Applied Maths III projects
2019

No Plagiarism Declaration

(To be handed in with final Project Report)

Please note that without this declaration, your project will not be marked.

Declaration by student:

- I know that plagiarism is wrong. Plagiarism is to use another's work and to pretend that it is one's own.
- Each significant contribution to, and quotation in, this project from the work of other people has been attributed, and has been cited and referenced.
- This report is my own work.
- I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Name: Jeremy du Plessis

Student number: DPLJER001

Signature: [illegible]

Date: 04/10/19

A Survey of Curiosity-Driven Reinforcement Learning

Jeremy du Plessis
DPLJER001
University of Cape Town
Department of Mathematics & Applied Mathematics
MAM3040W

Abstract

Reinforcement Learning is the sub-field of machine learning concerned with the study of learning algorithms which enable decision making agents to learn an optimal mappings from states to actions inside an environment in order to maximise a reward signal over time. In many real life scenarios the environment may be reward-sparse and thus in order for an agent to learn effectively, the learning algorithm should incorporate a mechanism which produces implicit rewards for exploratory behaviour. Although some theoretical foundations had been laid for such algorithms in the 1990's and early 2000's, significant breakthroughs in application have been experienced in the past five years. This report explores the development of curiosity-driven reinforcement learning and examines the theory and application of the most successful algorithms to date.

1 Introduction

Reinforcement learning is one of the three main subsets of the broad field of machine learning, along with Supervised and Unsupervised learning. A Reinforcement learning problem, abstractly speaking, is one that involves a decision making agent that exists within an environment. In general, we imagine that both the agent and the environment exist in a time continuum (as is the case with animals or autonomous robots in the real world). However, in order to frame the problem in a way that is mathematically tractable we think of the agent and the environment as stepping through time discretely. At each time step the environment is in some well-defined state, the agent observes the state and then takes an action in the environment, where after the environment responds in two ways; first, by changing or maintaining it's state and second, by producing a numerical reward signal, which is communicated to the agent, and may be positive (good), negative (bad) or zero. The goal of the agent in all reinforcement learning problems is to maximise the reward signal from the environment over time. A key point to note is that the agent is not explicitly aware of the dynamics of the environment and thus it does not have any inherent knowledge about which action, or sequence of actions, will produce the highest possible reward. Therefore the ultimate objective of all learning algorithms developed for such problems is to enable the agent to learn, through interacting with the environment, a mapping from states to actions that will yield the highest possible reward over time. Reinforcement learning thus distinguishes itself from the other two principle sub-domains of machine learning in the following way. Unsupervised learning algorithms are concerned with finding implicit structure and patterns in data without guidance from the programmer. Supervised learning algorithms rely on labels to explicitly communicate to the computer whether or not a classification is correct; we call this kind of learning instructive, since the computer is being told explicitly what the correct classification is. Reinforcement learning algorithms on the other hand should be thought of as evaluative, in that the decisions of the computer are not judged explicitly as 'right' or 'wrong', but rather numerically evaluated by a single number, with no further information about whether or not any alternative decision may have produced a higher or lower evaluation.

The field of Reinforcement learning originated as a result of two distinct areas of research: the study of optimal control in Engineering, and the study of animal learning in Psychology [1]. The study of optimal control is concerned with designing controllers for dynamical systems in order to maximise or minimise a certain measure of the systems behaviour. The field provided some of the core mathematical ideas used in modern reinforcement learning, including the notion of "optimal return functions", or value functions, and the Markov decision process, a mathematical object which defines the discrete stochastic optimal control problem [1]. At a high level, however, reinforcement learning problems are viewed mainly through the lens of animal learning; the fundamental concepts of a decision making agent capable of learning through positive and negative reinforcement are concepts that have deep roots in the field of psychology. Modelling learning algorithms on biological systems (e.g. neural networks and the brain) is therefore a natural strategy for success, and a further inspiration for augmenting learning algorithms was the idea of synthesising and embedding curiosity into the learning process. There are several definitions of curiosity in the literature, but it can broadly be described as "the desire to seek out novel information in the absence of a specific goal". In the early 1990's, prominent Computer Scientist Jürgen Schmidhuber proposed the idea that a mechanism simulating curiosity could be embedded into reinforcement learning systems in order to produce exploratory behaviour in the absence of explicit rewards. In recent years the development of curiosity-driven algorithms has lead to success on previously unsolvable problems and significantly improved performance in existing learning algorithms. This report aims to summarise the development the curiosity-driven reinforcement learning algorithms and give an overview of the recent developments which have produced the most promising results thus far.

2 PART A: Theory

3 The Full Reinforcement Learning Problem

We will now layout the mathematical definition for the full reinforcement learning problem, which will thereafter allow for exploration of the mathematical and algorithmic solutions to the problem, in particular those where the concept of curiosity plays a major role.

3.1 The Agent-Environment Interface

As outlined above, the reinforcement learning problem consists of a decision making agent which exists within the context of a dynamic environment. At each discrete time step t the environment is in a well defined state

$s_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states for a given environment. The agent observes, either fully or partially (as is most frequently the case), the state of the environment and must then choose an action $a_t \in \mathcal{A}(s)$ to perform, where $\mathcal{A}(s)$ is the set of all possible actions possible in a given state s. In order to decide which action to take the agent consults it's policy $\pi : \mathcal{S} \rightarrow \mathcal{A}(s)$, which is, abstractly speaking a mapping from states to possible actions. Practically speaking, an agents policy may be anything that takes in state information and returns a single action in the case of a deterministic policy, or a distribution over possible actions in the case that the policy is stochastic. The agent then executes the chosen action a_t, after which it will receive feedback from the environment in the form of a numerical reward r_{t+1}, which may be positive (good), negative (bad) or zero, after moving to time step $t+1$. Simultaneously, the environment makes a transition from s_t to s_{t+1}. The state transition may or may not be affected by the agents action in the previous time step. The loop shown in figure 1 summarizes what has been described by Richard Sutton and Andrew Barto in [1] as the agent-environment interface, where we see that the reinforcement learning problem can essentially be distilled down to three signals being passed back and forth between the environment and the agent: state, action and reward.

