BN 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 summarized in the pseudocode below.

Algorithm 3: BNIG DQN

Initialize variables according to algorithm 4
for *episode = 1, M* **do**
- Initialize environment
- $s_1 =$ initial environment state
- Sample β_a, σ_a for $a \in \mathcal{A}$
- **for** *t=1, T* **do**
  - $a_t = \arg\max_a \left(\phi(s_t)\beta_a + \varepsilon_a \right)$
  - 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 $\mathcal{D}$
  - Train online network using algorithm **??**
  - Train online model using algorithm 2
  - **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 T_{sync} is a hyperparameter that defines how often to sync the online and target network and model. Once again ε_a represents a sample from $N(0, \sigma_a^2)$. The initialization process in algorithm 4 sets up the models, networks and experience buffer required.

Algorithm 4: 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 $\mu_a^* = \mu_a^*$, $\Sigma_a^* = \Sigma_a^*$, $\alpha_a^* = \alpha_a^*$, $\beta_a^* = \beta_a^*$.

4.2 BNIG DQN Results

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

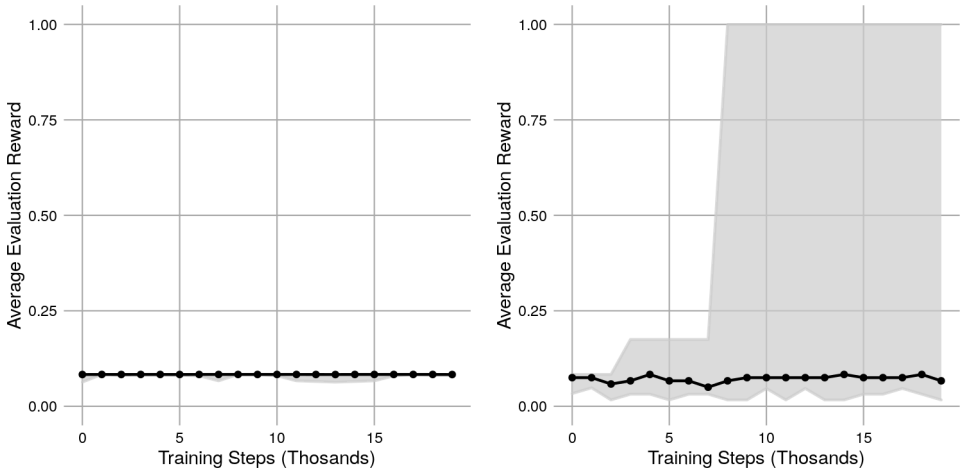
To ensure a correct implementation of the DQN and a fair comparison to the BNIG DQN the python package Dopamine(Castro et al., 2018) was used. This package contains an implementation of a DQN that matches the baselines of the Mnih et al. (2015). The BNIG DQN was implemented ontop of this DQN implementation.

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.

4.2.1 Corridor



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

Figure 4.1: 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.

4.2.2 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., 2016).

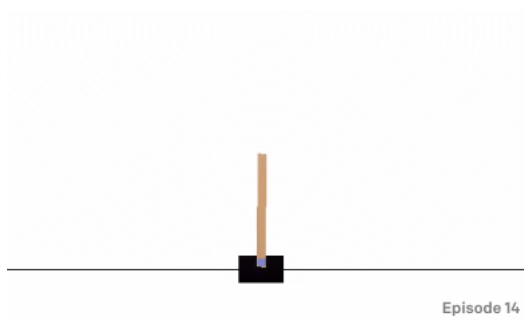
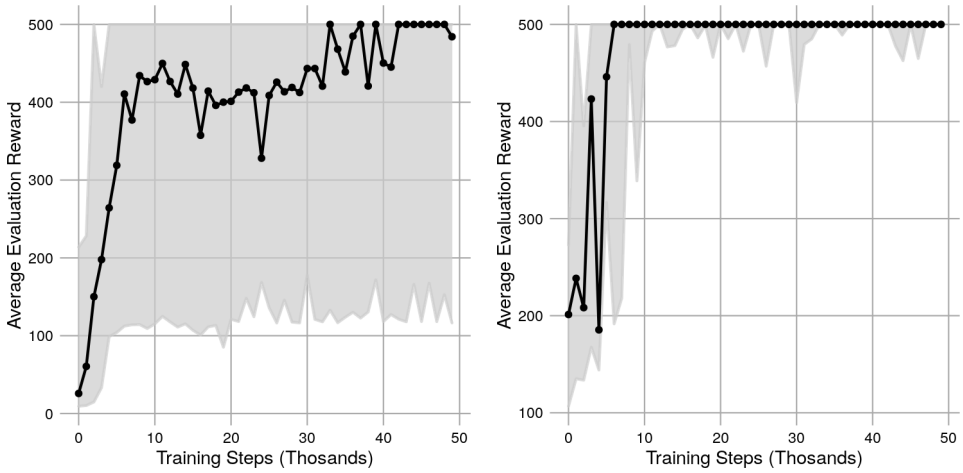


Figure 4.2: Cartpole Environment The pole is in brown and the cart in black. (OpenAI)

The task, seen in figure 4.2, 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., 2016). 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 4.3.



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

Figure 4.3: 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 4.3 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 C.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.

4.2.3 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, seen in figure 4.4, represents a vertically hanging 2D, 2 joint robot arm. The first joint's position is fixed to the origin while the second join has an actuator. This actuator can apply torque to the second joint. The arm starts hanging downwards under the origin.

