An Investigation into the Balance Between Exploration and Exploitation in Reinforcement Learning

Siphelele Danisa

Supervised by: Dr Jonathan Shock
Department of Mathematics and Applied Mathematics
University of Cape Town

Contents

Acknowledgements

I am profoundly grateful to Dr Jonathan Shock for all the opportunities for research he has given me since I started my undergraduate degree. This project has been life changing, to say the least, and I appreciate all the time he gave me during this journey as well as the others. I have learnt a lot, and I would like to think I that I have become a much better Mathematician through these opportunities.

Foreword

In Reinforcement Learning there is an agent whose task is to perform a certain duty. We define state and action spaces in a way related to the details of the problem at hand, and the agent can either learn to perform the duty in such a way that for every trial it places more value to the rewards of the immediate states compared to future states–we say that the agent takes actions greedily– or it can predominantly take greedy actions while occasionally exploring by taking random moves. This is the *exploration-exploitation balance*. Usually, a parameter is assigned to this trait, ϵ, where $\epsilon = 0$ implies greedy behaviour while $\epsilon = 1$ is the other extreme. In this project, we wish to find an algorithm that will find the optimal epsilon, i.e an epsilon that offers the best balance between exploring and exploiting. After that we shall use this algorithm in Grid World, discuss some pitfalls of this implementation and then investigate transferability of knowledge in this context in attempts to mitigate the pitfalls we shall mention. We shall introduce all necessary information as we move along, and in cases where there is information to share that is not directly related to the project itself, we shall include the details as an appendix instead.

Introduction and Background

Machine learning is often defined as an application of artificial intelligence that affords systems the ability to automatically learn and improve from experience without being explicitly programmed. Other views include, but are not limited to, the statistical sciences perspective in which machine learning is merely computational methods applied to statistics. In the standard setting, there are various types of machine learning such as supervised learning wherein the training data has correct labels as well as reinforcement learning. The goal in the latter is to learn optimal actions from past experiences. The rest of this project has this particularly at heart.

Reinforcement learning is a framework for solving reward-based problems. This field is largely inspired by the natural way in which humans and animals learn to perform various duties without an instructing figure, i.e by interacting with the environment. Mathematically, this framework can be describe as will be done in what follows.

Let $\mathcal{S}$ be a set of states, and (s_t), $t \in \{0, 1, \cdot, T-1\}$ be a sequence of states. If it is true that

$$\mathbb{P}(s_{T+1}|s_T) = \mathbb{P}(s_{T+1}|s_0, s_1, \cdots, s_T)$$

then we say that the sequence of states satisfies the Markov property. We shall always assume that the probability of transitions is independent of the time variable. The transition probability matrix denoted $\mathcal{P}_{ss'}$ is defined as $\mathbb{P}(s_{t+1} = s'|s_t = s)$. Furthermore, we define a Markov Decision Process as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \mathcal{R})$ where respectively the elements represent a finite set of states, a finite set of actions, the state transition probability matrix $\mathcal{P}^a_{ss'} = \mathbb{P}(s_{t+1} = s'|s_t = s, a_t = a)$, the discount factor–this takes a value in the unit interval – as well as a reward function from $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ to $\mathbb{R}$. All of the mentioned facts are used to model the environment in reinforcement learning. We see that in the Markov Decision Process transitions between states depend on both the current state at a given time as well as the action taken at that time from the state, and there is an associated reward to all state-action pairs.

The goal is to find a way to maximise some defined expected return. A return G_t can be defined as the total discounted reward from time t. We write

$$G_t = \sum_{k=1}^{\infty} \gamma^k R_{t+k+1}.$$

The discount factor is introduced for various reasons including the uncertainty handling concerning future rewards, to avoid infinite returns in cyclic processes, to handle various contexts where, for instance, states close to some current state have a different worth compared to those that are farther.

We define a policy as a distribution over actions given states. We denote this as $\pi(a|s) = \mathbb{P}(a_t|s_t = s)$. A policy serves the role of a guide for choosing actions given states. The goal is to then find a policy that maximises the return, we call this the optimal policy. In order to do this we define value functions as follows.

The state value function, v_π, for a Markov Decision Process is the expected return with s as a starting point, then following some policy π, that is $v_\pi(s) = \mathbb{E}_\pi(G_t|s_t = s)$. In fact, with very little effort we can express this in terms of the immediate reward for a state s_t and discounted value of successor state s_{t+1} as

$$v_\pi = \mathbb{E}_\pi(R_{t+1}|s_t = s) + \mathbb{E}_\pi(\gamma v_\pi(s_{t+1})|s_t = s).$$

Define $\mathcal{R}_s^a = \mathbb{E}_\pi(R_{t+1}|s_t = s, a_t = a)$, then we obtain

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)\Big(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\Big) \tag{1}$$

Proceeding similarly, we define a state-action-value function, $q_\pi(s, a)$, as the expected return from state s taking action a then following the policy. Explicitly, this is
$q_\pi = \mathbb{E}_\pi(G_t|s_t = s, a_t = a)$, and we can similarly break this expression into parts that are easier to make sense of intuitively as shown in what follows. We have that

$$q_\pi(s, a) = \mathbb{E}_\pi(R_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1})|s_t = s, a_t = a).$$

Rewriting this we obtain $q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$, and lastly, putting everything together, using $v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(s, a)$, we write q_π in the same form as (1) to obtain

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a'\in\mathcal{A}} \pi(a'|s')q_\pi(s', a') \tag{2}$$

The equations (1) and (2) are known as Bellman equations, and they are quite useful in computing the value functions. The computations are done through a variety of methods including monte carlo methods, dynamic programming as well as temporal difference learning.

When talking about optimal value policies, we are concerned with $v_*(s) = \max_\pi v_\pi(s)$ as well as $q_*(s, a) = \max_\pi q_\pi(s, a)$ for all $a \in \mathcal{A}$, $s \in \mathcal{S}$. We say $\pi \geq \pi'$ if for all $s \in \mathcal{S}$ we have that $v_\pi(s) \geq v_\pi(s')$. The optimal policy satisfies $\pi_* \geq \pi$ in the set of all policies π.

In fact, given a Markov Decision Process we can be sure that there exists an optimal policy, but this need not be unique, and all optimal policies achieve optimal value functions.

Having said all that we shall now give what is known as the Bellman optimality equations. Let $\pi_*(a|s) = 1$ if $a = \text{argmax}_{a\in\mathcal{A}} q_*(s, a)$ and 0 otherwise, then we have $v_*(s) = \max_a q_*(s, a)$, so that

$$v_*(s) = \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')\right) \tag{3}$$

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a') \tag{4}$$

These are key elements of the standard Reinforcement Learning formulation [1][9][8][26].

One has probably noticed that in the above we only focused on actions that would yield the highest immediate reward for the derivation of the optimality equations. These are

greedy actions. In fact, we need not do this. It is well-known that greedy methods are not necessarily the best in solving problems, and a reasonable alternative is ϵ-greedy methods. For this class of problem solving methods, we merely add a condition that for some $\xi \in [0, 1]$ chosen randomly, whenever $\xi < \epsilon$ choose a random move instead of the greedy one, and we say that the agent is exploring the state space. As shown in [8] [16] this method is quite useful in discovering more optimal policies.

The intermediate goal in this project is to present a simple but effective algorithm for finding an optimal epsilon for ϵ-greedy implementations. It is worth noting that the process for searching for any hyper-parameter is similar.

The Learning Framework

The goal for this section is to merely provide a few definitions, justify why they are reasonable constraints and prove a few propositions. The reader will hopefully be convinced if not after this section then on upcoming sections that all the assumptions we make are not too restrictive.

Definition 1 (Ordered Sets) *Let P be a set. An order (or partial order) on P is a binary operation $\leq$ on P such that for all $x, y, z \in P$:*
1. $x \leq x$; 2. $x \leq y$ *and* $y \leq x$ *imply* $x = y$; 3. $x \leq y$ *and* $y \leq z$ *imply* $x \leq z$.

This definition is as found on [12]. It would seem like orders can be applied on numerical contexts only but this is far from true. Given any context we can apply an order on related concepts, and this is what we shall do below. A brief introduction of how lattice and order theory can be used in general contexts is given on appendix A. The theory is inspired by the already mentioned resource. We hope that this will convince the reader that this setting is fitting for our purposes.

Definition 2 (Well-Posed Problems) *Let P be a problem. P is well-posed if it is solvable, i.e there exists a solution to the problem. Moreover, we assume that there is a unique solution to problems in consideration.*

The second part of this definition, uniqueness, means that we assume that the solution space is ordered in the sense of the definition, i.e if there are two solutions Q, Q', then we can define an order on the solution space so that either $Q \leq Q'$ or $Q \geq Q'$ where the former means that Q' is more optimal, and does not exclude equality where appropriate. This assumption is reasonable in the sense that, if we have multiple optimal solutions then we effectively have a smaller search space, so the problem is slightly easier in comparison. There are solution spaces where this structure does not hold, i.e where there is not one optimality, and so we obtain for instance objects whose underlying structure is a partially ordered set, $(P, \leq)$, without an element t such that for all $t \geq x$ for all $x \in P$. Such an element t is called a top element. Usually, instead of this we have maximal elements, i.e $s \in P$ such that $x \geq s$ implies that $x = s$ for all $x \in P$. These elements are not comparable with each other if there is more than one, hence we cannot say which one is preferable without context. This would be the more general structure we would want, but in order to simplify things we consider the case where we have a top element, at least on a local scale.

Let P be a problem whose framework involves a finite number of states and a finite set of appropriate actions. Suppose for each state there are at most n possible actions, and that the sequence-term $((\chi(P))_{k,j}$ is the outcome from taking some action given the $k-1$ state, for the j-th training trial.

Definition 3 *Suppose that given a problem P we can employ a algorithm χ to obtain $\chi(P) = Q$, the solution. In particular, we consider $X(P)$ as a sequence of states and actions (on these states) that ultimately achieve Q. Let Q^* be the optimal solution. If there exists, ρ, a monotone decreasing function to zero such that:*
1. $\rho \to 0 \iff \chi(P) \to Q^*$,

2. If $\mathcal{Y}$ is set of tuples (s, a), the states already visited as well as the actions taken on those states, then $\rho_{k,j} < \rho_{k-1,m} \iff$ for $((\chi(P))_{k,j}$, $((\chi(P))_{k-1,m}$ with $m \leq j$ we have that $(s_k, a_k) \notin \mathcal{Y}$,
then we call χ a learning algorithm.

We are using this definition as a necessary condition, and we do not claim it is sufficient. The monotonically decreasing function is uncertainty about the environment with respect to the duties to be fulfilled. In this view, all non-trivial moves decrease uncertainty about the environment. It is clear that the definition above is essentially an order on the experience of the agent.