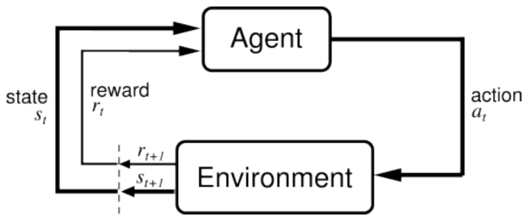

Figure 1: The agent-environment interface [1]

3.2 Rewards & Returns

The purpose of the numerical reward received by the agent at time t is to evaluate the agents action a_t given state s_t. The reward signal is the principle focus of the reinforcement learning problem as defines the primary goal of the agent, which is to maximise the return R_t, which is the cumulative reward over time following t. In mathematically defining the cumulative reward over time we must consider two different categories of tasks in which an agent may engage: the first category is one in which the tasks are finite in time, breaking up into episodes, called episodic tasks. The second category are continuing tasks, which are tasks that, unsurprisingly, continue indefinitely. In the case where the tasks are episodic we have the notion of a terminal state, which is the final state of an episode. In order to maintain mathematical consistency we will force equivalence between the representation of cumulative rewards for both types of tasks by defining the terminal state of an episode not as the end of the episode, but as the state that transitions to itself recursively once it is reached, returning a reward of zero each time [1]. A simple example of a continuing task and an episodic task are illustrated in figures 2 and 3 respectively. In the illustrated examples, the states are represented as nodes and the state transitions as edges, with each transition from a non-terminal state yielding a reward of $+1$.

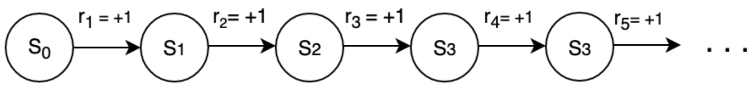

Figure 2: An example of a continuing task

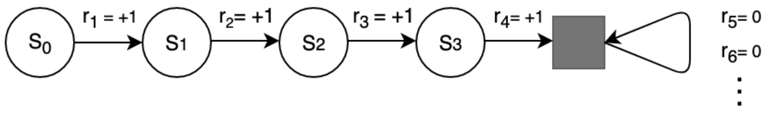

Figure 3: An example of an episodic task

We can now define the return for both continuing and episodic tasks as

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad 0 \leq \gamma < 1, \gamma \in \mathcal{R}$$

where γ is called the discount rate, and ensures that the infinite sum converges to a finite value so long as the reward sequence $\{r_k\}$ is bounded. Say $r \leq M$, $M \in \mathcal{R}$ for all possible values of r, then

$$\begin{aligned} R_t &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \\ &\leq M \sum_{k=0}^{\infty} \gamma^k \\ &= \frac{M}{1-\gamma} \end{aligned}$$

Tuning the discount factor affects how the goal of the agent is defined (and thus how the agent will behave); if $\gamma = 0$ we call the agent "myopic", being concerned only with the immediate reward after each time step, and as $\gamma \rightarrow 1$ the agent becomes more farsighted, concerned less about immediate rewards and more about how future rewards will be affected by current actions.

3.3 The Markov Property & Markov Decision Processes

An important point to emphasise about the relationship between the agent and the environment (in almost all problems of importance in reinforcement learning) is that the agent has no explicit knowledge about the rules that govern the dynamics of the environment. Indeed, if it did we would not have a learning problem, but an optimisation problem as is the case in problems concerning optimal control and the solutions developed in dynamic programming (not covered in this report). Instead, the agent must attempt to learn the best possible actions to take under certain states through sampling - taking actions in the environment, observing its response in terms of states and rewards and adapting its behaviour based off its experience. Since the agents choice of actions is a function only of the state signal produced by the environment, a requirement of just about all modern learning algorithms is the state signal to have (or at least approximate) the Markov Property [1]. Informally this means that a state signal is required which summarizes past sensations in such a way that all relevant information needed to make effective decisions is maintained. Formally, we define the Markov Property by considering how the environment may respond at time $t+1$ to the action taken at time t. In the most general case the response may depend on all states, actions and rewards from previous time steps. We treat the environment as stochastic, and define the dynamics according to the complete probability distribution

$$P\{s_{t+1} = s', r_{t+1} = r | a_t, s_t, r_t, a_{t-1}, s_{t-1}, ..., r_1, a_0, s_0\}$$

which is true for all s' and r and all possible preceding states, actions and rewards $s_t, a_t, r_{t-1}, s_{t-1}, a_{t-1}, ..., r_1, s_0, a_0$. In the case the state signal at time t summarises compactly the past sensations experienced by the agent then the following statement holds

$$P\{s_{t+1} = s', r_{t+1} = r | a_t, s_t, r_t, a_{t-1}, s_{t-1}, ..., r_1, a_0, s_0\} = P\{s_{t+1} = s', r_{t+1} = r | a_t, s_t\}$$

Which means that, given a state signal that is Markov (has the Markov property), one should be able to predict all possible future states and rewards just as well regardless of whether the entire history of events or just the most recent events are known. The majority of modern learning algorithms developed for solving the reinforcement learning problem depend on the environment in question emitting a state signal which has the Markov Property, or at least approximates it. A simple example of a state signal which would not have the Markov property would be one where the state was a snapshot in time of a moving object, in this case we would be missing information about the objects velocity and would be unable to make predictions without knowing the objects history in time and space. We could then construct a Markov signal by creating a state signal out of, for example, the four previous positions of the object in the four most recent time steps, granted small enough time delta.

If a reinforcement learning task consists of a state signal which has the Markov property, we may define it as a Markov Decision Process (MDP). In the discrete case it is called a *finite* Markov Decision Process, and although the methods examined later in the report are designed to handle continuous state spaces, representing a reinforcement learning task as a discrete case is a useful tool for thinking about the problem in general. An MDP is a mathematical structure that may be visualised as a directed graph with nodes and edges. The nodes would represent states and the edges would represent state transitions, each having a transition probability and reward associated with them. No restriction is placed on the value of rewards associated with the transitions in an MDP, however the sum of all outgoing transition probabilities from a given state (node) must sum to 1.

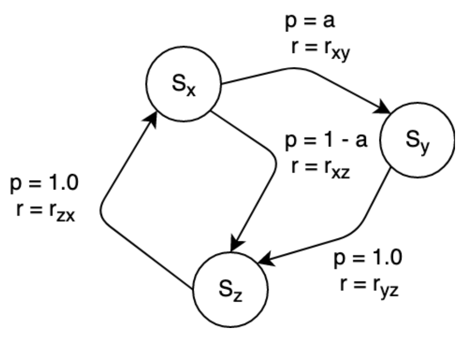

Figure 4: A simple discrete Markov Decision Process

One can then define the dynamics of a finite Markov Decision Process through it's one-step transition probabilities, given the previous state and action. In the same way, one can define the expected reward from a one-step transition given the previous state and action, as well as the current state. The definitions are given below.