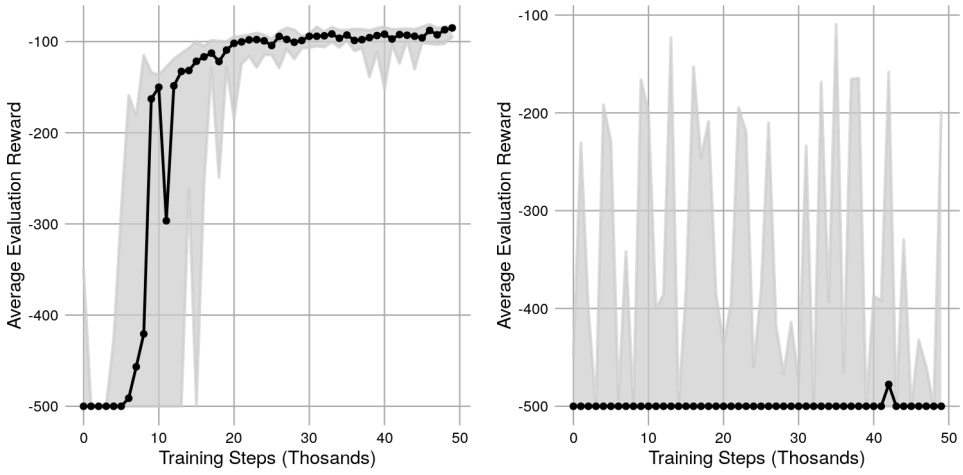
The goal is to balance the force of gravity, the actuators output and the momentum of the arm to swing the end of the arm over a line above the origin. Acheiving this terminates the episode. A reward of -1 is given each timestep. The state input is

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

where θ_1 is the angle of the first joint and θ_2 is the angle of the second joint relative to the first. The agent can apply -1, 0 and +1 torque to the second joint.

Figure 4.4: Acrobot Environment: A still frame from the OpenAI acrobot implementation. The joints are marked in green, the arms in blue and the line represents the threshold. (OpenAI)

The environment implementation from OpenAI Gym was used (Brockman et al., 2016). Each method was run for 100,000 steps with 10 different seeds and every 2000 steps an evaluation phase is run over 2000 steps. The environment and performance is seen in figure 4.6



(a) DQN

(b) BNIG DQN

Figure 4.5: 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 4.8 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 C.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. It 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. UPDATE: Works with a higher alpha but this feels like just pushing the method towards a DQN. With a large alpha relative beta the noise term ”disappears”.

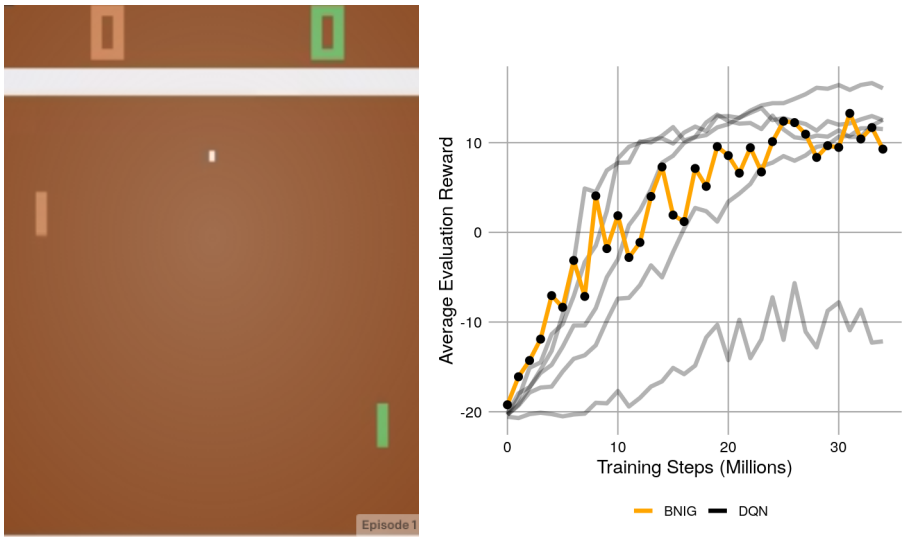
4.2.4 Atari

A standard benchmark in the deep RL field is a large set of atari 2600 games called the Arcade Learning Environment(ALE) introduced in **?**. Papers presenting new methods in this field generally tested for 200 million frames on 40-50 of these games, such as in Mnih et al. (2015), Mnih et al. (2016) and O'Donoghue et al. (2017). All environments must be learned based purely on the image input from the game. Due to the difficulty of the games and the large networks required the learning process is heavy to run, often taking close to a week on a GPU per game. Due to limited compute the BNIG DQN was tested once a subset of these games. The results were compared to 5 seeds of DQN runs that

can be found in the Dopamine package. Both methods use all the same hyperparameters with the exception of the update horizon, which was set to 5 for the BNIG DQN. This is known to increase the performance of the DQN, so the results are not a completely fair comparison.

Pong

Pong is one of the simpler environment in the ALE with superhuman performance acheived already in 2013(Mnih et al., 2013). The game consits of two paddles and a ball. The aim of the game its to hit the ball past the other players paddle which results in a +1 reward. If the other player hits the ball past the agents paddle it receives a -1 reward.