Consider a learning problem, P, that is solvable for some $\epsilon \geq 0$ (i.e we consider a problem that can be solved using a learning algorithm in an epsilon greedy sense) with, say, Q as the solution. We claim that it is always possible to find an $\epsilon' \in [0, 1]$ such that there is a Q' for which $Q \leq Q'$.
Proposition 1 *There exists an epsilon that improves learning for any well-posed learning problem.*

Proof. The existence of a solution by assumption (say Q) implies that the problem can be solved by some $\epsilon \geq 0$. Consider a second solution to the problem, Q' with ϵ'. If $Q \leq Q'$, let $\epsilon = \epsilon'$, otherwise $\epsilon = \epsilon$. If this process is finite, then all is well. Now, suppose there is a sequence of progressively optimal solutions converging to Q^*, $(Q_k)_{k\in\mathbb{N}}$, then we have a sequence of epsilons $(\epsilon_k)_{k\in\mathbb{N}}$. It is sufficient to note that since $(\forall\delta > 0)(\exists N \in \mathbb{N})(k \geq N)(|Q^* - Q_k| < \delta)$, we can merely take ϵ_k. □

In fact, the existence of an epsilon for Q^* then follows from the following theorem.
Theorem 1 ([15], Theorem 3.7) *The set of all subsequential limits of a sequence (p_n) in a metric space X is a closed subset of X.*

In order to see this, we only have to realise the sequence of solutions as a sequence of limit points of convergent subsequences, then we can conclude given the assumption of the existence of a globally optimal solution.
Corollary 1 *Let P be a solvable problem, then $\mathcal{Z}$, the set of possible solutions, with the assumptions already considered is a closed subset of $(\mathbb{R}, |\cdot|)$.*

The proof immediately follows from the theorem as well as the argument on the relevance of the theorem to the arguments of the proposition. Another result which might be useful is the following result in approximation theory is the following proposition,
Proposition 2 *Consider a sequence of functions $f_k : X \to \mathbb{R}$ such that $f_k \to f$ with convergence in the Euclidean metric. If $\tilde{f}_k = f_k$ except for $x \in A_k \subset \mathbb{R}$ such that $|A_k|_{k\to\infty} \to 0$, then $\tilde{f}_k \to f$.*

Proof. We only have to note that the triangle inequality gives $|\tilde{f}_k - f| \leq |\tilde{f}_k - f_k| + |f_k - f|$, so that $|\tilde{f}_k - f| \to 0$. □

We consider the definition of convergence as it is in classical analysis literature, using the metric space $(\mathbb{R}, |\cdot|)$ unless stated otherwise, i.e. Consider a metric space (X, d) and $(x_n)_{n\in\mathbb{N}}$

be a sequence in X, then we say $x_n \to x$ if

$$(\forall \epsilon > 0)(\exists N \in \mathbb{N})(\forall n \geq N)(d(x_n, x) < \epsilon)$$

.

Related Literature

There is a lot of work related to this problem either in a direct sense or not, and we shall mention a few. The paper [7] discusses the efficiency of random parameter searching as compared to other alternatives. Similar ideas to the algorithm have been around for some time, for instance [17] has the same objective as we do. On the other hand, [18] is much closer to what we present although arguably more constrained in comparison, in addition to not having theoretical justifications. The paper [3] offers a very captivating idea as well by adding a penalising layer on neural network to penalise the learning, but the implementation does get quite involved, and the same goes for [2] whose fruits grow from the same tree, etc.

The Search for Optimal Parameters

At this point we wish to introduce an algorithm for finding optimal epsilon. We shall give a brief description of how the algorithm works, then proceed to provide more formal argument on why we expect convergence for this algorithm. Let's introduce some notation for the algorithm. Given some array $\mathcal{R}$, $\mathcal{R}[-k] =: r_{-k}$ for some $k \in \mathbb{N}$ denotes indexing from the end of the array. We let P denote a problem whose solution is (or is approximated by) Q. The function $sgn(\nabla\epsilon)$ is the sign function determined by the sign of $\tilde{\epsilon^*} - \tilde{\epsilon}$, i.e the difference between the approximately optimal epsilon from the previous execution and the current epsilon. We present the following algorithm in order to compute the optimal epsilon.

Algorithm 1: Epsilon-Greedy-Epsilons

Initialise some storage array, K.
repeat
  Input: P,ϵ
  Sweep the solution space until some approximate solution is found.
  Output: Q
  Store Q in some array, $\mathcal{R}$. Store both Q and ϵ in K
  Let (as real numbers) $\gamma, \delta, M, N > 0$.
  if $|\mathcal{R}| > 1$ **then**
    if $r_{-1} > r_{-2}$ **then**
      $\epsilon \leftarrow \epsilon + \beta \times sgn(\nabla\epsilon)|r_{-1} - r_{-2}|$ with $\beta \in \mathbb{R}_{\geq \mathbb{0}}$
    else
      if $\gamma < M\epsilon + \delta$, $M \propto |r_{-1} - r_{-2}|$ **then**
        $\epsilon =$ random()
      else
        $\epsilon = \epsilon$
  else
    if $\gamma < N\epsilon$ **then**
      $\epsilon =$ random()
    else
      $\epsilon = \epsilon$
until $\tilde{N}$ *times*;
Clear $\mathcal{R}$

The idea is that in the beginning we allow and encourage the algorithm to explore epsilon values, then as it gets more and more solutions whose efficiency measures are quite close to each other, we want there to be less randomly chosen epsilons. Observe that the term $|r_{-1} - r_{-2}| \to 0$ as the agent learns enough to find approximately optimal solutions, and is equal to zero whenever the most recent approximations are equal, i.e $r_{-1} = r_{-2}$. So we obtain varied exploration rates whenever there are prevalent inconsistencies, then this quiets down as the learning progresses. The *'quieting down'* is ensured by the fact that

the problem is assumed to be solvable, and that we are using a learning algorithm in the above context. It is easy to note that although we have kept more generality with regards to the constants (for example, M, N, β) it is possible to have a similar process for this kind of search, but it gets a bit tedious. If all else fails, one can simply do a random search for the parameters, which has strong practical backing to be the best method of going about this particular task. The weights N, M can be tuned so that exploration does occur often or does not. The parameter β can be tuned similarly, and is effectively the learning rate.

Step (8), as can be seen, takes a lot of inspiration from the gradient update approach. The update is towards the epsilon that yields a more optimal value. This is chosen as it is the most natural candidate in the sense that we obtain the values by solving the objective (P, Q) problem without further work requiring a lot of function evaluations. A natural improvement to this algorithm would be to incorporate the Upper Confidence Bound Action Selection [8]. Not only that, but we can compute dynamically to avoid keeping large arrays. It is interesting to note that (8) is also quite close to evolutionary techniques [5], but of course, the presented algorithm is designed to be faster in searching the space for a solution following the explorative behaviour in addition to the perturbative search.

Convergence Theory

The convergence on the proposed algorithm relies heavily on the convergence of algorithm used to solve the original problem, and we would now like to show that as long as we have a convergent algorithm everything will work within expectation.

Proposition 3 *If χ is a learning algorithm that converges with probability 1 greedily, then it convergences in the epsilon greedy context.*

Proof. Since $\chi(P) = Q$, we have a sequence $(Q_n)_{n\in\mathbb{N}}$ such that $Q_n \to Q$. Suppose $\epsilon \geq 0, \xi \in [0,1]$, and that $\tilde{\chi}(P,\rho)$ corresponds to the exploratory path.

$$\chi_\epsilon(P,\rho) := \begin{cases} \chi(P,\rho) & \text{whenever } \xi \leq \epsilon \\ \tilde{\chi}(P,\rho) & \text{otherwise} \end{cases}$$

We have that $(\forall \epsilon > 0)(\exists N \in \mathbb{N})(n \geq N) \implies |Q_n - Q| < \epsilon$, then there is a subsequence (Q_{n_k}) such that $n_k \geq n$. Consider $n_k \geq j$, then it follows that $Q_{n_k} \to Q$ as $n \to \infty$, and this is sufficient for conclusion since $\rho_{n_k,j} \to 0$ as $j \to \infty$, hence $\chi(P)_{n_k,j} \to Q'$ where $Q' \geq Q$. □

It follows from proposition 1 then that we could obtain another convergent sequence that improves the convergence. Another useful result from [15] is Theorem 3.6,

Theorem 2 ([15], Theorem 3.6) *Every bounded sequence in R^k contains a convergent subsequence.*

The assumption of boundedness follows from the assumptions of existence and uniqueness of solutions. This offers and alternative idea for the proof.

Suppose then that the algorithm that solves the original problem is convergent, call this χ.

Theorem 3 *Algorithm 1 converges to an optimal epsilon (i.e an epsilon for which we get Q^*).*

Proof. We have that χ converges for any epsilon. If the process $\chi(P)$ is finite, then we have that K is finite, and by considering the order on the space of solutions we obtain the desired result as done previously. In the limit $t \to \infty$, ϵ will take all values on $[0,1]$ $(*)$. So if the process if infinite, then suppose that there is exactly one such epsilon. Write for some $t = k \in \mathbb{N}$, $\epsilon_k = \epsilon$. Let $r_k := |\epsilon^* - \epsilon_k|$. It is clear that for $t = j$ with $j > k$, if (7) is true, then $r_j < r_k$. We observe that $\sup_k r_k = 0$, since otherwise there is a $\kappa \in (0,1]$ such that $\kappa \leq r_k \forall k \in \mathbb{N}$, so that $\epsilon \notin B_\kappa(\epsilon^*)$, contradicting $(*)$. Hence we obtain a sequence of decreasing r_k-radius balls around ϵ^*, and from this we obtain a sequence $(\epsilon_k)_k^\infty$ such that $\epsilon_k \to \epsilon^*$. □

If the epsilon is not unique, then again we have a better chance of converging to an epsilon. In fact, although the algorithm itself might not converge in this case it will cycle through the epsilons, and because all of them are optimal this still achieves the desired result.

We can also note the following. If there is a sequence of problems, $(P_k)_{k=1}^{\infty}$ such that $P_k \to P$, and we have an algorithm χ as in the above context such that we not only know $\chi(P_i)$ for all $i \in \mathbb{N}$, but we also know $\chi(P)$. Then it is interesting to ask whether or not we would ordinarily expect $\chi(P_k) \overset{k\to\infty}{\longrightarrow} \chi(P)$.

In order to answer this, we need to know precisely what this means. We would essentially have that

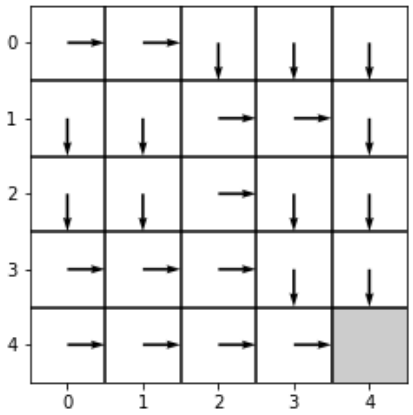
$$(\forall \epsilon > 0)(\exists \delta > 0)(|P_k - P| < \delta) \implies (|\chi(P_n) - \chi(P)| < \epsilon)$$

This is exactly the definition of continuity of maps. This is not related to what we would like to achieve, so we shall not be spending time on it, but is nonetheless worth mentioning.