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\}$$
$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

In the following section, these ideas are utilised in order to define the idea of a value function, followed by the Bellman equations.

3.4 Value Functions & The Bellman Optimality Equations

When faced with making a decision, given a particular state, one would ideally like to know the expected value of the return to be received following each possible state or state-action pair. Nearly all modern learning methods center around approximating value functions for the task at hand, in order to inform the policy of the agent. There are two types of value functions; one for states $V^{\pi}(s)$:$\mathcal{S} \to \mathbb{R}$ the other for actions $Q^{\pi}(s, a)$:$\mathcal{S} \times \mathcal{A} \to \mathbb{R}$, where the superscript π indicates that the value of states or state-action pairs depends on the state $\to$ action mappings determined by the policy π.

The unique path followed by an agent, from state $\to$ action $\to$ state $\to \ldots$ and so on, through reinforcement learning task is called a *trajectory*. Trajectories may be visualised using so-called back up diagrams, where state nodes (white) and action nodes (black) are connected in a tree-like structure. They can be helpful in developing an intuition for how we compute value functions, an example of one is seen in figure 5.

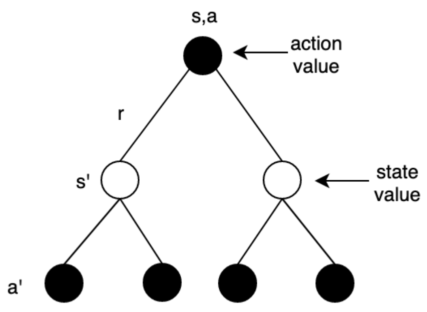

Figure 5: A back up diagram

In order to define the value functions it is necessary to define $\pi(a|s)$ as the probability of selecting action a while in state s under the stochastic policy π (policies are explained in detail below). The formal definitions for the state and action value functions for reinforcement learning tasks structured as MDPs are:

$$\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} && (1)\\
&= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} && (2)\\
&= E_{\pi}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} && (3)\\
&= \sum_{a} \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\}] && (4)\\
&= \sum_{a} \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')] && (5)\\
Q^{\pi}(s, a) &= E_{\pi}\{R_t | s_t = s, a_t = a\} && (6)\\
&= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} && (7)\\
&= E_{\pi}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} && (8)
\end{aligned}$$

$$= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(a'|s') E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s', a_{t+1} = a'\}] \tag{9}$$

$$= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s', a')] \tag{10}$$

where (1) and (6) are the standard definitions for the state and action value functions respectively, and (5) and (10) are their associated bellman equations. The Bellman equations express the relationships between states or state-action pairs and their successor states or state-action pairs, these expressions are used below in the definition of optimal value functions. The true value functions, $V^\pi(s)$ and $Q^\pi(s, a)$, under policy π may not be determined explicitly from finite experience (sampling of trajectories) since neither the state transitions in the MDP (determined by the rules governing the environment) nor the policy need necessarily be deterministic. This means that sampling a finite number of trajectories under π and keeping track of rewards received from states/state-action pairs would only allow one to approximate true value functions. However, sampling an infinite number of trajectories *would* cause the approximation to converge, so sampling a large number of trajectories and following the same process would therefore yield a good approximation of the value functions associated with π [1]. These approximations are notated as $V(s)$ and $Q(s, a)$. The process of learning will ultimately involve approximating the value functions $V^\pi(s)$ and $Q^\pi(s, a)$ through experience then using these approximations to improve the policy by making it greedy with respect to the value functions (explained in detail below); repeating this cycle drives convergence to a policy under which action choices in each state that would be likely to yield the highest return, and hence admit the optimal strategy for a given task.

We must now define the notion of optimal value functions. A policy π is considered superior to a second policy π' if, for all possible states, the following statement holds $V^\pi(s) \geq V^{\pi'}(s)$ (a similar statement would hold true for action value functions). In this way value functions define a partial ordering over policies, and one can therefore define the *optimal state-value function* and the *optimal action-value function* as the value functions which yield the maximum return over all policies π, for all possible states and state-action pairs:

$$V^*(s) = \max_\pi V^\pi(s) \qquad \forall s \in \mathcal{S}$$

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \qquad \forall s \in \mathcal{S}, \quad \forall a \in \mathcal{A}(s)$$

The definition of the optimal value functions allow one to define the Bellman 'optimality' equations:

$$V^\pi(s) = \max_{a \in \mathcal{A}(s)} Q^{\pi^*} \tag{11}$$

$$= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')] \tag{12}$$

$$Q^\pi(s, a) = E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a) | s_t = s, a_t = a\} \tag{13}$$

$$= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^\pi(s', a')] \tag{14}$$

which each admit a system of equations - one for each state or state-action pair. The Bellman optimality equations mathematically define the goal of reinforcement learning tasks - that is to find the value functions which will inform of the action choice that will yield the optimal return. For finite MDPs the Bellman optimally equations (i.e. the system of equations) have a unique solution [1], and if solved would yield the optimal strategy.

Unfortunately, unless all the environment dynamics are known, as is the case with many optimal control problems, the system of equations cannot be solved explicitly. In the case where the environment dynamics are not known one must rely on strategies that involve sampling trajectories and using them to iteratively approximate and improve value functions.

In practice, a value function may take many forms so long as it fulfills the mapping function defined above from states or state-action pairs to estimated returns. For problems where the action and state spaces are discrete it is often enough just to use a table where the row numbers map to states and the column numbers map to actions. In the case where the action and/or state spaces are continuous we need to use something that can approximate a continuous function, such as an artificial neural network (ANN) or or a random forrest model. Figure 6 illustrates examples of both cases. The universal approximation theorem for ANNs (specifically the theorem refers to a single layer perception) assures us that we are able to approximate any function with arbitrary accuracy using an ANN, hence ANNs are at the core of many of the modern reinforcement learning algorithms desinged to solve high-dimensional, continuous state space tasks.