(a) Environment **(b)** Performance

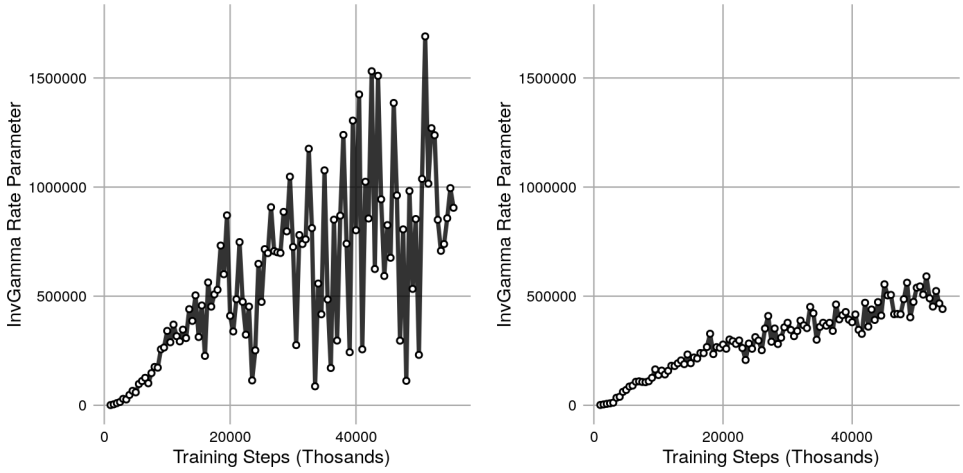
Figure 4.6: Pong environment results: **(a)** shows an input state of the pong environment(OpenAI). The green panel is the agent and the ball is in white. **(b)** compares the BDQN performance in orange to the 5 DQN runs provided in Dopamine.

BNIG started off well, with better evaluation performance than all DQN seeds for the first 5 iterations. From there on out the improvements decrease and the evaluation becomes more unstable. After 35 million steps BNIG has similar to slightly worse results than the DQN. Since this test only compares 1 seed to 5 seeds there is not enough data to confidently evalute the performance but the results indicate the performance is similar to the DQN albeit more unstable. Taking into consideration that BNIG is using a 5-step update, which should improve performance, the underlying BNIG method seems to actually worse than DQN on this game.

4.2.5 Tuning BNIG

Due to the poor performance on acrobot and pong an attempt was made at tuning the hyperparameters of the BNIG model to see if this could get better results. A deeper look at the development of the model parameters for the acrobot environment showed that the rate parameters b_a dramatically increase leading to a too large variance. Multiple attempts were made at controlling this increase but the only successfull result was increasing the scale parameter a_a.

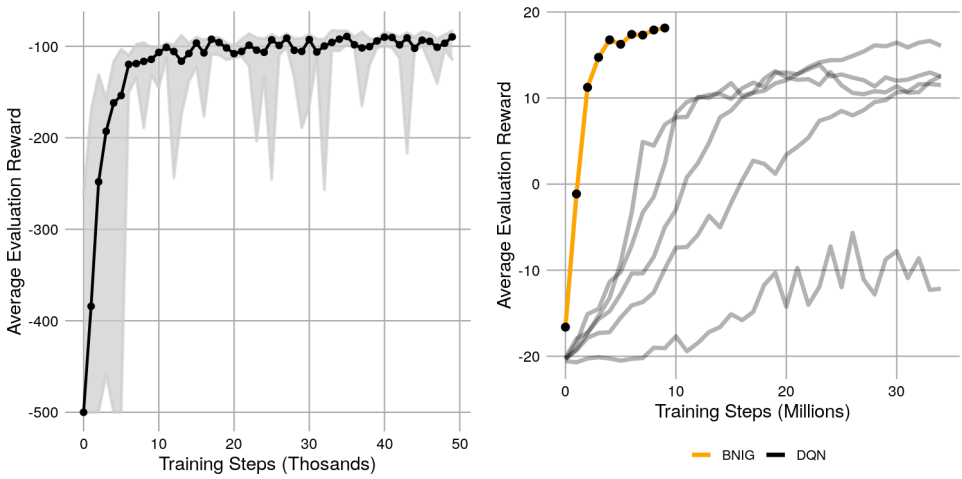
TODO:replace scale plot, based on wrong hyperparameters, but the gist is correct



(a) Prior Scale of 1

(b) Prior Scale of 10,000

Figure 4.7: Development of rate parameter given different scale priors.



(a) Acrobot BNIG

(b) Pong BNIG

Figure 4.8: High scale prior results on acrobot and pong

Chapter 5

Discussion

5.1 BNIG model limitations

5.1.1 State Independent Variance

One constraint of the BNIG model is that it assumes a constant variance over all states given an action. This does not necessarily deteriorate performance as much as one might expect. Azizzadenesheli et al. (2019) achieves performance better than DQN in multiple ALE games with the same restriction. However it seems that this might be a problem on environments that require more complex exploration. For example Montezuma's revenge is not included in the attempts in Azizzadenesheli et al. (2019) and attempts at using the BNIG model converges to a zero reward policy.

It is likely that an environment such as Montezuma's Revenge requires the variance to be adjusted per state. In these environments the same action might lead to states with largely different varainces based on the current state. Consider a left action on the edge of one room in Montezuma's Revenge. This leads to a new room with a large set of possibilities and thus should have a large variance. In another case a left action might cause the character to fall to its death. The agent should recognize what is happening and have a low variance estimate that there won't be anymore reward. With the current BNIG model the variance estimate for the left action will have to be between the two resulting in too much exploration in states the model should be certain about and more importantly too little exploration in unexplored states.