Introduction to Grid World

We would now like to put the algorithm to practice, and we choose to do this in the Grid World setting. Grid World is an environment where there is a $2D$ grid, and the agent has to learn the optimal way to traverse the grid to some cell, say (i, j) from (p, k) where we are labelling the cells as coordinates in the usual sense. We consider the simple case where the agent starts at a point $(0, 0)$ on the grid and finds an optimal policy to the point $(N-1, N-1)$. For instance, we can consider the grid world without any obstacles whose size is $N = 5$, and we obtain the following plots

(a) State Action Value Plot



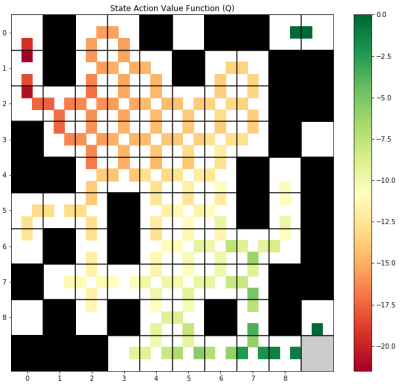
(b) Greedy Moves

Figure 1: Example: $N = 5$

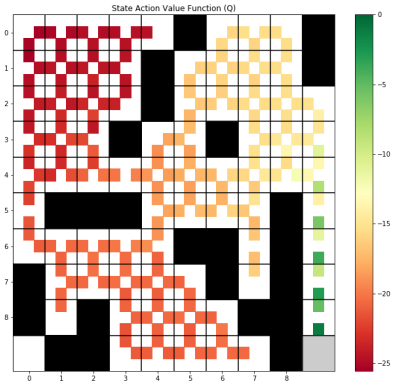
We initialise the environment as well as the agent, and the agent is penalised for staying in one place, as well as for being in any state that is not terminal. This encourages the agent to move in order to minimise the loss. Figure 1 (a) is a plot of the grid showing how this

loss decreases as the agent traverses the grid towards the terminal state. The other plot figure 1 (b) shows possible optimal routes from the initial states. It is important to note that these are not hard-coded but rather the agent explores and discovers these paths.

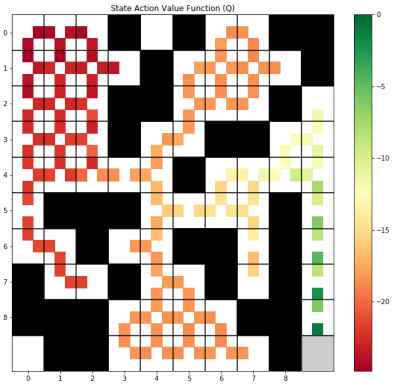
Next we generate a few obstacles (denoted by black cells) on the grids, and we do this randomly keeping the grid size fixed. In doing so we obtain the following sample grids using $N = 10$. In fact, this is the type of data used in the rest of this project.



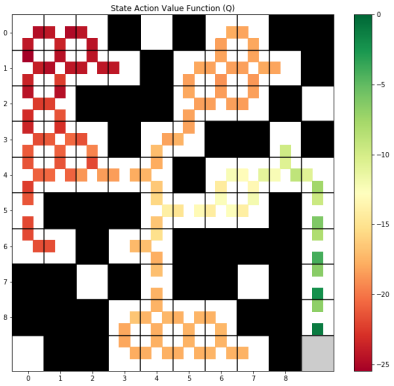
(a) Random Grid I



(b) Random Grid II



(c) Random Grid III



(d) Random Grid IV

Figure 2: More Generated Random Grids for $N = 10$

Learning on Grids

In the first section, we mentioned methods for computing the value functions amongst which were monte carlo methods, dynamic programming and temporal difference learning. Dynamic programming algorithms are used to compute optimal policies given a perfect model like a Markov Decision Process. This assumption, of course, limits the power of applicability. Monte carlo methods are used similarly, but they only require experience, and learning is an inference process based on average sample states, actions and returns. This is a more robust method. Temporal difference learning is a generalisation of these two methods wherein instead of, for instance, waiting until the return following the visit is known then using that return as a target on updates, we need only wait until the next step for temporal difference learning. Like dynamic programming methods temporal difference methods bootstrap, and so once can see that temporal difference methods have useful properties derived from monte carlo methods as well as dynamic programming methods. An example of such is TD(0), and we provide the algorithm above.

In terms of notation, we can read $V(S)$ as $v_\pi(s_t)$ in our usual notation. This is also known

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
    $S \leftarrow S'$
  until S is terminal

Figure 3: TD(0) [8]

as one-step temporal difference learning, because it is an special case of a more general algorithm that uses $n-$step returns for updates, and this can be see from the update equation of the method being

$$v_\pi(s_t) \leftarrow v_\pi(s_t) + \alpha[R_{t+1} + \gamma v\pi(s_{t+1}) - v_\pi(s_t)] \tag{5}$$

where $v_\pi(s_t)$ is the estimated value function at s_t.

We now have an idea of how the ideas in the background section are implemented to find the value functions, and so we proceed to show how we find the policy. There are two cases to consider here namely on-policy and off-policy learning. The on-policy approach learns action values not for the optimal policy but for an approximately optimal policy

that still explores. This is some sort of a compromise. If we use two policies where one is exploratory while the other is being learned about and becomes optimal, then learning is achieved through this exploratory policy which is often referred to as a behaviour policy, and using this some target policy is achieved. Off policies are usually slower to converge while offering more varied experiences, and thus rewards, but they are more general. In fact, on-policy methods are a special case where the target policy and behaviour policy are the same. However, given an on-policy method we can iterate greedily over the achieved policy obtain to attain the same optimality.

For this project, we use an on-policy algorithm known as Sarsa on the temporal difference setting as well as policy iteration to make policies optimal. We give the algorithm below. In what follows α is called a learning rate, and this is merely a scaling of the update.

Sarsa: An on-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figure 4: SARSA [8]

In producing the plots that we have seen so far, we used exactly these algorithms. Let n be the number of training trials on the grids. We set the parameters as follows.

Parameter	Values
γ	0.99
α	0.35
n	125

After this, we then we plotted the state-action values as can be seen on the images with colour mapping together with the greedy paths as can be seen in the first two plots on the previous section. In everything that follows from this point, we shall use these values exactly unless stated otherwise.

Grid Size and Computational Complexity

As one would expect, increasing the grid size leads to more computations, and as a result the rest of this project uses $N = 4$ for grid size unless otherwise stated. In order to quantify the computational cost we note that given a $N \times N$ grid, the optimal policy π^* has length $|\pi^*| = 2N$, so for any other policy π we have that $|\pi| \geq |\pi^*|$ (or $\pi^* \geq \pi$). Moreover we have

$$\binom{2N}{N} = \frac{(2N)!}{N!(N!)}$$

ways of traversing the grid efficiently, because essentially $N+N$ steps towards the right and down in any order are required in order to solve the problem. The paper [10] deals with the problem of finding the number of self-avoiding walks between arbitrary points on a given grid of size N. This turns out to be a hard problem. They optimize an existing algorithm in order to compute the number of paths for $N = 25$ a number after which convergence takes a very long time.

Let $\Omega = \binom{2N}{N} \times 2N$. This is the possible number of states given the different ways of solving the problem efficiently. Taylor series analysis, thanks to Wolfram Alpha, yields that

$$\frac{\Omega}{2N} = \binom{2N}{N} \sim \left(\frac{\sqrt{1/N}}{\sqrt{\pi}} - \frac{\left(1/N\right)^{\frac{3}{2}}}{8\sqrt{\pi}} + O\left(\left(\frac{1}{N}\right)^2\right)\right)\exp\left(\log(4)N + O\left(\left(\frac{1}{N}\right)^3\right)\right) \quad (6)$$

We give a plot to give some sense of fast this number grows in Figure 5.

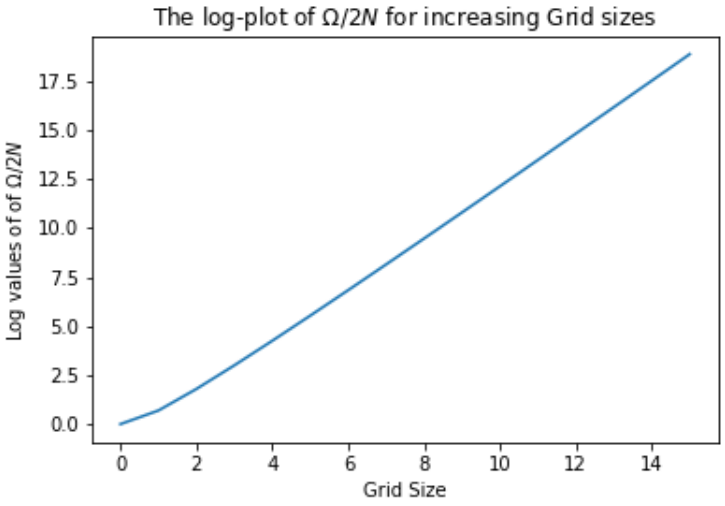


Figure 5: Visualising the Growth of $\Omega/2N$

This growth is exactly what we see in the Taylor series expression. In particular, the logarithm has predominantly linear behaviour, which implies that the object $\Omega/2N$ grows

exponentially. We care about this because we need to generate a significantly huge number of trials, and the plots in some sense give us the minimum number of computations required to have approximately fully explored the space of solutions of interest. In order to generate training data then, a few quality samples would take a lot of time to produce for large N, and this is simply inefficient far more than is desirable. An appendix, B, with more details on the asymptotic analysis is available.

A Short Investigation into Grid Orders

In the context of ordered sets, we would like to define some notion of order in the space of modified grids. We want to answer the following question: Given two grids of size N, say G, G' which have respectively k, l obstacles where $k, l \in \mathbb{N}$ and $k, l \geq 0$, what are reasonable ways to put an order on the '*complexity*' of the grids?

In other words, we want to be able to say that, for instance, in G the agent will find the optimal path more easily than in G' given any two grids G, G'. It is clear that putting obstacles in some grid makes some paths obsolete. It is interesting to note how this affects the time it takes to find the optimal path. A brute force approach to the problem is to consider constructing many random scenerios and use the *average experience* to draw conclusions, otherwise we might try to see how the connectedness of the grid-structure relates to the average time it takes to find the optimal path. This is of interest because it affords us better understanding of the problem. A handful of different methods were investigated, but we only present the mentioned two. The procedure for investigating the rest was done similarly.

Random Walks on Graph

Consider a random walk X on a grid. A random walk is a path traced by an agent starting at some initial state and selecting an action from the list of available actions randomly, i.e at every time step t we assume that there is uniform preference of all available actions, and one is selected randomly. If we consider a random walk over a finite grid, in the limit as time t goes to infinity, we obtain a nonzero probability that all states will be explored after time τ where this has to be at least $2N$ where N is the size of the grid. If N, the grid size, is large then so is τ. A practical way of answering the question posed is to find the average time it takes for a random walk to hit the states of interest, i.e we want the number of steps it takes on average for the random walk to go from some initial state to the state of interest. This method is more robust, because it involves computing the quantity of interest directly from the grids in the same spirit that an agent will when it is still learning in its environment.