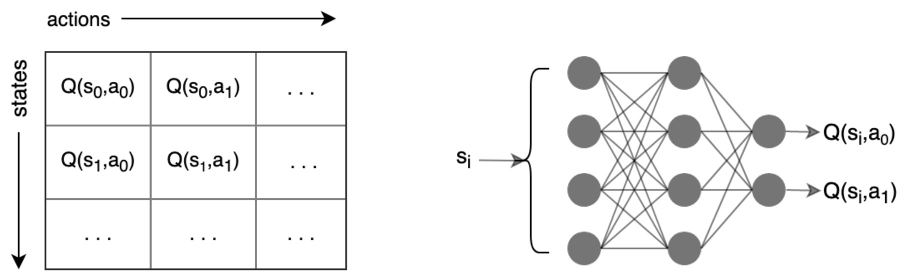

Figure 6: An Action Value function as a table (left) or a neural network (right)

As illustrated in figure 6, an ANN parameterised by $\vec{\theta}$, $f_{\vec{\theta}}$:$\mathbb{R}^n \to \mathbb{R}^m$, serving as an approximator to an action-value function would take as input a vector of real-valued components defining the state space at time t, s_t, and output a vector of real valued components corresponding to the value of possible actions in that state, $\mathcal{A}(s_t)$. Implementations of value functions as neural networks are examined later on in the report.

3.5 Elementary Policies & the Exploration vs. Exploitation Problem

As stated previously, a policy is a set of rules defining a state to action mapping, and although policies are often informed by their associated value functions, they need not be synonymous with them. A class of policies which *are* completely informed by their associated value functions are called *greedy* policies; these are policies under which the chosen action in any given state must be the action a such that $a = \max_a Q^{\pi}(s, a)$, where π is the policy in question. Now, unless one finds oneself in possession of the optimal value functions, this policy will usualy produce very poor returns in the long run as it encourages no exploration of the state space whatsoever, and hence little to no improvement to the value functions will be admitted. A second class of policies are called ϵ-soft policies. These are policies under which action selection in a state s in tantamount to sampling from a multinomial distribution over all possible actions $a \in \mathcal{A}(s)$, along with the condition that $\pi(a|s) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$, for some $\epsilon \geq 0$. In words this means that with probability ϵ a random action may be selected uniformally. When a random action is not selected under an ϵ-soft policy, with a probability $1 - \epsilon$, the action $a \in \mathcal{A}(s)$ such that $a = \max_a Q^{\pi}(s, a)$ is selected. This is a simple mechanism for encouraging exploitative behavior, doing so by ensuring that new trajectories may be experienced by the agent regardless of what the value function deems the most valuable action.

For many modern solutions to the reinforcement learning problem, the type of stochastic action selection described above has been a satisfactory method of encouraging exploration of the state space and has proved successful in several of them [1]. In implementing such solutions, where something like an ϵ-soft policy is used, the question of how big epsilon should be - and thus how often actions are randomly selected - gave rise to one of the core problems in reinforcement learning: the **Exploration Exploitation Problem**, which is the problem of deciding to what degree the agent should explore the state space though stochastic action selection, versus the degree to which the agent should exploit it's current knowledge of the environment in order to generate the highest possible return over time. In a simple benchmark task, called the n-armed bandit task, the agent interacts with a static (single state) environment, and has a choice at each time step of pulling one of n levers. Pulling a single lever returns a value sampled from a Gaussian distribution with a fixed mean and variance of 1. The means of each of the distributions belonging to the n levers are generated by sampling initially from a Gaussian distribution with a mean of 0 and a variance of 1, so each of the distributions belonging to the levers overlap significantly. Given 2000 time steps per episode, the agent selects a lever at each time step according to an ϵ-soft policy, receives a reward sampled from the distribution belonging to the lever and updates the value estimates of the lever selected after each time step. One can show that varying ϵ has a significant effect on the return generated per episode, with the effect becoming more obvious as more episodes are played. ϵ=0 (a greedy policy) performs very poorly, where as ϵ values of 0.01 and 0.1 both improve significantly on the performance. It is also important to observe that performance starts to degenerate for $\epsilon \geq 0.3$ as it prevents the agent from exploiting it's knowledge effectively once it has reasonable estimates for the action value functions. These results can be seen in figure 7.

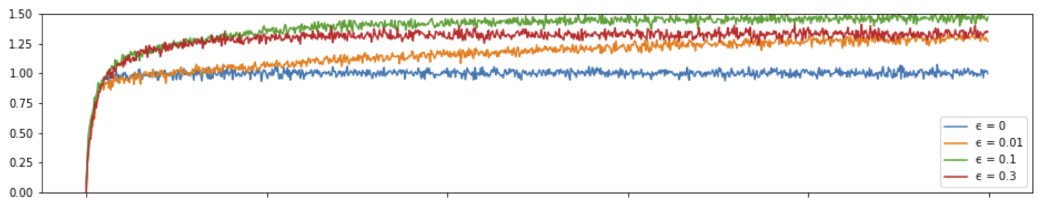

Figure 7: Returns per episode, over 1000 episodes for different ϵ, generated during an n-armed bandit task

In theory, given any MDP and an infinite amount of time in which to train the agent, an ϵ-soft policy with a reasonable value of ϵ should admit convergence to the optimal value functions over time. This is because, given $\pi(s,a) \geq \frac{\epsilon}{|\mathcal{A}(s)|}$ for all $s \in \mathcal{S}$ and all $a \in \mathcal{A}(s)$, all possible trajectories allowed by the environment (i.e. where state transitions are non-zero) have a non-zero probability:

$$P(\tau) = P(s_1, a_1, ..., s_T, a_T) = P(s_1) \prod_{t=1}^{T} \pi(s_t, a_t) \mathcal{P}_{s_t s_{t+1}}^{a_t} > 0.0$$

which essentially means that all trajectories allowed by the environment should be experienced by the agent given infinite time for training. In practice however this does not hold, as nobody has an infinite amount of time in which to perform training and in the finite amount of time available, the compound probabilities for many trajectories are far too small to occur frequently enough for the agent to learn effectively from them, if they occur at all. This problem is not necessarily an obstacle for environments which are "reward-dense", where the agent is likely to receive regular evaluation of it's actions in the form of rewards from the environment. However, several tasks which we would like to solve are "reward-sparse" meaning that, in order to succeed at acheiving the explicit goal of the task, an agent may have to take several consecutive actions randomly without receiving any feedback at all. In such environments, the problem of very small compound probabilities as described above comes into play. In such cases, the effectiveness of stochastic action selection, as in ϵ-soft policies, breaks down. In order to solve these tasks researchers have had to develop mechanisms encouraging more ordered exploratory behaviour by agents in the absence of explicit rewards. It is for this reason that the methods described in Part B of this report have been developed.