In a game like cartpole this is still the case but to much smaller extent. The state of the cartpole has little effect on the variance of the next state. Once a policy is learned the pole is relatively certainly going to stay balanced over future states meaning low variance. If the pole is starting to fall the pole will most likely completly fall leading to termination, once again with low variance. In these types of environments one should expect BNIG to perform well.

5.1.2 Variance Stability

Another shortcoming of the BNIG DQN method developed in this thesis is the instability of the variance estimate. Though an increase in the prior scale parameter improves performance this lowers variance based on a hyperparameter rather than the data. It greatly constricts the exploration and must be tuned on an enviornment to enviornment basis. Further developments should focus on how to stabalize this variance to allow for weaker priors. There are multiple issues that could decrease the stability of the variance.

Maximization Bias

One source of instability could be maximization bias(Sutton and Barto, 2018, p. 134) occuring on the variance estimate. Early in the learning process the maximum operator on the target leads to picking actions that lead to high variance states as these provide the highest values. This leads to higher variance targets which starts a feedback loop that can lead to the large jumps in the rate parameter of the inverse gamma distribution as seen in figure 4.7.

A solution to the maximization bias in regular Q-learning is double Q-learning(Sutton and Barto, 2018, p. 134). This consists of creating two completely decoupled models and for each update use one for target action selection and one for action-value calculation. Which is used for what is randomly chosen each update. This doubles the memory complexity of the model but keeps the same computational complexity. The double DQN uses an approximation to this by using the target network for target action selection and the online network for action-value calculation. This is also what is done with the BNIG target model in this thesis. However a complete decoupling by creating two BNIG models per action might help reduce the variance instability.

State Independent Variance

It has already been discussed how the state independent variance might be constraining the environments BNIG is usefull on. However it is also possible that this is leading to variance instability on all environments.

Consider regular Q-learning. The magnitude of the Q-values is controlled by the terminal states where the target is bound by the maximum reward from the environment. As Q-learning learns, this bounded value is propagated backwards towards the initial state, leading to Q-values that are realistic relative to the return from the environment.

In the variance case one should expect the same behaviour. The propagation tests in chapter 3 showed that given a known posterior state all other states leading up to this will converge towards the same variance. However this test is in a tabular setting where each state has its own variance parameter. In the linear setting a state that should have low variance, such as a penultimate state, will have to have a similar variance to all other states. As such there is no low variance state that slowly bounds the variance of all states. This allows for an ever increasing variance. If one instead has a state dependent variance one could expect a similar result as in Q-learning, with the low variance terminal states slowly bounding the variance of the rest of the states.

Multimodal Returns

TODO:Is this actually a problem? It doesn't seem to be when I think about it. But I feel that these large changes here indicate we are partially modelling the wrong target. Won't $y^T y - \beta^T \Lambda \beta$ when $y = r$ push the model towards modelling the return distribution variance?

Another possible source is caused by the terminal state variance estimate. This is an issue when the model believes an episode will keep going but instead something unforseen happens resulting in the episode terminating. In games such as cartpole or acrobot the prediction will be of magnitude 10^2 while the target will be of magnitude 10^0. A direct effect of this is large drops in the scale parameter. However this also effects the mean estimate, pushing it to be lower to accomidate for the model taking wrong. This results in a higher target than prediction in all other states which pushes the variance up.

5.2 Method Improvements

5.2.1 Allowing for State Dependent Variance

A few attempts were made to make the variance state dependent. Due to computational and time constraints these are not included in this thesis. However this subsection highlight some of the possible methods for doing this and which are most promising.

One approach would be to use a random effects model(Gelman et al., 2013, p. 382-383) which allows different variance levels in different groups. The main challenge would then be how to group different state. This could either be done through manual feature engineer or some unsupervised learning method. The first in undesireable in the general case as it requires manual work and subjective modeling of the environment, reducing the geralizability of the model. The latter would require further research.

To avoid this one could instead set up a new model that encorporates the unequal variances and correlated errors by considering $y \sim N(X\beta, \Sigma_y)$ instead of $y \sim N(X\beta, \sigma_a)$. However these models are more complicated to setup and usually some prior knowledge about the structure of the covariance matrix. It could be possible to somehow encorporate the RL structure to define a weighting for bayesian weighted regression(Gelman et al., 2013, p. 370-373).

One way to define this weighting structure would be to use a model that assumes a known noise variance per datapoint. This is equivalent to a diagonal Σ_y. Then one could try to directly model the variance parameters by considering the strucuture of the RL problem. This approach is similar to the one introduced in O'Donoghue et al. (2017). However note that the resulting model would be substantially different as istead of propagating explicity through the uncertainty bellman equation the propagation would occur through sampling the target values.

5.2.2 Runtime

The runtime of the BNIG DQN model is worse than the DQN model. On acrobot the DQN model averaged 220 frames per second during training while the BNIG DQN model averaged 145. However the BNIG implementation has not been optimized. Notably the covariance posterior update can be reduced from $O(p^3)$ to $O(p^2)$ where p is the size of the final layer by using the Sherman-Morrison-Woodbury formula as in O'Donoghue et al. (2017).