Graph Connectivity

A graph is a pair (V, E) where $\emptyset \neq V$ is the set of vertices/nodes and E is the set of edges between the vertices. A grid can be visualised as a graph with very little work where the nodes are the states, and the edges a representative of paths between the cells. For example, consider the following,

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4
4,1	4,2	4,3	4,4

(7)

The entries in the grid are the coordinates of the cells, i.e each cell is associated with a pair of numbers k, j and we write (k, j). The graph representation can be realised in the following way.

- Start at, say, (n, n').
- Consider the possible states with *effective* moves, i.e moves that are towards the target cell.
- Construct edges from (n, n') to (a, b), (c, d) from above, then repeat for each of these new states until you have the desired terminal state.

As an example using the above grid, from (cell) $(1, 1)$, if you are moving to $(4, 4)$, then effective moves will lead to $(1, 2)$ and $(2, 1)$. From here we observe that from $(1, 2)$ we can move to $(1, 3)$ or $(2, 2)$ and from $(2, 1)$ we can move to $(3, 1)$ or $(2, 2)$. Proceeding this way, we obtain a grid representation. Although we only speak of effective moves, we do not put direction to the edges. Considering *all* possible states, we also obtain the same representation, but it gets messy too quickly. Starting at cell $(1, 1)$ to cell $(4, 4)$ produces a graph representation that is exactly like the grid itself.

We say that a graph is connected if between any two vertices there is an edge from one vertex to the other. Otherwise, the graph is disconnected. If G is a graph S is a set of vertices of G. If G is connected and $G \backslash S$ is disconnected then S is said to be a vertex-cut. A graph G is said to be complete if every edge is connected to all the other edges, and incomplete otherwise. Given a connected graph G we define the connectivity as

$$\kappa(G) = \begin{cases} \min |S| : S \textit{ is a vertex-cut of } G, & \text{if } G \text{ is not complete} \\ n - 1, & \text{otherwise} \end{cases}$$

Loosely speaking, connectivity is the minimum number of vertices you have to remove in order to disconnect a given graph structure.

A Brief Contextual Discussion

There are two observations that one can make easily. If a graph G is more connected than G' then there are most likely more ways of going from one vertex to another. This would be a good thing if the agent would take *good* moves from the start, then the movement would be towards the desired cell and any path taken would lead to a win. The price to pay is that initially, the agent is learning and this means that it does not take *good* moves only. The curse of having more paths to take is that there are more ways of *getting lost*. We want to investigate how this affects the expected time taken to hit the terminal state.

Hence, we do the investigation as follows:

- Given a graph G. Compute the number average time it takes a random walk to hit the states of interest, τ, and the connectivity of the graph.
- Given a second graph G', we do the same.

- After generating a sufficiently big number of different graphs we investigate the relationship between the τ's and the connectivities.

Numerical Implementation of Methods

As mentioned above, the grid variation procedure for our context involves putting obstacles on the cells. This is essentially taking out vertices on the graph. In this section, we generate grids whose order we compute using these two methods and we discuss whether or not the results are reasonable. We specifically choose variations that are simple, but give us an important pitfall of the connectivity approach. We had no *apriori* knowledge that a variation like this would fail, and it came up in the testing, but instead of presenting all the variations that work, we offer the simple case that fails. In this variation, first we have a grid with no obstacles, then we put an obstacle at $(1, 1)$. After this, for the next grid we consider an obstacle with cells $(1, 1), (1, 2), (2, 1), (2, 2)$. Note that we can describe this as a block with diagonal elements $(1, 1), (2, 2)$. We proceed similarly to add a block with diagonal elements $(1, 1), (2, 2), (3, 3)$, then continue until we have an embedded a block of obstacles whose diagonal is $(1, 1), \cdots, (N - 2, N - 2)$. The agent starts at $(0, 0)$ and attempts to find a way to the cell $(N - 1, N - 1)$. Figure 8 shows subplots that illustrate this process for $N = 10$.

We obtain figure 6 when plotting the average time it takes to hit the terminal state (the hitting time) against the sequence of variations seen above in the same order. Computing the connectivity of the graphs using the *Networkx* library on Python, we obtain figure 7.This is not very surprising, but at the same time shows that although the idea seems practical and attractive, it breaks down quite easily with very simple graph structures, which in this case is the particular variation of grids.

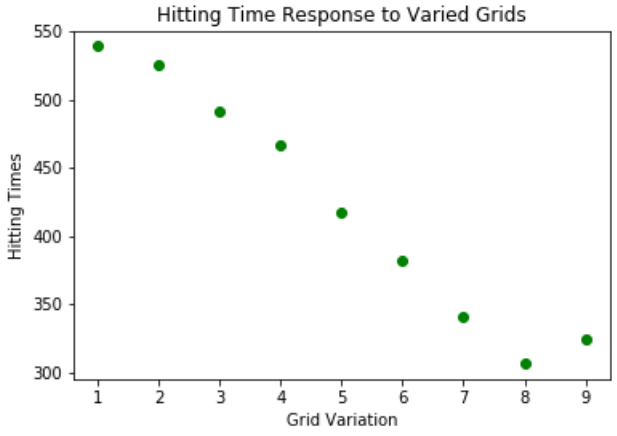


Figure 6: Hitting times with Complexity

It is easy to see that removing the nodes that represent the cells $(0, 1), (0, 1)$, i.e the cells right after the initial cell $(0, 0)$ is sufficient to disconnect the graph structure. We can do this for any of the corners, and this turns out to be the best we can do, and we can't

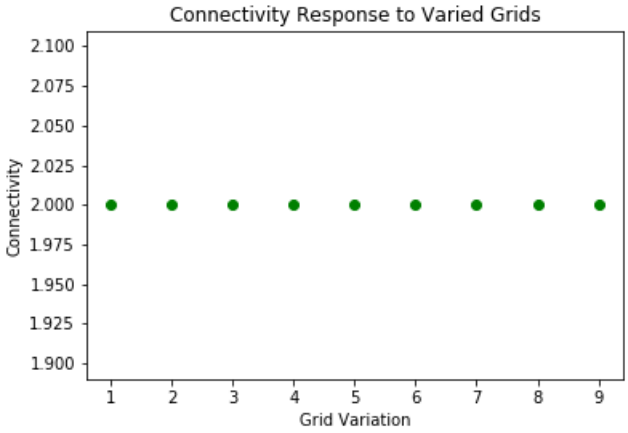
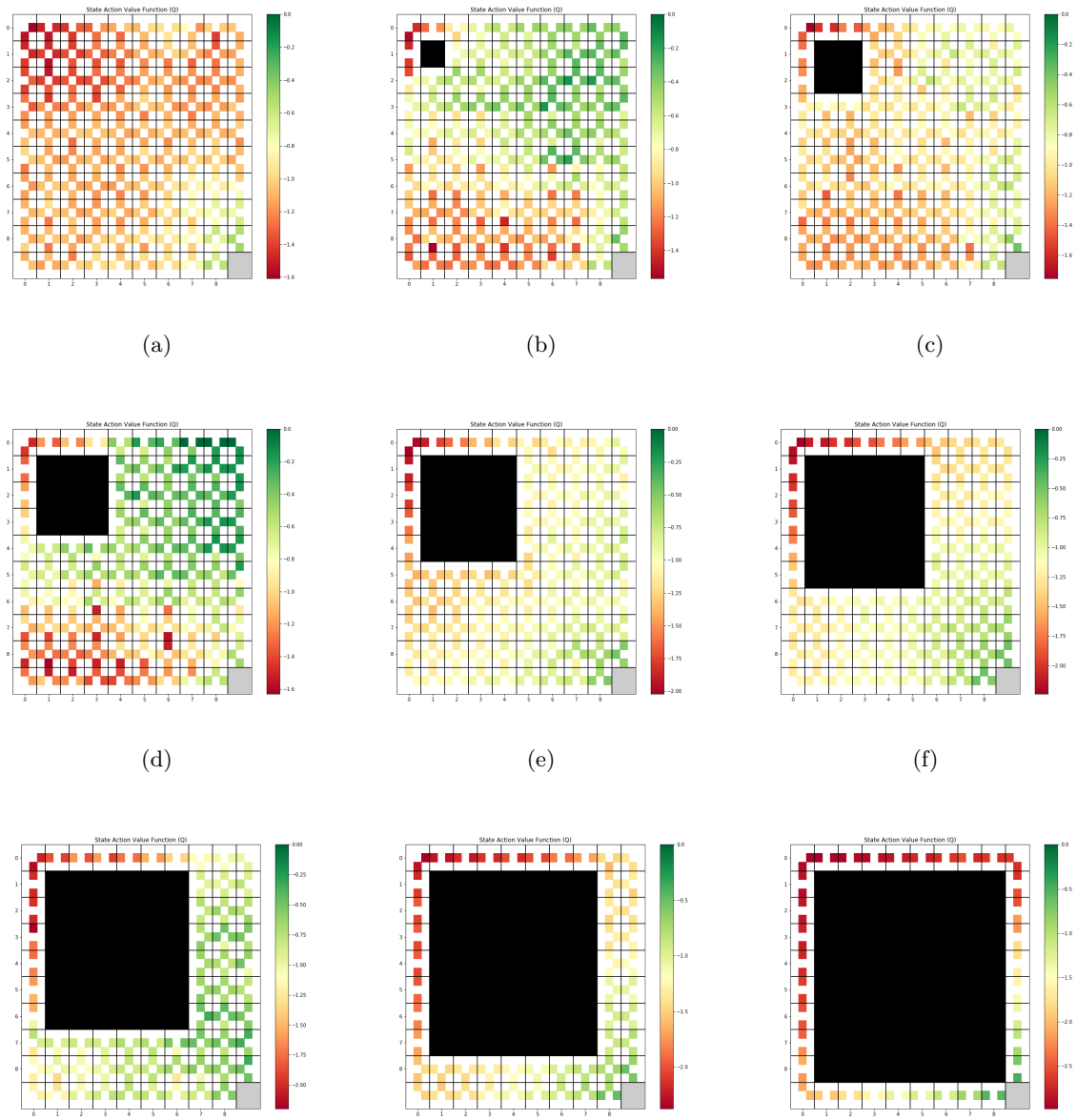


Figure 7: Connectivity with Complexity

disconnect the graph by only removing one vertex. So connectivity does not quite work as a result, because it is not sensitive enough to the deformations we are considering. Hence, although there would have been many exciting things to consider had it worked we have to abandon the idea. There are other alternatives to consider, of course, for instance we could consider the different paths that emanate from the initial cell of interest to the desired terminal state. This should clearly be a more sensitive number to the variations in question than the general connectivity of the graph, but it did not improve the speed in any way, so we do not consider it.