What follows is a description of a general framework for solving the reinforcement learning problem, followed by a description of two of the most widely used modern deep reinforcement learning strategies used, where after we will go on to describe how these methods have been augmented to include mechanisms for encouraging exploration in reward-sparse environments.

4 Generalised Policy Iteration

We will now outline a general, abstract framework for solving the reinforcement learning problem. The main algorithm, Generalised Policy Iteration, involves two sub-processes namely Policy Evaluation and Policy Improvement. The algorithm as described below is abstracted away from any notions of implementation, as it may be applied generally in solving reinforcement learning problems regardless of the type of value functions, policy, state space or actions which define any specific problem which we may want to solve.

In general, policy evaluation is the process of approximating the value functions under a policy π. In the case where we know the state transition probabilities for a finite MDP, as is the case with problems of optimal control, the dynamic programming methods involve computing a sequence of approximations for the state value function, initialised with arbitrary values for each state, updated using the Bellman Equation (5) and ultimately converging to the true value function under π: $V_1(s), V_2(s)... \rightarrow V^{\pi}(s)$.

In cases where the transition probabilities are not known, it is then desirable to compute action value functions rather than state value functions since we cannot know what the optimal action choice would be at each time step as we would not be able to compute the expected value over all possible states. In such cases, the action value function may be approximated by sampling trajectories from the environment, using the returns generated during the experience to update the value approximations for state-action pairs. Methods which use only sampled returns generated from experience to iteratively update value functions are called Monte Carlo methods, and methods which use a combination of sampled returns and existing value approximations from successor states - referred to as bootstrapping - are called Temporal Difference (TD) Methods. Where Monte Carlo methods rely on the reinforcement learning task being episodic in nature, TD methods can be used to solve both episodic and continuing tasks. This is because Monte Carlo methods rely on the existence of a terminal state, and update the estimation of a state-action pairs using sampled returns averaged over several episodes. In $n - step$ TD methods the updates to the value estimate for a state-value pair are done by sampling and recording rewards gained for the n-steps following an action, and create an incremental update of the estimate using the sampled rewards in combination with the current estimated value of the state-action pair following the n^{th} reward. In continuing tasks, values may be updated online (while the agent is engaged in the task) using n-step TD methods. In both Monte Carlo and TD methods, the objective during policy evaluation is to begin with an arbitrarily initialised action value function $Q(s,a)$ and, following a policy π, which may simply involve random action selection initially, update the value estimates for each state-action pair encountered, thus approximating the value function belonging to that policy $Q^{\pi}(s,a)$. In order to ensure that all (or sufficiently many) state-action pairs are visited during sampling, some mechanism must be put in place to ensure exploration; in most elementary solutions this simply involves following an ϵ-soft policy.

Following a single round of policy evaluation, we would like to adjust our policy in order to make action-choices which will yield higher returns. In order to do this we need to make our policy greedy with respect to the new, updated estimate of the value function. Meaning that for each $s \in \mathcal{S}$, the preferred action under the policy π is

$$\pi(s) = arg \max_a Q(s,a)$$

As with before, under an ϵ-soft policy, in order to maintain exploitative behaviour one would not select the identified greedy action in a deterministic way, but sample from a multinomial distribution where the probability of selecting the greedy action would be $1-\epsilon$, choosing randomly from all available actions otherwise.

In order to converge on the optimal action value function Q^*, one would continue iterating through cycles of policy evaluation; sampling trajectories from the environment and updating the action value function, and policy improvement; adjusting the policy making it greedy with respect to each new approximation of Q. This process is called *Generalised Policy Iteration* (GPI).

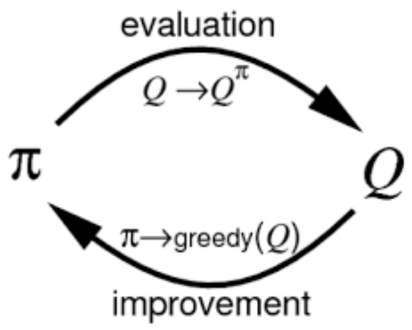

Figure 8: An illustration of Generalised Policy Iteration from [1]

GPI gives us a sequence of action value function approximations and their associated greedy policies, $\pi_0 \rightarrow Q^{\pi_0} \rightarrow \pi_1 \rightarrow Q^{\pi_1} \ldots \pi^* \rightarrow Q^*$, which we can prove will converge on $Q*$ in the following way. Let $\pi_{k+1}(s)$ be the greedy (deterministic) policy with respect to $Q^{\pi_{k+1}}$, then for the previous (deterministic) policy π_k we have that for all $s \in \mathcal{S}$:

$$\begin{aligned} Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, arg \max_a Q^{\pi_k}) \\ &= \max_a Q^{\pi_k}(s,a) \\ &\geq Q^{\pi_k}(s, \pi_k(s)) \\ &= V^{\pi_k}(s) \end{aligned}$$

So each π_k is better than in successor π_{k+1} unless it is equivalent, in which case both policies are the optimal policy, which is a fixed point in the system. This is the proof of the *policy improvement theorem* and it is dependant on who assumptions: first, that we are allowed an infinite number of episodes (in the case of episodic tasks) and, second, that initial state in each episode is selected at random from all possible states, which is called the assumption of "exploring starts", which will not be considered in this report as it is not a common feature of most reinforcement learning tasks we would like to solve. However, the same goal of ensuring that all states are visited can be guaranteed through stochastic action selection or other methods that encourage exploration, which we will explore. With respect to the first assumption, since sampling infinite trajectories is not practically possible, we will at best get asymptotic convergence to Q^*.

5 Modern Reinforcement Learning Algorithms

Two highly-utilised modern algorithms for solving reinforcement learning tasks are described in the section below. Both follow the strategy of GPI in order to converge to optimal action value functions, and both make use of artificial neural networks as function approximators. The first is Monte Carlo Policy Differentiation, which is a Monte Carlo Method, and the second is Deep-Q Learning, which is a TD method.