5.2.3 Seperate N-Step

A minor improvement would be to allow seperate n-step updates for the neural network and the BNIG model. As seen in chapter 3 increasing the n can lead to much better variance propagation and better results in the RL setting. However a high n-step reduces performance and earlier papers have found $n = 3$ performs best on ALE(Hessel et al., 2017). However since the training procedures for the BNIG model and network are seperate one can use seperate n values for each model. This is used in O'Donoghue et al. (2017) where n is set to 150 for only the variance model.

Chapter 6

Conclusion

Motivated by the results from the UBE method(O'Donoghue et al., 2017) this thesis has tried to use bayesian linear regression to drive efficient exploration using thompson sampling. Through a linear toy example it was shown that previous bayesian linear regression RL methods do not properly propagate variance between states leading to an underestimation of variance. Furthermore it was shown that y sampling targets from the posterior and including the regression noise variance as a bayesian parameter proper variance propagation was acheived.

To generalize this method to more complex enviroments a generalization to neural networks was created based on the similar Azizzadenesheli et al. (2019) BDQN. The method showed promising results on some enviroments but had unstable noise variance estimates resulting is bad results on other environments.

This thesis suggests that further work should focus on making the variance estimate dependent on the enviroment state. This should both increase the stability of the variance and improve performance on enviroments that require complex exploration, such as Montezuma's Revenge.

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 doi:10.7551/mitpress/7503.003.0006.

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., Naddaf, Y., Veness, J., Bowling, M., 2013. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research 47, 253–279.

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.

Berger, J.O., 1985. Statistical decision theory and Bayesian analysis. Springer.

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

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

Fahrmeir, L., Kneib, T., Lang, S., Marx, B., 2013. Regression: Models, Methods and Applications. Springer Berlin Heidelberg.

Gelman, A., Stern, H.S., Carlin, J.B., Dunson, D.B., Vehtari, A., Rubin, D.B., 2013. Bayesian data analysis. Chapman and Hall/CRC.

Goodfellow, I.J., Mirza, M., Xiao, D., Courville, A., Bengio, Y., 2013. An Empirical Investigation of Catastrophic Forgetting in Gradient-Based Neural Networks. arXiv e-prints .

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.

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. 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.

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, arXiv:1709.05380.

OpenAI, . Openai gym documentation. URL: https://gym.openai.com/.

OpenReview, I.., 2018. Icrl 2019 open review of efficient exploration through bayesian deep q-networks. https://openreview.net/forum?id=B1e7hs05Km.

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.

Rasmussen, C., Williams, C., 2006. Gaussian Processes for Machine Learning. MIT Press.

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., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D., 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, 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.

Ting, J.A., D'Souza, A., Schaal, S., 2007. Automatic outlier detection: A bayesian approach, in: Proceedings 2007 IEEE International Conference on Robotics and Automation, IEEE. pp. 2489–2494.

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

Appendix A

Bayesian Regression Posterior Updates

A.1 Normal Prior with Known Noise

The following development of the posterior updates for the BN model is summarized from chapter 2.1 in Rasmussen and Williams (2006) but with the slight modification of allowing a non-zero mean prior.

Start with the linear regression model $y = X\beta + \varepsilon$. In this setting it is assumed that this models a deterministic process with mean $X\beta$ and variance σ^2. Assuming that the targets y_i are independent the likelihood function over the training data is given by

$$\begin{aligned} p(y|X,\beta) &= \sum_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - x_i^T\beta)^2}{2\sigma^2} \right) \\ &= \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} \exp\left(\frac{1}{2\sigma^2}|y - X\beta|^2 \right) \\ &= N(X\beta, \sigma^2 \boldsymbol{I}) \end{aligned} \tag{A.1}$$

Consider the bayesian setting where the coefficients β are random variables but the noise variance σ^2 is known. One choice of prior distribution is the normal distribution $N(\mu, \sigma)$ as this is a conjugate prior in this case.

Using bayes rule it can then be shown that

$$
\begin{aligned}
p(\beta|X,y) &\propto \exp\Big[-\frac{1}{2\sigma^2}\big(y-X\beta\big)^T\big(y-X\beta\big)\exp\Big(-\frac{1}{2}\beta^T\Sigma^{-1}\beta\Big)\Big] \\
&\propto \exp\Big[-\frac{1}{2}\Big(\beta-\tilde{\beta}\Big)^T\Big(\frac{1}{\sigma^2}X^TX+\Sigma^{-1}\Big)\Big(\beta-\tilde{\beta}\Big)\Big]
\end{aligned}
\tag{A.2}
$$

where $\tilde{\beta} = \sigma^{-2}(\sigma^{-2}X^TX+\Lambda^{-1})^{-1}(\Lambda\mu+X^Ty)$ and $\Lambda = \Sigma^{-1}$. This form matches the quadratic form of the normal distribution meaning the posterior coefficients are distributed

$$
p(\beta|X,y) \sim N\Big(\sigma^{-2}\big(\sigma^{-2}X^TX+\Lambda\big)^{-1}\big(\Lambda\mu+X^Ty\big),\quad \sigma^{-2}X^TX+\Lambda\Big). \tag{A.3}
$$

Here it can be seen that the normal distribution is in fact a conjugate prior as the posterior of β is also normal. The posterior update can finally be written as seen in equation 3.1.