Conclusion

The goal was to impose an order for grid complexity in the context of Reinforcement Learning. We considered two methods for this report that both seemed reasonable–a monte-carlo approach (based on the idea of diffusion) as well as a graph theoretic approach. In the end, we did experiments to see which method works best. The monte-carlo approach works best. It is more sensitive to changes in graph structure. We chose to use the monte-carlo approach in the context of random walks on graphs for the progress of the research project. Hence, the ordering needed is pursued using this method and all analysis, where needed, is done on the context of random walks on graphs.



(a) (b) (c)

(d) (e) (f)

(g) (h) (i)

Figure 8: A Sample of Generated Variations for the Numerical Experiment

Epsilons on Grids

At this stage we would like to discuss how Algorithm 1 was used to compute the optimal epsilon in the Grid World setting. The procedure was as follows.

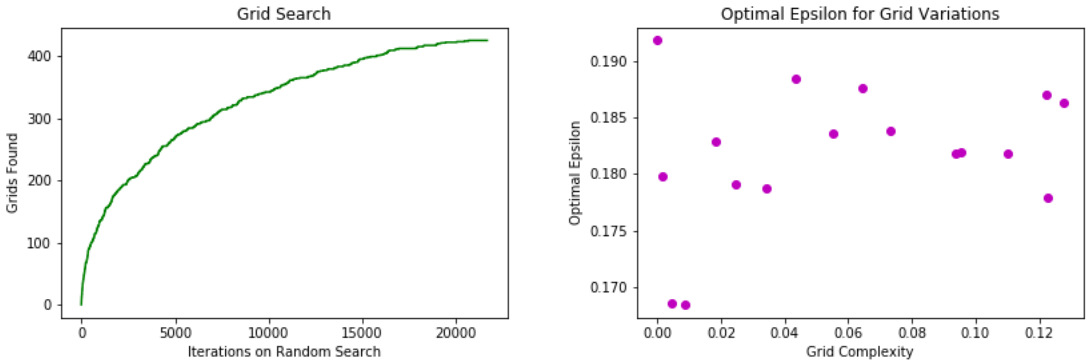
- Generate a random grid and check that it contains the initial state, final state and that the final state is a descendent of the initial state, i.e there is a path between them.
- We applied first step temporal difference learning to a particular grid, then used policy iteration to obtain optimal value functions.
- After this, we ran through different epsilon values according to Algorithm 1, and the update condition was having the final the value functions within some tolerance from the optimal values after a fixed number of computations.
- The epsilons were then updated according to Algorithm 1, fed back into the algorithm, and then this process would repeat for a selected number of iterations.
- After this number of iterations, generate another different grid and continue until some number of grids is achieved or a certain number of iterations is achieved without finding new grids.

Grid Search

Generating grids for $N = 4$ in particular lead to an unreasonably large search space. Instead of attempting to find a closed form for the expected number of variations, which turns out to be a very hard problem, we do a search as hinted above. We run an algorithm to generate random obstacles, discard all graphs where there is no path between nodes of interests, and we do this for as long as we do not get a period where the number of elements we have generated has remained the same for more than 1000 iterations. This is not a very efficient bound, but it is slightly reassuring in its magnitude though it is also true that we might be missing some samples. Figure 9 (a) shows the number of grids as a function of iterations

Epsilons on Ordered Grid Variations

Figure 9 (b) shows the variations arranged in ascending order as done before, with the optimal epsilon plotted accordingly for each grid variations. The complexity axis is normalised so that the sum of the discrete values is 1. It is interesting to note that there is a discrete spectrum of complexities for a given graph size. The worst and best cases can be thought of as in the previous short investigation on grid ordering.



(a) (b)

Figure 9: Grid Search and Optimal Epsilons on Grid Variations

Bounds on Agent Experiences

The most extreme behaviour is observed when $\epsilon = 1$ which is the random walk. The other experiences are less erratic, and we can in fact think of the mentioned case as an upper bound of behaviours, so that if we understand it then everything that falls beneath $\epsilon = 1$ will also be understood to some degree. At the very least we will have quantified measures that bound the extreme cases, and this is the goal of this subsection. Consider the following graph.

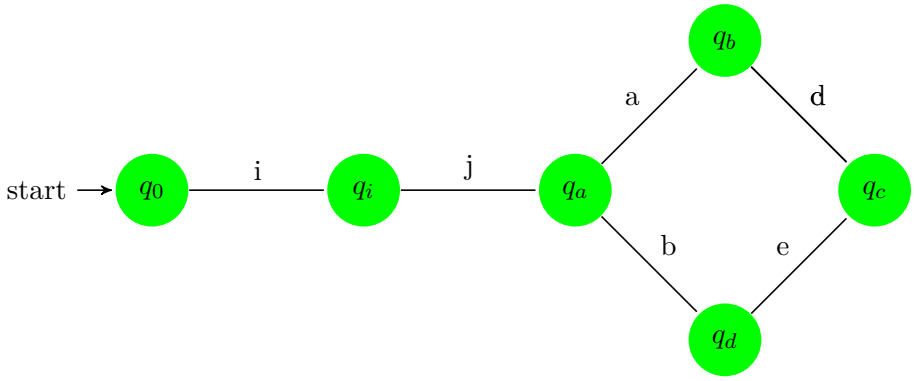


Figure 10: State Diagram, G

Observe that this graph has an initial state q_0 which is then followed by states q_a, q_b, q_c, q_d. It is interesting to compare the expected time it would take a random walk to reach some nodes from selected starting points. This is known as the hitting time from the initial state to the terminal state, i.e if i, j are nodes then the access time or hitting time from $i \to j$ is the expected number of steps before node j is visited, starting from node i [13].

For example, does it take the same time for a random walk to move from $q_0 \to q_a$ as it takes to move from $q_a \to q_e$? It turns out this is not true. The paper above has general

results together with [11], and we present a few special cases. In what follows, $G\backslash\{s\}$ is G with the state s removed together with edges that are incident on the state.

Graph Types	Hitting Times
$G\backslash\{q_b, q_d, q_e\}$	The hitting time from $q_0 \to q_a$ is 3^2
$G\backslash\{q_0, q_i\}$	The hitting time from $q_a \to q_c$ is $2(4-2)$
$G\backslash\{q_0, q_c\}$	The hitting time from $q_i \to q_d$ is $1+2(3)-1$
$G\backslash\{q_0, q_d, q_c\}$	The hitting time from $q_a \to q_b$ is 3^2-1^2

It is relatively easy to see that for any pair of nodes we consider it is possible to find the hitting time by considering a linear combination of the tabulated results. This gives us some understanding of the structures that lead to a greater hitting time as the agent traverses from the initial state to the terminal state. This links very closely to behaviour we already observed in the investigation of the grid ordering methods, but we have to be careful about how we model behaviour along a bigger graph. Roughly speaking, there are two points to take home from this:

- Cycles make learning easier, i.e we expect less time for the agent to explore all the space.
- Numerous smaller cycles are preferred to a few big cycles.
- Hitting times on trees structures are often bounded below by hitting times on cycles and above by hitting times on line graphs.

Comments Epsilon Behaviour

It is apparent from figure 9 that the epsilon value that is optimal is nearly constant for all grid variations given that they lie in a very narrow band of the unit interval. This might be surprising to some, and although we developed no rigorous argument as to why this should be the case, when one considers what's actually happening it seems plausible that this is the case. This behaviour is because we put obstacles along paths, which is not equivalent to putting cliffs. That is, in generating the variations we merely sever some paths as well as possible states by restricting access to these, which is different from allowing the agent to have access to these then imposing heavy penalties for moving into such states. The former allows exploration, but discourages a lot of it, while the latter can either weaken or enhance this effect depending on the context. Thus the latter would most likely have more influence in discouraging exploration than that former, and so it makes sense that all grids have very close reasonable values. This is one of the things we would like to make more precise, or at least study numerically, in the future.

Ideas on Generalising the Methods

Suppose now we ask more ambitious questions about the structure of learning elements in general, i.e elements of a learning architecture. Suppose for instance we have a neural

network, and we wish to use dropout [24]–for say a recurrent neural network [23]. Generally, implementations usually use some ad-hoc value and hope it works. Otherwise, a brute force comparison technique would suffice. This is a very tedius task, and we find that with some contextual modifications we achieve a method to find an approximation of the efficient dropout percentage.

Generally, we adopt the frame work of the first section wherein we discussed the assumptions on solvability of problems as well as the definition of learning algorithms. Then we require instead of an epsilon we consider some learning element G. The assumption is that this is related to the structure of the network. We suppose that there is a set Ω of relevant variations of G, or possible forms. For instance, $\Omega = [0, 1]$ in the case for epsilon. We hypothesize that the rest of the results we had are then implied from this, but we shall not spend much time on this. It is also clear that for parameter searching the method is competitive, since it has the random search property which has been numerically show to be the best method for finding parameters efficiently.

Introduction to Neural Networks

Artificial Neural Networks, usually just referred to as Neural Networks when the context is clear, can be thought as computational models whose working mechanism is largely inspired by the human and animal neurophysiology, i.e how our brains are built, and how they process information. The human brain has billions of neurons connected together by synapses. If sufficient synaptic inputs fire to a neuron, that neuron will also fire. This process is often called "thinking". With artificial neural networks, we emulate this process by creating a neural network on a computer.

Neurons are building blocks of neural network. A neural network has input and output neurons, which are connected by weighted synapses. The weights affect how much of the forward propagation goes through the neural network. We think of the signal incident on the neural as some projection from the input space. For example, the following.

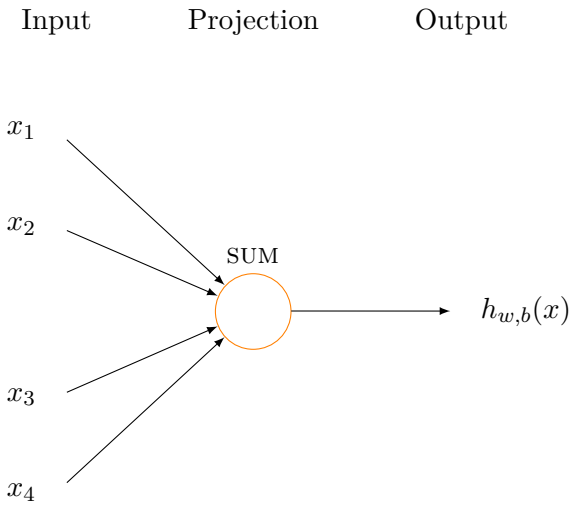
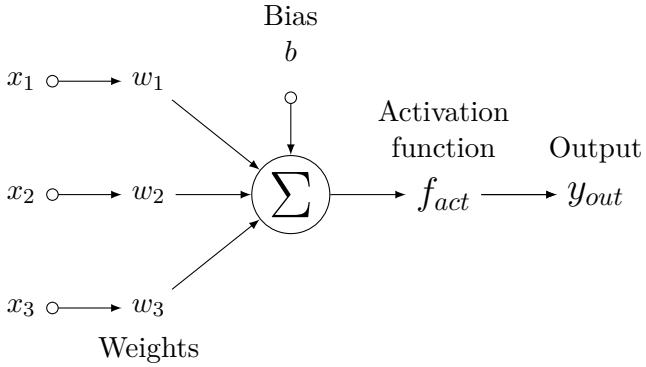


Figure 11: An Example of an Active Neuron

The elements $(x_i)_{i=1}^4$ are inputs to a neuron , the output is $h_{w,b}(x)$ where the b is some associated bias and w denotes the weights. In fact, we can alter this slightly to make it more representative of the bigger picture.