5.1 Monte Carlo Policy Differentiation

Monte Carlo Policy Differentiation (MCPD) is a method for solving reinforcement learning tasks in which the state space, $\mathcal{S}$, may be defined by a vector of continuous, real-valued variables. The policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$ is governed by a function approximator (e.g. an artificial neural network or a random forest), which is parameterised by $\vec{\theta} \in \mathcal{R}^b$, and gives a distribution of actions over states. We will denote the parameterised policy as $\pi_{\vec{\theta}}(a|s)$, giving the probability of an action a given a state s. In practice, at each time step t the function approximator would take as input a vector $\vec{x} \in \mathcal{R}^m$ defining the state s and output a vector $\vec{y} \in \mathcal{R}^n$ (where $\sum_{i=1}^{K} y_i = 1$) which represents a multinomial distribution over possible actions $a \in \mathcal{A}(s)$. The broad strategy of the algorithm will be to iteratively update the parameters of the function approximator upon which the policy depends in order to converge upon an optimal policy.

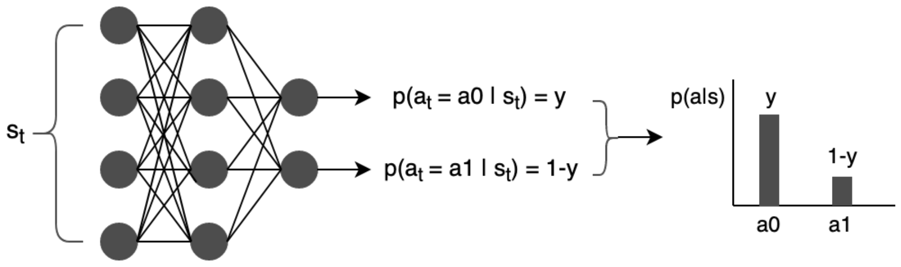

Figure 9: An example of using an ANN to produce a distribution of actions over states

In short, the learning algorithm for MCPD is to have the agent sample several trajectories τ through experience with the environment, where we define a trajectory to be a sequence of consecutive states and actions observed and taken by the agent respectively, during an episode. At each time step of each trajectory, after a single action selection, the gradient $\nabla_{\vec{\theta}} log(\pi_{\vec{\theta}}(a_t|s_t))$ is computed and stored for the chosen action, along with the resultant the reward. This allows one, after $N >> 1$ trajectories have been sampled, to approximately compute the gradient of an objective function $J(\vec{\theta})$, defined as the expected reward under the policy $\pi_{\vec{\theta}}$, by averaging over the product of policy gradients and rewards from all N trajectories. The parameters $\vec{\theta}$ are then updated, using the computed gradient $\nabla_{\vec{\theta}} J(\vec{\theta})$, in the direction of steepest **ascent** which drives the agent to make decisions that yield higher returns. In terms of GPI, we can think of the process of sampling trajectories from the environment and storing the generated returns and their corresponding gradients as the process of policy evaluation, then computing the gradient of the objective function and updating the parameters may be thought of as policy improvement. One would repeat the cycle of evaluation and improvement several times and then, given that the Markov Property holds, expect convergence to an optimal policy. Formally the learning algorithm is defined by repetition of three steps:

1. Sample $\{\tau\}_{i=1}^{N}$ trajectories from the environment using $\pi_{\vec{\theta}}$, storing generated returns and associated gradients
2. Compute $\nabla_{\vec{\theta}} J(\vec{\theta}) = \frac{1}{N}\sum_{i=1}^{N}[\sum_t \nabla_{\vec{\theta}} log(\pi_{\vec{\theta}}(a_t^i|s_t^i))][\sum_t r(a_t^i|s_t^i)]$ where $N >> 1$
3. Update parameters $\vec{\theta} \leftarrow \vec{\theta} + \alpha\nabla_{\vec{\theta}} J(\vec{\theta})$

The objective function and its gradient shown in step 2 are derived in the following way. Begin by defining $P_{\vec{\theta}}(\tau)$ as the probability of a given trajectory τ occurring. $P_{\vec{\theta}}(\tau)$ may be decomposed into the product of conditional probabilities at each time t; the probability of the agent taking an action a given s, $\pi_{\vec{\theta}}(a_t|s_t)$, multiplied by the probability of the transition to some new state at time $t+1$, $\mathcal{P}_{s_t s_{t+1}}^{a_t}$, which depends on the previous state and action, and is governed implicitly by the environment. If $\mathcal{P}_{s_0}$ is the probability of the initial state occurring then we define the full compound probability of a single trajectory as

$$\begin{aligned} P_{\vec{\theta}}(\tau) &= P_{\vec{\theta}}(s_0, a_0, r_1, ..., a_{T-1}, r_T, s_T) && (15)\\ &= \mathcal{P}_{s_0}\prod_{t=1}^{T-1}\pi_{\vec{\theta}}(a_t|s_t)\mathcal{P}_{s_t s_{t+1}}^{a_t} && (16)\end{aligned}$$

where s_T is the terminal state of the episode. Since possible trajectories are sampled from the distribution defined by (16) we write $\tau \sim P_{\vec{\theta}}(\tau)$. Then defining $R(\tau) := \sum_t r(a_t, s_t)$ as the return generated from a single trajectory, we can express the objective function as the expected reward over all possible trajectories as

$$\begin{aligned} J(\vec{\theta}) &= E_{\tau\sim P_{\vec{\theta}}(\tau)}[R(\tau)] && (17)\\ &= \int P_{\vec{\theta}}(\tau)R(\tau)d\tau && (18)\end{aligned}$$

Ignoring for the moment that the integral at (18) is intractable, since the explicit probability density function governing the initial state and the state transitions is not know, what one would like to do next is take the gradient of the objective function so as to perform an update to the parameters, as in step 3 of the algorithm. Using the properties of the logarithm we can manipulate the integrand in order to transform the gradient of the objective function into something we can approximate:

$$\begin{aligned} \nabla_{\vec{\theta}} J(\vec{\theta}) &= \nabla_{\vec{\theta}} E_{\tau\sim P_{\vec{\theta}}(\tau)}[R(\tau)] && (19)\\ &= \nabla_{\vec{\theta}}\int P_{\vec{\theta}}(\tau)R(\tau)d\tau && (20)\\ &= \int P_{\vec{\theta}}(\tau)\frac{\nabla_{\vec{\theta}} P_{\vec{\theta}}(\tau)}{P_{\vec{\theta}}(\tau)}R(\tau)d\tau && (21)\end{aligned}$$

$$= \int P_{\vec{\theta}}(\tau)\nabla_{\vec{\theta}} log(P_{\vec{\theta}}(\tau))R(\tau)d\tau \tag{22}$$

$$= E_{\tau\sim P_{\vec{\theta}}(\tau)}[\nabla_{\vec{\theta}} log(P_{\vec{\theta}}(\tau))R(\tau)] \tag{23}$$