A.2 Normal Inverse Gamma Prior

For the bayesian setting where both β and σ^2 are unknown the result is summarized from section 4.4.1 in Fahrmeir et al. (2013). In this case one keeps the normal prior over the β coefficients and assigns σ the prior

$$
\sigma^2 \sim \text{InvGamma}(a,b) = \frac{b^a}{\Gamma(a)}\frac{1}{(\sigma^2)^{a+1}}\exp\Big(-\frac{b}{\sigma^2}\Big). \tag{A.4}
$$

Under the assumption that the noise term is homoscedastic the joint prior can then be written as

$$
\begin{aligned}
p(\beta,\sigma^2) &= p(\beta|\sigma^2)p(\sigma^2) \\
&= \frac{1}{(2\pi\sigma^2)^{\frac{p}{2}}|\Sigma|^{\frac{1}{2}}}\exp\Big(-\frac{1}{2\sigma^2}\big(\beta-\mu\big)^T\Lambda\big(\beta-\mu\big)\Big)\cdot \\
&\quad \frac{b^a}{\Gamma(a)}\frac{1}{(\sigma^2)^{a+1}}\exp\Big(-\frac{b}{\sigma^2}\Big) \\
&\propto \frac{1}{(\sigma^2)^{\frac{p}{2}+a+1}}\exp\Big(-\frac{1}{2\sigma^2}\big(\beta-\mu\big)^T\Lambda\big(\beta-\mu\big)-\frac{b}{\sigma^2}\Big)
\end{aligned}
\tag{A.5}
$$

Bayes rule then states that

$$
p(\beta,\sigma^2|y,X) \propto p(y|X,\beta,\sigma^2)p(\beta|\sigma^2)p(\sigma^2).
$$

which gives

$$
\begin{aligned}
p(\beta, \sigma^2 | y, X) \propto \\
&(\sigma^2)^{-\frac{n}{2}} \exp\Big[-\frac{1}{2\sigma^2}(y - X\beta)^T(y - X\beta)\Big] \\
&(\sigma^2)^{-\frac{k}{2}} \exp\Big[-\frac{1}{2\sigma^2}(\beta - \mu_0)^T\Lambda_0(\beta - \mu_0)\Big] \\
&(\sigma^2)^{-(a_0+1)} \exp\Big[-\frac{b_0}{\sigma^2}\Big].
\end{aligned}
\tag{A.6}
$$

In section 4.5.2 in Fahrmeir et al. (2013) a proof is given that shows

$$
\begin{aligned}
(y - X\beta)^T(y - X\beta) + (\beta - \mu_0)^T\Lambda_0(\beta - \mu_0) \\
=& y^T y + (\beta - \mu_n)^T\Lambda(\beta - \mu_n) - \mu_n^T\Lambda\mu_n + \mu_0^T\Lambda\mu_0
\end{aligned}
\tag{A.7}
$$

where $_0$ denotes the prior and

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

which one recognizes as the posterior updates for the BN model without the noise term. This leaves the expression

$$
\begin{aligned}
p(\beta, \sigma^2 | y, X) \propto \\
&\exp\Big[-\frac{1}{2\sigma^2}\Big(\beta - \mu_n\Big)^T\Lambda_n\Big(\beta - \mu_n\Big)\Big] \\
&\frac{1}{(\sigma^2)^{a_0+\frac{n}{2}+1}} \exp\Big[-\frac{b_0 + y^T y - \mu_n^T\Lambda\mu_n + \mu_0^T\Lambda\mu_0}{\sigma^2}\Big]
\end{aligned}
\tag{A.8}
$$

which shows that the normal inverse gamma prior is a conjugate prior with the posterior update shown in equation 3.2.

Appendix B

Linear Model Experiments

B.1 2 State Propagation

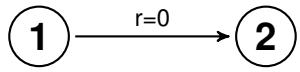


Figure B.1: 2 State Toy Environment

For the 2 state propagation experiments a regression model $\mathcal{M}$ with the form $y = X\beta + \varepsilon$ where $\beta = [\ \beta_1 \quad \beta_2\ ]$ was used. Since state 2 has a known posterior $\mathcal{M}$ only trains on state 1. This means x is always $[\ 1 \quad 1\ ]$. The pseudocode for the two target methods follows.

Algorithm 5: Sample Target Variance Propagation

```
Initialize a model of choice M
for iteration = 1, 1000 do
    y = sample from N(1, σ²)
    Update M priors given input (x, y)
end
```

Algorithm 6: Mean Target Variance Propagation

```
Initialize a model of choice M
for iteration = 1, 1000 do
    y = 1
    Update M priors given input (x, y)
end
```

B.2 N State Propagation

The N state propagation experiments were only running using BNIG models. One model was made per unknown state and each model was trained on one sample from the next state every iteration. This is summarized in the following pseudocode:

Algorithm 7: Multiple State Variance Propagation

$s =$ number of states
$\mathcal{M}$ = array of $s - 1$ of BNIG models
for *iteration = 1, T* **do**
  for *i = 1,* $s - 2$ **do**
    Sample β, σ from $\mathcal{M}$[i+1]
    Sample $\varepsilon \sim N(0, \sigma^2)$
    $y = x^T\beta + \varepsilon$
    Update $\mathcal{M}[i]$ priors given input (x, y)
  end
  $y =$ sample from $N(1, \sigma^2)$
  Update $\mathcal{M}[s-1]$ given input (x, y)
end

The 3 state experiments were run for 10000 iterations while the 6 state experiments were run for 1000 iterations.

Instructions on how to run the code for all the propagation experiments can be found in the readme file that comes with the provided code.