The activation function aids the learning. It turns out that feeding the output of a neural to a well-chosen class of functions that we call activation functions improves learning. The weights can adjusted during a process called back-propagation. During this process the neural network is learning. Backward and forward propagation are done iteratively over the training data set. Adding more neurons in a network improves performance very often. We give a figure to illustrate how back propagation works.

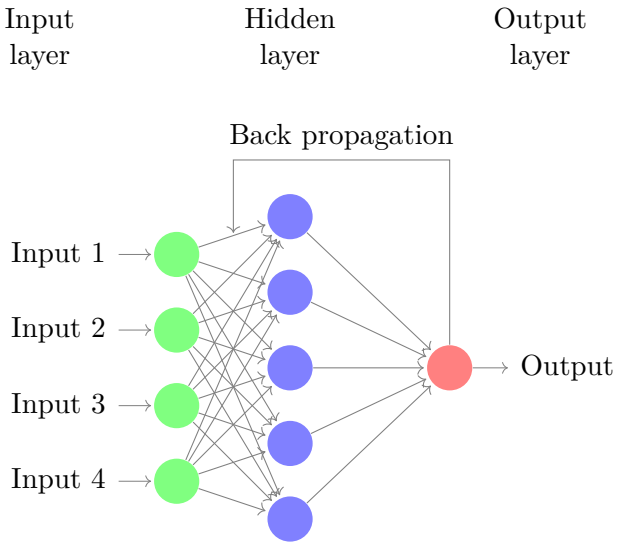


Figure 12: An Example of a Neural Network

When one builds a network, they have initialise weights. Back-propagation is a way of communicating the error that results when the samples on the training set are passed through the network such that the weights are adjusted in a way that 'learns' whenever mistakes are made, for instance if we have a classification problem, then whenever there is a misclassification an error is propagated backwards and the weights are adjusted to acknowledge the mistake.

We can add as many neurons as we want and as many layers as we have computational power for. Adding more layers tends to increase the abstraction of details from training. There are other known methods to aid the learning of neural networks such as drop out, regularisation [21], etc. These are both methods to minimise overfitting for a given network. Models that do not overfit tend to generalise well to unseen testing data, and that is why it is important to avoid, or to at the very least reduce overfitting. Although we will be using the mentioned methods in our architecture, we shall not provide any in-depth explanation of how they work, but the cited resources should be sufficient.

Investigating Transfer Learning

In Gridworld, we have an agent whose task is to move from an initial state s_0 to a terminal state s_T. The agent is expected to find an optimal solution by maximising a numerical reward. Suppose we train a neural network to whenever given a grid, be able to estimate an epsilon. There are a few simplifying assumptions, one of which we simply feed it the data we generate from above. It is possible to have a network that given grid can create a map of the grid including the obstacles, but this is at some resource cost on the scale of what we have already discussed with grid size computations if not more. Instead of requiring that we stop at simply training and observing accuracy, we want to study the transferability of learning in the following context.

Consider a $N \times N$ grid and consider a function $\mathcal{L}\sigma$ that learns some parameter of interest from whatever problem we are attempting to solve, for example ϵ in the discussion above. Suppose we have a variety of structures as we saw above, and for each we want to find this parameter. We can treat this as a classification problem, i.e we consider inputs in $\mathbb{R}^d$ and transformation σ to the feature space $\mathbb{R}^d$ from which we can define another transformation $\mathcal{L}$ to an element of the classification space, so that the objective is learning an optimal way of defining a transformation $\mathcal{L}\sigma : \mathbb{R}^d \to \mathbb{R}$ [6].

Suppose that we have a mapping $\phi : N \times N \to M \times M$ where $N \leq M$, and suppose that ϕ is an isometry, i.e the points in the $N \times N$ grid to a grid of size $(N+1) \times (N+1)$ in a way that preserves distances between the points. We then ask the question: Is there an embedding that will ensure the highest level of transferability of knowledge from training? That is, is there an embedding scheme such that whenever we use the neural network to predict a parameter we obtain a reasonable estimate?

As an example, $S := N \times N$ such that $S_{ij} = (i, j)$ and similarly $S' = N + 1 \times N + 1$. An example of an embedding is

$$\phi_{ij} = \begin{cases} S_{ij} & \text{if } i, j \leq N \\ S'_{ij} & \text{otherwise} \end{cases}$$

Otherwise, we can translate this embedding along the bigger grid to cover all space, i.e consider $\phi_{i(j+1)}, \phi_{(i+1)j}$, etc. It is worth investigating then if it matters how exactly we embed S into S'. This will be the theme of the rest of the project.

In what follows, we introduce a neural network architecture, and then we proceed to report test results after training. After this, we shall put the model to the test in hopes to answer the question about optimal embedding schemes.

We construct a neural network whose inputs are grids, passed through as arrays with zeros except where there is an obstacle, and the outputs are scalars on the unit interval of the real line. We give the structure of the network on the next page.

The Neural Network Architecture

The network was built on Python using *Keras*. The number of layers is kept low together with parameters as can be seen on 13, because the problem at hand does not involve high level detail that for instance facial recognition would. However, we ensure that the structure can handle abstraction, and not merely fit the noise in the training data by choosing the architecture through a process of careful trial and error as well as observation of training statistics.

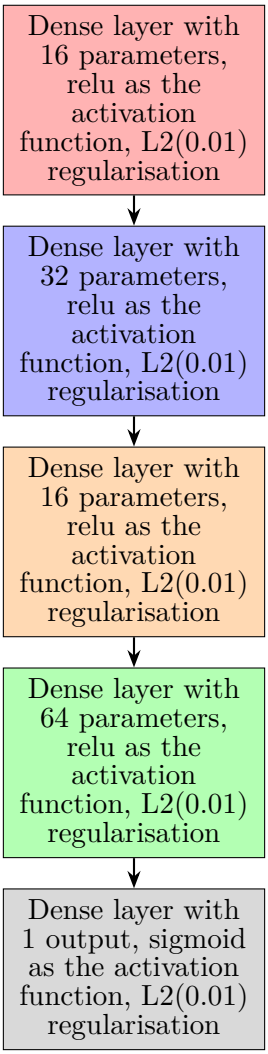


Figure 13: The Neural Network used for the Investigation

We use $dropout(0.2)$ in between all layers, the optimiser is adam, the loss function is based on the mean square error and the metrics are the mean square error as well as the mean absolute error.

Training Statistics

The training data is of the form of what is shown on figure 9 (b). We inject some random noise on the values of epsilon to reduce overfitting on each of values. This increases predicting power of the network by a factor of 10^{-1}. We partition the data we generated into training, validation and testing data sets, and upon training we obtain the loss statistics on figure 14. It is worth noting that usually it is the case that the validation loss is greater than the training loss, but the use of regularisation and dropout can lead to the opposite effect, which is what we observe in the figures below.

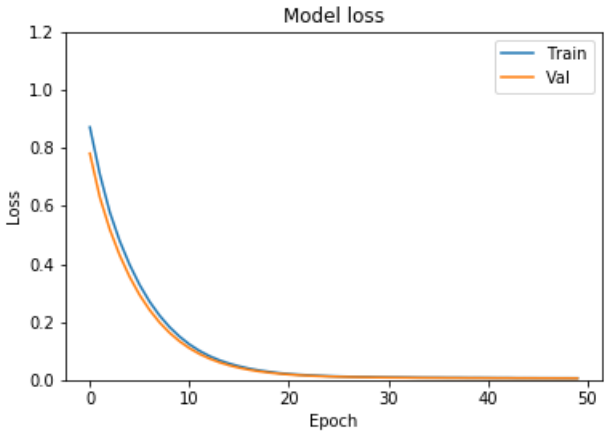


Figure 14: Statistics of the Loss during Training

We tune down the precision to reduce overfitting. We do this by truncating the training after some reasonable accuracy is obtained, and for this we choose 90%. We obtain the following.

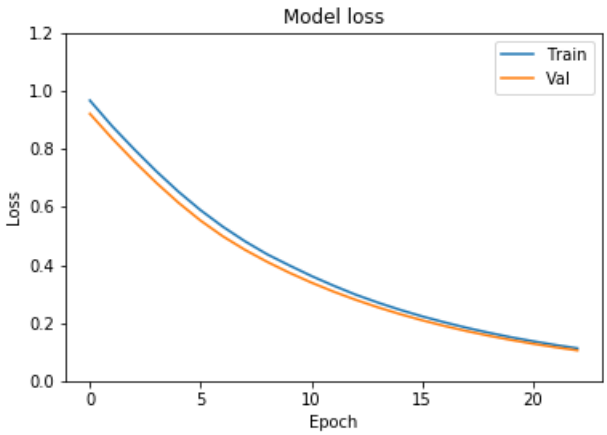


Figure 15: Truncation of Training Time to Reduce Overfitting Effects

Performance Evaluation

Evaluations on known Data

We observe the following behaviour when we predict the values obtained on figure 9 (b).

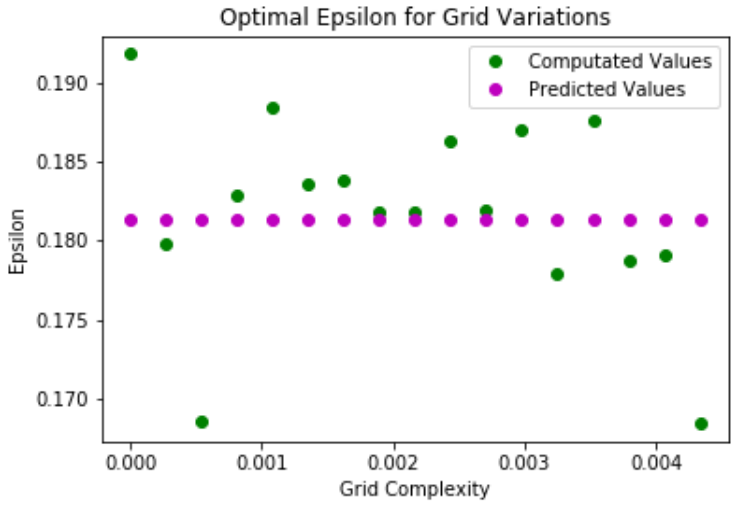


Figure 16: Prediction of Epsilons for the Elements of Figure 8

It seems as if the network finds a value that minimises the error of prediction. Moreover, we find that this strategy for predicting values leads to very low relative errors. However, we could also be seeing patters of overfitting. Figure 17 shows that the magnitude of the relative error of all predictions is bounded above by 0.100, which is a reasonable bound.