Thus, turning the gradient of an expectation (19) into the expectation of a gradient (23). Now, notice that, from (16)

$$\nabla_{\vec{\theta}} log(P_{\vec{\theta}}(\tau)) = \nabla_{\vec{\theta}} log(\mathcal{P}_{s_1}\prod_{t=1}^{T-1}\pi_{\vec{\theta}}(a_t|s_t)\mathcal{P}^{a_t}_{s_t s_{t+1}}) \tag{24}$$

$$= \nabla_{\vec{\theta}}[log(\mathcal{P}_{s_1}) + \sum_{t=1}^{T-1}(log(\pi_{\vec{\theta}}(a_t|s_t)) + log(\mathcal{P}^{a_t}_{s_t s_{t+1}}))] \tag{25}$$

$$= \sum_{t=1}^{T-1}\nabla_{\vec{\theta}} log(\pi_{\vec{\theta}}(a_t|s_t)) \tag{26}$$

where the the gradient operator annihilates all the state transition probabilities, as they do not depend on $\vec{\theta}$. Then we perform a Monte Carlo Simulation, which is the process of drawing a random variable from a distribution a large number of times and averaging out in order to approximate the expected value of that variable, or a function which depends on it. In theory, it is possible to approximate the true expected value arbitrarily accurately as N gets large since $E[f(x)] = \lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N} f(x_i)_{x_i\sim p(x)}$. Applying this idea along with the result in (26) to the expression for the gradient of the objective function in (23)

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = E_{\tau\sim P_{\vec{\theta}}(\tau)}[\nabla_{\vec{\theta}} log(P_{\vec{\theta}}(\tau))R(\tau)] \tag{27}$$

$$= \lim_{N\to\infty}\frac{1}{N}\sum_{i=1}^{N}[\sum_t \nabla_{\vec{\theta}} log(\pi_{\vec{\theta}}(a^i_t|s^i_t))][\sum_t r(a^i_t|s^i_t)] \tag{28}$$

which gives the final expression for the gradient of the objective function:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}[\sum_t \nabla_{\vec{\theta}} log(\pi_{\vec{\theta}}(a^i_t|s^i_t))][\sum_t r(a^i_t|s^i_t)] \qquad \text{where } N >> 1 \tag{29}$$

5.2 Deep-Q Learning

One-step Q-learning is a temporal difference method where, at each time step, the observed reward resulting from the previous action is used to update the action value function using the the update rule

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{30}$$

Where $0 < \alpha < 1$ is a learning rate parameter. If infinite sampling of trajectories is allowed, and in each state all possible actions have a non-zero probability of being selected, we are guaranteed that the values of all state-aciton pairs will continue to be updated. It has been proven that if these conditions hold, we are guaranteed convergence of $Q \to Q^*$ with probability 1 [1].

As with MCPD, we would like to generalise the method of Q-learning for reinforcement learning tasks involving discrete state spaces to ones where the state spaces are continuous. Mnih et al. (2013) developed a generalisation of Q-learning for such tasks. In their paper [2], they propose a *Deep Q-network* (DQN) $Q_{\vec{\theta}(s,a)} \approx Q^*(s,a)$; an ANN parameterised by $\vec{\theta}$ which approximates the optimal action value function. An example illustration for this may be seen in figure 6. The technical details of the algorithm in [2] will not be described here, but essentially the strategy is, as with MCPD, to iteratively update the parameters of the network in order to converge on the optimal action value function. Again, this is achieved through taking the gradient of the expression for the chosen action (produced by the function approximator) with respect to the parameters. The difference being that, in Deep Q-Learning, instead of an objective function that measures expected reward, we have a loss function that measure the error between our parameterised Q-network and Q^*. This is achieved through sampling of one-step trajectories and updating the value estimate using the gradient of a loss function. In fact, in the algorithm described in [2], there are a sequence of loss functions that occur during training, at each iteration i the loss function is defined as

$$L_i(\vec{\theta}_i) = E_{s,a\sim\rho(.)}[(y_i - Q_{\vec{\theta}_i}(s,a))^2] \tag{31}$$

where $y_i = E_{s'\sim\mathcal{E}}[r + \gamma \max_{a'} Q_{\vec{\theta}_{i-1}}(s',a')|s,a]$ is the target at iteration i, $\rho(s,a)$ is the distribution over states and actions (as with MCPD, a product of conditional probabilities), and $\mathcal{E}$ is the environment. In a similar way to MCDP, the learning algorithm works by sampling one-step trajectories from the environment, storing batches of tuples containing transition information at each time t; $(s_t, a_t, r_{t+1}, s_{t+1})$. Then, at a regular intervals, two identical copies of the Q network are used for training, $Q_{\vec{\theta}_{i-1}}$ and $Q_{\vec{\theta}_i}$, where the weights $\vec{\theta}_{i-1}$ are held constant, and mini-batches of transitions are selected randomly from memory to minimise (31) using the gradient

$$\nabla_{\vec{\theta}_i} L_i(\vec{\theta}_i) = E_{s,a\sim\rho(.),s'\sim\mathcal{E}}[(r + \gamma \max_{a'} Q_{\vec{\theta}_{i-1}}(s',a') - Q_{\vec{\theta}_i}(s,a))\nabla_{\vec{\theta}_i} Q_{\vec{\theta}_i}(s,a)] \tag{32}$$

where the expectation of gradient, as with MCPD, may be approximated by averaging over a large number of sampled tuples. Alternatively, as noted in [2], stochastic gradient descent may be used for efficiency. Since the loss function depends on the weights of the Q-network, a moving target is created (and hence a sequence of loss functions), which is the reason for holding one copy of the network fixed and performing gradient descent on a batch at each iteration. The complete algorithm may be found in [2], but a variation of it is used in the final section of the report to solve the Mountain Car problem.

5.3 Exploratory Behaviour in MCDP and DQN

A final comment on both the Deep Q-Learning (DQN) and MCPD algorithms is that, in order for the conditions for convergence to the optimal policy and value function to be satisfied, they must both operate as *off policy* algorithms. This means that they should learn the greedy strategy $\pi(s) = \max_a Q_{\vec{\theta}}(s,a)$, but choose actions during sampling according to an ϵ-soft policy to maintain sufficient exploration of the state space. Typically, both algorithms use this method of stochastic action selection and learn optimal behaviour without a world model, which is a model of the environment dynamics learned through experience. It will be shown below that the core learning algorithms presented above may be augmented to encourage broader exploratory behaviour in reward-sparse environments by incorporating such strategies as learning a world model (among other strategies) in order to define an implicit reward for mapping out the state space, even in the absence of explicit rewards.