B.3 Linear Corridor Experiment

The code ReadMe contains instructions on how to run the linear corridor experiment presented in chapter 3. The hyperparameters used to create the linear corridor experiment figure 3.6 are given below. A short description per hyperparameter is also provided

B.3.1 Hyperparameters

ε-Greedy

The ε-greedy method has few hyperparameters. The learning rate is the learning rate used for the stochastic gradient descent over the regression parameters and is often denoted by α in RL literature. The ε term is the probability of performing a random action. This is decayed during training from some initial ε to a final ε over a set number of episodes.

Hyperparameter	Value
Learning Rate	0.01
Initial ε	1
Final ε	0.001
Num. episodes for decay	200

BN

The hyperparameters for the BN model consist of the priors, the ”known” noise variance and the decay rate for the exponential forgetting. The β prior mean is se equal for all coefficients, with a value denoted μ. The prior β covariance is set have the value σ_β^2 along the diagonal and zero otherwise. The noise term σ_ε^2 is assumed known but treated as a hyperparameter. Finally the decay rate defines the discount factor used for the exponential forgetting of previous data. The terms used to update the posterior are multiplied by this factor terms as in formula 3.5.

Hyperparameter	Value
Prior μ	0
Prior σ_β^2	0.001
σ_ε^2	1e-12
Decay Rate	0.999

BNIG

The BNIG hyperparameters are the same as the BN model apart from the known noise term which is replaced by the inverse gamma prior parameters. These two parameters are a and b.

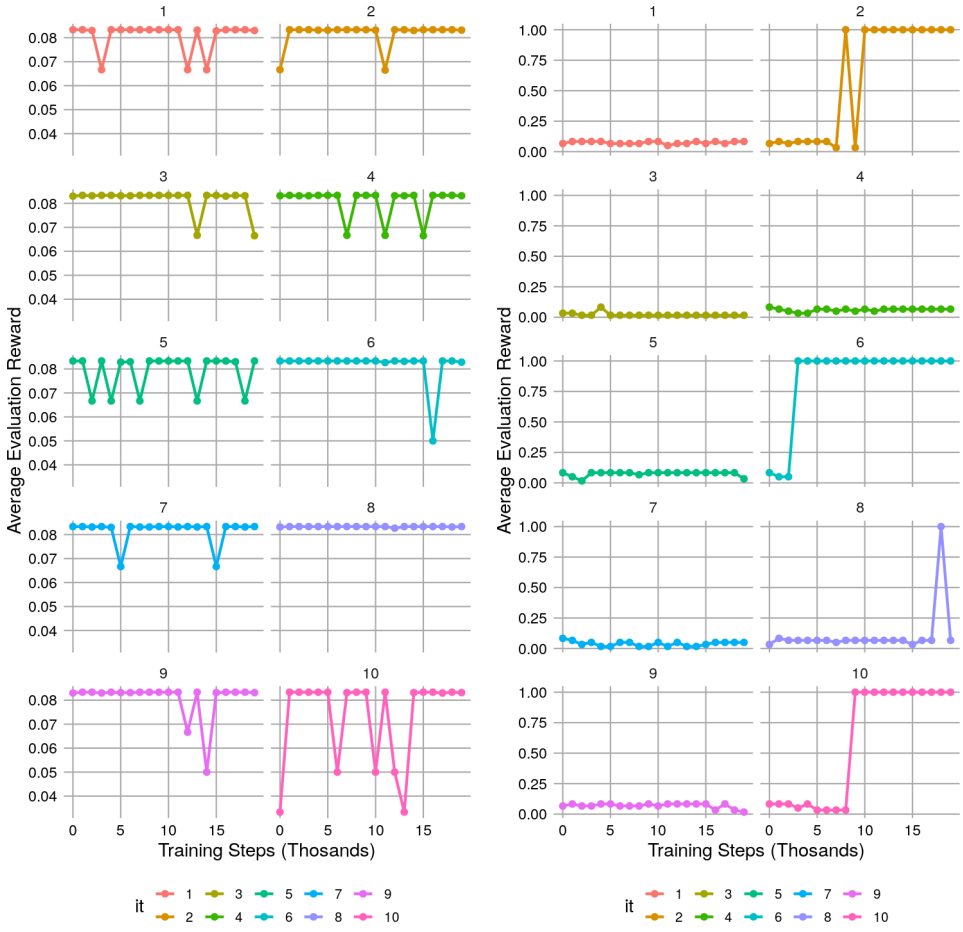
Hyperparameter	Value
Prior μ	0
Prior σ_β^2	1
Prior a	1
Prior b	0.01
Decay Rate	0.999