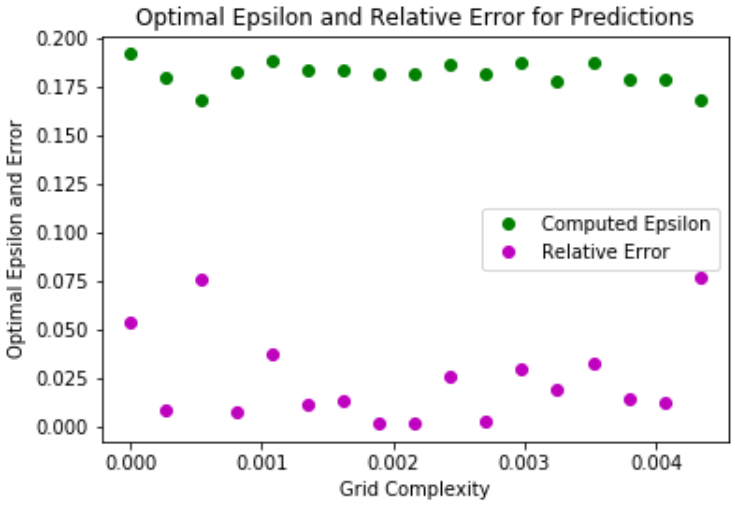


Figure 17: Optimal Epsilon and Relative Error Magnitudes for Predictions

Transfer Learning

Recall the embedding introduced recently, ϕ.

$$\phi_{ij} = \begin{cases} S_{ij} & \text{if } i,j \leq N \\ S'_{ij} & \text{otherwise} \end{cases}$$

We consider cases: $\phi_{ij}, \phi_{(i+1)j}, \phi_{i(j+1)}, \phi_{(i+1)(j+1)}$ and we shall refer to them respectively as the natural embedding, the row translation, the column translation and the row and column embedding for obvious reasons given the notation.

In what follows, the following has been done:

- We select elements randomly from the testing data set.
- We create 4 classes using the isometries by taking each randomly generated element and applying all 4 transformations on the element.
- The ϵ is explicitly computed for each element in all the 4 classes.
- After this, the neural network model is used to predict the ϵ for each of the elements.
- Then we compare the difference between the predicted value as well as the value that is explicitly computed.

We obtain the following results:

Embedding	Mean Error	Uncertainty of Result
Natural Emedding	0.00070	0.00016
Row Translation	0.00069	0.00011
Column Translation	0.00076	0.00015
Row and Column Translation	0.00064	0.00011

Keeping in mind, this is an oversimplification of the outputs. Here there are a few things to note:

- Translating both the row and column beats all of the other translations.
- The column translation gives the worst performance.
- Despite the differences, the error is still quite small, i.e of order 10^{-3}.

Conclusion

Our first goal was to develop a light algorithm that can find an optimal epsilon in the epsilon greedy context. We achieved this goal, and we applied this algorithm to Grid World wherein we found epsilons for different grid variations. Our second goal was to see how the learning transfers to larger grids for small ones. We successfully built a neural network for this task, and we observed how the error compares for different samples, as well as how the error compares given different ways of embedding a small grid into a larger one. We find that we can, to a reasonable degree, transfer learning done on smaller grids to larger ones, and that it does not matter how exactly we do this act of embedding.

References

[1] Sutton, R. S., McAllester, D., Singh, S. and Mansour, Y. *Policy Gradient Methods for Reinforcement Learning with Function Approximation.* In *Advances in Neural Information Processing Systems* **12**, MIT Press (2000), 1057-1063.

[2] Lin, M., Chen, Q. and Yan, S. *Network in Network* `arXiv:1312.4400v3 [cs.NE]`

[3] Andrychowics, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T. and de Frietas, N. *Learning to learn by gradient descent by gradient descent.* `arXiv:1606.04474v1 [cs.NE]`

[4] Kakade, S. *A Natural Policy Gradient.* In *Advances in Neural Information Processing Systems*, MIT Press (2002), 1057–1063.

[5] Salimans, T., Ho, J., Chen, X. and Sutskever, I. *Evolution Strategies as a Scalable Alternative to Reinforcement Leanring.* `arXiv:1703.0386v1 [stat.ML]`

[6] Dupont, E., Doucet, A. and Teh, W. Y. *Augmented Neural ODEs.* `arXiv:1904.01681v2 [stat.ML]`

[7] Bergstra, J. and Bengio, Y. *Random Search for Hyper-Parameter Optimisation.* Journal of Machine Learning Research **13** (2012), 281-305.

[8] Sutton, R. and Barto, A. *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press, 1998.

[9] Sutton, S. R. *Open Theoretical Questions in Reinforcement Learning.*

In *EuroCOLT '99 Proceedings of the 4th European Conference on Computational Learning Theory* (1999), 11-17.

[10] Iwashita, H., Nakazawa, Y., Kawahara, J., Unok, T. and Minato, S.

Efficient Computation of the Number of Paths in a Grid Graph with Minimal Perfect Hash Functions. TCS Technical Report, 2013.

[11] Jacobsen, M. *Exit times for a class of random walk: Exact Distribution results.* In *Journal of Applied Probability*, 2011.

[12] Davey, B. A. and Priestley, H. A. *Introduction to Lattices and Order, 2nd Edition.* Cambridge University Press, New York, 2002.

[13] Lovasz, L. *Random Walks on Graphs: A Survey. Combinatorica*, 1996.

[14] Robbins, H. *A Remark on Stirling's Formula. The American Monthly* **62**, 1995.

[15] Rudin, W. Principles of Mathematical Analysis, 3rd Edition. McGraw-Hll, Inc., United States of America, 1976.

[16] Patrascu, R. and Stacey, D. *Adaptive exploration in reinforcement learning.* In *International Joint Conference on Neural Networks* **4** (1999), 2276-2281.

[17] Tokic, M. *Adaptive ϵ-greedy Exploration in Reinforcement Learning Based on Value Differences.* In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence* **KI'10**, Springer-Verlag, Berlin, Heidelberg (2010), 203-210.

[18] Dos Santos, M. A., da Rocha, R.L.A. *An Adaptive Implementation of ϵ-Greedy in Reinforcement Learning.* In *Procedia Comput. Sci.* **109** (2017), 1146–1151.

[19] Schapire, Robert, E. and Warmuth, Manfred, K. *On the Worst-Case Analysis of Temporal-Difference Learning Algorithms.* DOI:10.1007/978 − 0 − 585 − 33656 − 5_6, (1996).

[20] Dayan, P. and Sejnowski, T. *TD(λ) Converges with Probability 1.* Machine Learning - ML **14** DOI:10.1007/BF00993978 (1994), 295-301.

[21] Mianjy, P., Arora, R., and Vidal, R. *On the Implicit Bias of Dropout.* arXiv:1806.09777v1 [cs.LG], 2018.

[22] Gordon, G. J. *Stable function approximation in dynamic programming.* In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning* (1995), 261-268 San Francisco, CA. Morgan Kaufmann.

[23] LeCun, Y., Bengio, Y. and Hinton, G. *Deep learning.* Review, DOI:10.1028/nature14539.

[24] Hinton, G. E., Srivastava, N., Krizhevski, I., Sutskever, I. and Salakhutdinov, R. R. *Improving neural networks by preventing co-adaptation of feature detectors.* arXiv:1207.0580v1 [cs.NE].

[25] Bengio, Y. *Learning deep architectures for AI.* Foundations and Trends in Machine Learning, (2009).

[26] Han, X. *A Mathematical Introduction to Reinforcement Learning.* https://cims.nyu.edu/ donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf.

[27] Géron, A. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* O'Reilly Media, (2017).

[28] van Seijen, H., van Hasselt, H.P., Whiteson, S. and Wiering, M.A. *A theoretical and empirical analysis of Expected Sarsa.* In *Proceedings of the IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning* (2009), 177–184.

[29] Singh, S., Jaakkola, T., Littman, M. L. and Szepesvari, C. *Convergence results for single-step on-policy reinforcement-learning algorithms. Tech. rep., University of Colorado, Department of Computer Science, Boulder, CO* (1998).

[30] Robbins, H. *A Remark on Stirling's Formula.* The American Monthly **62** (1955), 26-29.

Appendix A: Formal Concept Analysis

In the entirety of this document, we talk loosely about abstract problems and we put an order on the space of solutions of problems of interest. In fact, there is a rich theory that forms the basis of this approach as seen in [12]. This basis of this theory is essentially the notion of context and concepts with order theory. In this part of the appendix, we shall only give an exposition and it should be clear how this can be used to further formalise, as well as generalise, the whole theory we have seen.

Definition 4 (Context) *A context is a triple (G,M,I) where G,M are sets and* $I \subseteq G \times M$. *The elements of G and M are objects and attributes respectively.*

We write gIm instead of $(g, m) \in I$ and say the object g has attribute m. For $A \subseteq G$, $B \subseteq M$, define

$$A' = \{m \in M | (\forall g \in A) gIm\}$$

$$B' = \{g \in G | (\forall m \in B) gIm\}$$

The concept of the context (G, M, I) is defined to be a part (A, B) where $A \subseteq G$, $B \subseteq M$, $A' = B$, $B' = A$. The extent of the concept (A, B) is A while its intend is B.

The maps

$$\phi : A \to A'$$

$$\psi : B \to B'$$

are called polars of the relation $I \subseteq G \times M$. The set of all concepts of the context (G, M, I) is denoted $\mathcal{B}(G, M, I)$

Let (G, M, I) be a context and consider $(A_1, B_1), (A_2, B_2) \in \mathcal{B}(G, M, I)$. We write $(A_1, B_1) \leq (A_2, B_2)$ if $A_1 \subseteq A_2$, and this implies $A_1' \supseteq A_2'$ and the reverse implication is valid since $A_1'' = A_1$, $A_2'' = A_2$, so that we obtain

$$(A_1, B_1) \leq (A_2, B_2) \iff A_1 \subseteq A_2 \iff B_1 \supseteq B_2.$$

It follows then that $\leq$ is an order on $\mathcal{B}(G, M, I)$, and the book establishes that $\langle \mathcal{B}(G, M, I); \leq \rangle$ is a concept lattice and it is complete with the following proposition.

Proposition 4 *Let (G,M,I) be a context. Then* $\langle \mathcal{B}(G, M, I); \leq \rangle$ *is a complete lattice in which the join and meet are given by*

$$\bigvee_{j \in J} (A_j, B_j) = \left((\bigcup_{j \in J} A_j)'', \underset{j \in J}{B_j} \right)$$

$$\bigwedge_{j \in J} (A_j, B_j) = \left(\underset{j \in J}{A_j}, (\bigcup_{j \in J} B_j)'' \right)$$

It is also true that any complete lattice L is isomorphic to a concept lattice $\mathcal{B}(L, L; \leq)$. These results collectively form what is known as the fundamental theorem of concept lattices. The book further discusses a few practical examples of this theory.

Appendix B: Approximate Bounds on Computational Complexity

Analytic Derivations

We wish to study the behaviour of $\frac{\Omega}{2n} = \binom{2n}{n}$ as we saw it earlier, but instead of using a Taylor Series expansion for functions in the forms of integrals –which would be a much more messy route– we shall use Stirling's approximations,

$$\ln n! = n \ln n - n + O(lnn).$$

Another form of this is,

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

One of the proofs for these approximations can be found on [30].

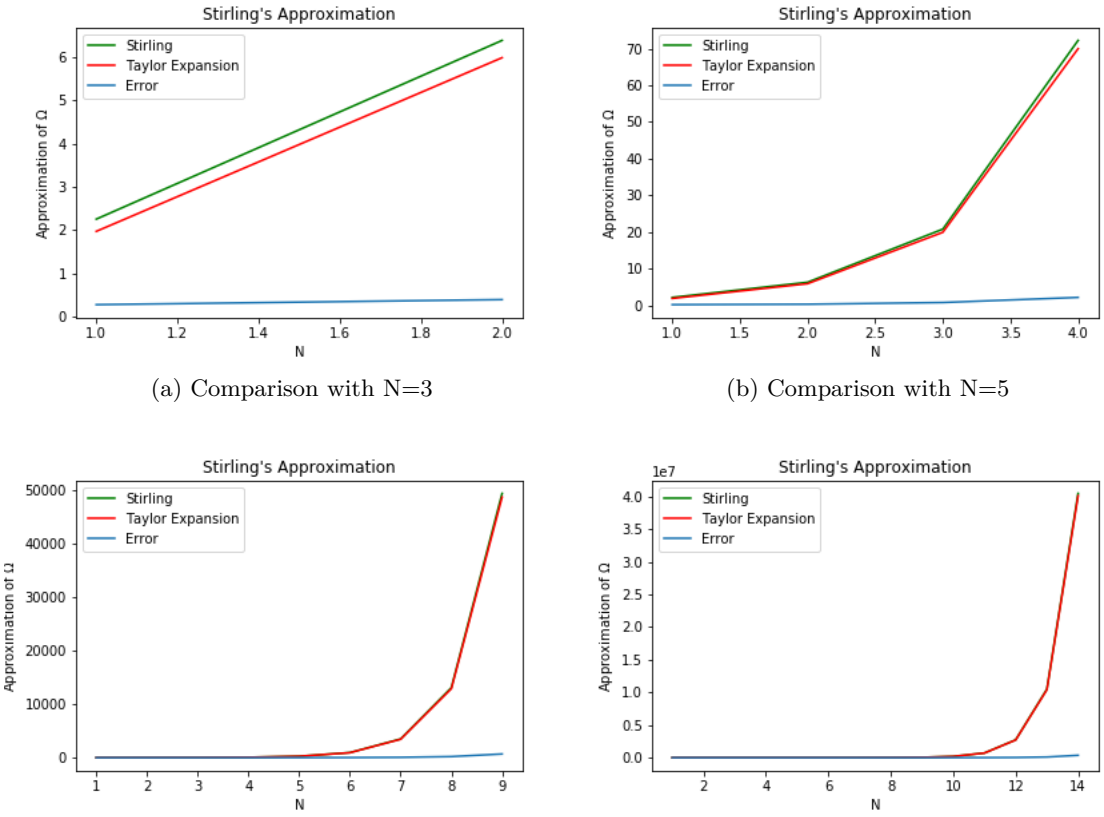
$$\begin{aligned}
\binom{2n}{n} &= \frac{2n!}{(n!)^2} \\
&\sim \frac{\sqrt{2\pi(2n)}\left(\frac{2n}{e}\right)^{2n}}{2\pi n\left(\frac{n}{e}\right)^{2n}} \\
&\sim \frac{1}{\sqrt{\pi n}} \times \left(\frac{2n}{e}\right)^{2n} \times \left(\frac{e}{n}\right)^{2n} \\
&\sim \frac{1}{\sqrt{\pi n}} \times \left(\frac{2n}{n}\right)^{2n} \\
&\sim \frac{1}{\sqrt{\pi n}} 2^{2n} \\
&\sim \frac{2^{2n}}{\sqrt{\pi n}}
\end{aligned} \tag{8}$$

It is easy to see the resemblence between this approximation as well as the one that was

stated earlier, i.e

$$\begin{aligned}\frac{\Omega}{2n} = \binom{2n}{n} &= \left(\frac{\sqrt{1/n}}{\sqrt{\pi}} - \frac{\left(1/n\right)^{\frac{3}{2}}}{8\sqrt{\pi}} + O\left(\left(\frac{1}{n}\right)^2\right)\right)\exp\left(n\log(4) + O\left(\left(\frac{1}{n}\right)^3\right)\right) \\ &\sim \frac{1 - 1/8n}{\sqrt{\pi n}}\exp(n\log(4)) \\ &\sim \frac{1 - 1/n}{\sqrt{\pi n}}\exp(\log 4^n) \\ &\sim \frac{2^{2n}}{\sqrt{\pi n}}\end{aligned} \tag{9}$$

Numerical Tests



(a) Comparison with N=3

(b) Comparison with N=5

(c) Comparison with N=10

(d) Comparion with N=15

Figure 18: Comparison of Approximations

It is clear that the approximations show differences for low values of N but this difference is rapidly reduced for values $N \geq 5$, from which point the two methods essentially generate

neighbouring sequences in the limit as $n \to \infty$. Hence although we introduced an asymptotic expression on a trust basis earlier we have now derived its asymptotic behaviour (validity) from known results. The plots visually demonstrate the convergence of the two forms mentioned as well as the error.

Appendix C: Implementation

For implementation, we considered the grid world first using SARSA as well as policy iteration. The theory coming up as well as techniques on implementation would transfer easily with algorithms like Q-Learning, generalised Temporal Difference Learning TD(λ), etc. It is well known that SARSA converges from [20] and [29]. The author of [29] uses the following Lemma 1 to prove the convergence, and the lemma often becomes exceedingly useful for implementations where the numerical accuracy becomes a hurdle, in which case one should consider both the convergence to a near optimal solution of the algorithm at a particular time as well as the speed with which this is achieved. There are other interesting papers on convergence such as [22]. At this point we wish to state the lemma from [29] and argue the reasonableness of constraints on convergence rates. We state the lemma as is found on [28].

Lemma 1 *Consider a stochastic process* (ζ_t, Δ_t, F_t) *where* $\zeta_t, \Delta_t, F_t : X \to \mathbb{R}$ *such that*

$$\Delta_{t+1}(x_t) = (1 - \zeta_t(x_t))\Delta_t(x_t) + \zeta_t(x_t)F_t(x_t)$$

where $x_t \in X$ *and* $t = 0, 1, 2, \cdots$. *Let* P_t *be a sequence of increasing* σ*-fields such that* ζ_0 *and* Δ_0 *are* P_0*-measurable and* ζ_t,Δ_t *are* P_t*-measurable,* $t \geq 1$. *Assume that the following hold:*

1) the set X *is finite,*

2) $\zeta_t(x_t) \in [0, 1]$, $\sum_t \zeta_t(x_t)$, $\sum_t (\zeta_t(x_t))^2 < \infty$ *with probability* 1 *and for all* $x \neq x_t$ $\zeta_t(x) = 0$,

3) $||E\{F_t|P_t\}|| \leq \kappa||\Delta_t|| + c_t$ *with* $\kappa \in [0, 1)$ *and* c_t *converging with probability* 1,

4) $Var\{F_t|P_t\} \leq K(1 + \kappa||\Delta_t||)^2$, *where* K *is some constant,*

with $||\cdot||$ *being the maximum norm. Then* Δ_t *converges to zero with probability one.*

The paper further elaborates on the idea of the using the lemma following that if we apply it to the case where $X = S \times A$, $P_t\{Q_0, s_0, a_0, r_0, s_1, a_1, \cdots, s_t, a_t\}$, $x_t = (s_t, s_t)$, $\zeta(x_t) = \alpha_t(s_t, a_t)$ and $\Delta_t(x_t) = Q_t(s_t, a_t) - Q^*(s_t, a_t)$. If Δ_t converges with probability one, then we obtain the convergence of Q to an optimal value. It follows that $||\Delta_t|| = \max_s \max_a |Q_t(s, a) - Q^*(s, a)|$. It is clear then that the convergence of rate of Δ_t to zero is the rate at which Q converges to the optimal value, because

$$\frac{d||\Delta_t||}{d\tau} = \frac{d}{d\tau}\left(\max_s \max_a |Q_t(s, a) - Q^*(s, a)|\right)$$

using τ as a time unit. Given that the function approximations are quite close to each other on some defined metric then it makes sense to consider the rate of convergence of each approximation. This is clearly related to the number of function evaluations in the algorithm that solves the original problem, which in our case is SARSA. It follows then that an optimal epsilon is one for which we get the fastest convergence of the algorithm, and the epsilon perturbations are done consider the rate of convergence instead.

Lastly, although Proposition 3 ensures us that we do not have to worry about the convergence of SARSA given different epsilons, a more rigorous approach to this problem can be solved using inpsiration from [Policy Grad Methods for RL with function Approx R, Sutton et al]. The following is the main theorem on the paper.

Theorem 4 *Let π and f_w be any differentiable function approximators for the policy and value function respectively, where w is a parametrization of f, such that*

$$\frac{\partial f_w(s,a)}{\partial w} = \frac{\partial \pi(s,a)}{\partial \theta}\frac{1}{\pi(s,a)}$$

and for which $\max_{\theta,s,a,i,j} |\frac{\partial^2 \pi(s,a)}{\partial_i \partial_j}| < B < \infty$. *Let $(\alpha_k)_{k\in\mathbb{N}}$ be any step-size sequence such that $lim_{k\to\infty}\alpha_k = 0$ and $\sum_k \alpha_k = \infty$. Then for any Markov Decision Process with boound rewards, the sequence $(\rho(\pi_k))_{k\in\mathbb{N}}$, defined by any θ_0, $\pi_k = \pi(\cdot,\cdot,\theta_k)$, and*

$$w_k = w \text{ such that } \sum_s d^{\pi_k}(s)\sum_a \pi_k(s,a)[Q^{\pi_k}(s,a) - f_w(s,a)]\frac{\partial f_w(s,a)}{\partial w}$$

$$\theta_{k+1} = \theta_k + \alpha_k \sum_s d^{\pi_k}(s)\sum_a \frac{\partial \pi_k(s,a)}{\partial \theta} f_{w_k}(s,a),$$

converges such that $lim_{k\to\infty}\frac{\rho(\pi_k)}{\partial\theta} = 0$

Using this it can be shown, but this requires a bit more work, that we do not have to worry about convergence given a varying range of epsilons. In particular, approximating the function f_w appropriately and showing that this approximation is reasonable in the context of the paper. One could then use proposition 2 with a well-chosen approximation. With that done, the proof follows falls out from the theorem.