Appendix C

BNIG DQN Experiments

C.1 Single Seed Plots

C.1.1 Corridor

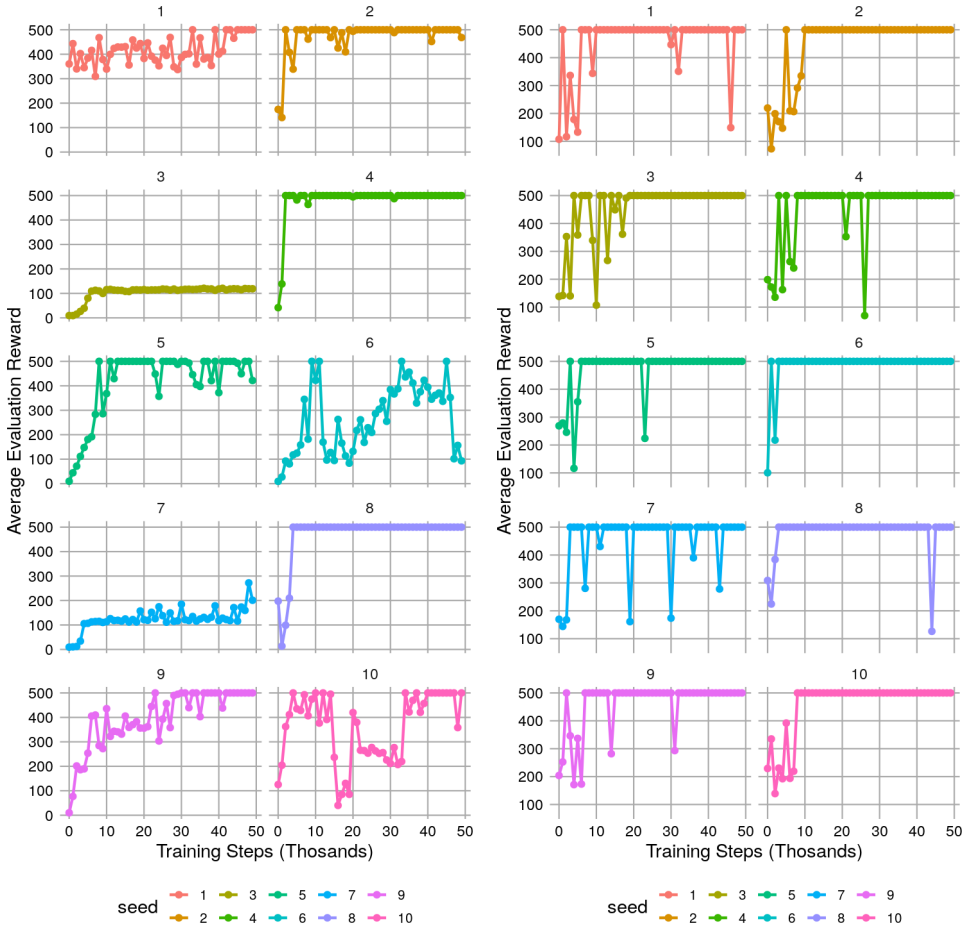


(a) DQN

(b) BNIG DQN

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

C.1.2 Cartpole

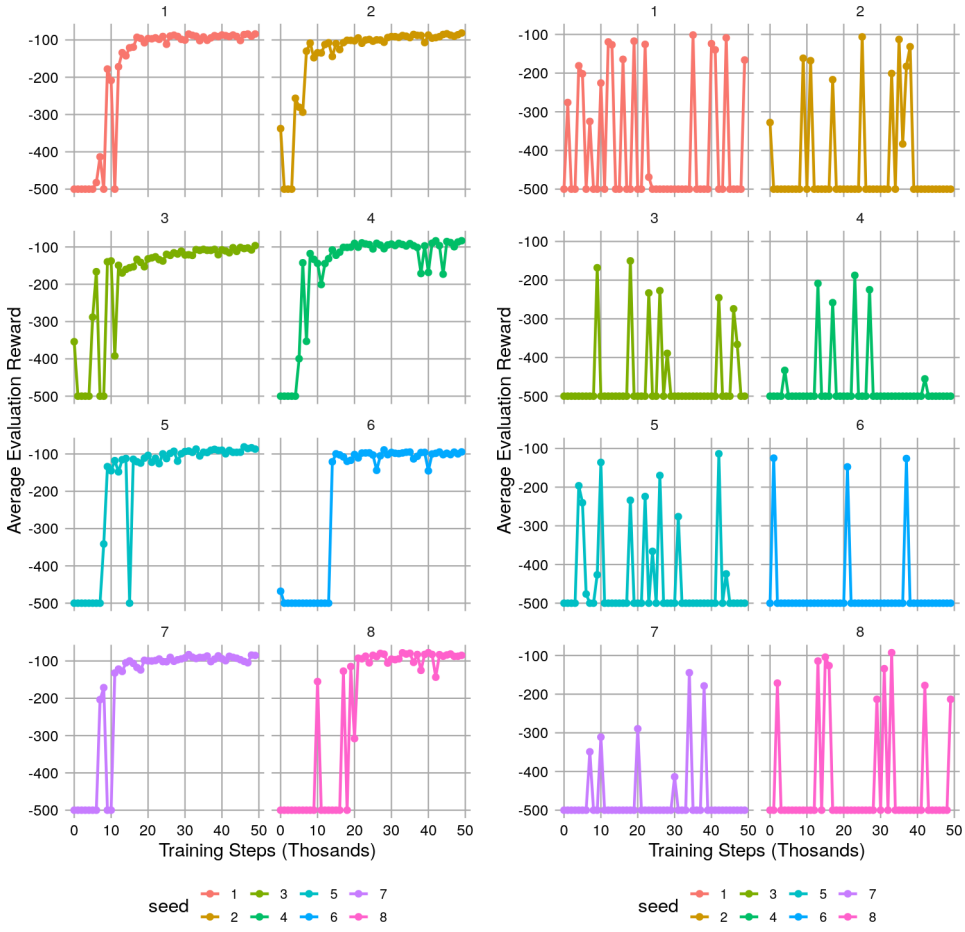


(a) DQN

(b) BNIG DQN

Figure C.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.

C.1.3 Acrobot



(a) DQN

(b) BNIG DQN

Figure C.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.