PART B: A Survey of Curiosity-Driven Learning Algorithms

6 Curiosity Driven Reinforcement Learning

In this section, a summary of some of the relevant historical work on curiosity driven reinforcement learning is presented, followed by a summary of three promising curiosity based algorithms developed recently. The historical period considered is from 1991, when the initial ideas were proposed, until 2016, where after the most effective methods to date have been developed.

6.1 Curiosity, Novelty and Memory

In thinking how to embed an implicit drive to explore in a reinforcement learning agent, algorithmic solutions appear to fall into two broad categories, both involving world models of one form or another. The first is based on the idea of next-state prediction error, where the agent has a model of the world and tries to predict the next state, given the previous state and action. The second is based on memory, where the agent builds up a representation of the world and attempts to acquire a more complete knowledge of it's environment. In both cases the reward for actions leading to unexpected or unknown outcomes must be supplemented in order to encourage a exploratory behaviour. There are several ways for such a mechanism to be implemented, and methods differ across environments and tasks. The history of this research has been a process of attempting to formalise, both conceptually and mathematically, curiosity and exploration in the context of reinforcement learning in an attempt to find a general theory that may be broadly applied.

6.2 A Brief History of Curiosity and Exploration

In 1991 Schmidhuber published a paper titled *A Possibility for Implementing Curiosity and Boredom in Model-Building Neural Controllers* [3] in which he proposes the foundational ideas for using world models to encourage exploration in reinforcement learning agents. He then published a paper in 1993 along with two other researchers titled *Reinforcement Driven Information Acquisition In Non-Deterministic Environments* [5] where he builds on ideas from his previous paper, using ideas from the field of information theory and proposing a method for maximising the acquisition of information about an environment.

In [3] Schmidhuber proposed the idea of a learning algorithm that would reflect what is observed in many biological learning systems; an interplay between goal directed learning, such as minimising pain and maximising reward, and explorative learning. In explorative learning, Schmidhuber frames the objective as simply 'increasing ones knowledge about the world', defining curiosity loosely, writing "one gets curious when one believes there is something about the world one does not know". A claim made in the paper is that this idea had not yet been explored in connectionist literature (connectionism is a movement in cognitive science that hopes to explain intellectual abilities using artificial neural networks). The learning algorithm proposed by Schmidhuber makes use of one larger artificial neural network, composed of two smaller networks, a controller network and a model network. The controller network has the job of making and learning optimal action selections in order to maximise reward over time, similar to the parameterised policy described in MCDP. The model network has the job of learning an accurate model of the environment dynamics, taking in a vector representation of an action at each time step and trying predict the resulting state of the environment. The proposed model relies on a framing of the reinforcement learning problem that varies from the standard one described above in that the state vector resulting from the previous time step, used as input to the controller network, includes the reward signal which is split up into reinforcement (positive) and pain (negative) signals. This makes the model recurrent in nature. The proposed learning algorithm then has two main goals; first, to minimise pain and maximise reinforcement, and second, to build an accurate model of the environments dynamics. At each time step, the agent observes the state information, including reward and pain signals resulting from the previous action selection, where after it selects an action for the current time step which it feeds into the environment, as well as the world model. Updating of the parameters works by computing the gradient of a loss function, defined as the euclidean norm of the difference between the predicted and actual resultant state. The gradient is computed and used to iteratively improve the models knowledge of the environment dynamics. The parameters of the controller network are then updated in order to minimise the difference between the pain and reinforcement values predicted by the model network, and the desired values. In order to update the parameters, the gradients are essentially propagated through the model network, to the controller network. Since this is the case, only if the model network is a good predictor of the environments dynamics can we expect the controller network to converge. An illustration of the model proposed by Schmidhuber is shown below.

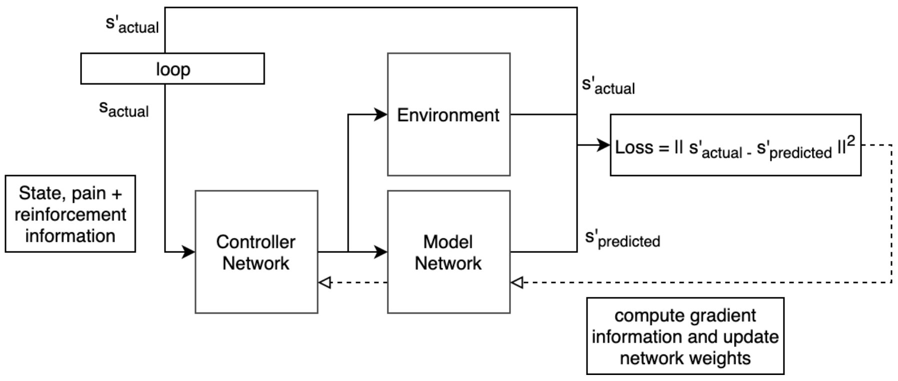
s'actual
loop
sactual
State, pain + reinforcement information
Controller Network
Environment
Model Network
s'actual
Loss = || s'actual - s'predicted ||2
s'predicted
compute gradient information and update network weights

Figure 10: The model proposed in Schmidhuber's 1991 paper [3]

The key idea proposed in [3] for augmenting such a model to encourage exploration, is to include as part of the reinforcement (reward) feedback at each time step a small positive amount proportional to the prediction error made by the model network at each time step. In this way, the agent is rewarded for taking actions that result in something that the model network predicts poorly, and the behaviour resulting in such situations will be reinforced until the model network learns to predict the outcome accurately. In this way, Schmidhuber proposed that it should be possible to encourage exploration of the state space in the absence of explicit reward, and that upon sufficient exploration of a local subset of the state space the agent would exhibit something akin to boredom. Then, as the world model more accurately predicts the dynamics of the environment, one would expect convergence of the controller network since the predictions of pain and reinforcement resulting from actions in familiar situations would be accurate, too. Although results of testing an implementation were not included in the paper, the idea of using the next-state prediction error as an implicit reward to reinforce explorative behaviour has been used in several models developed in recent years which have achieved state of the art performance in reward-sparse settings (mostly video game environments) for which significant progress was not possible using stochastic action selection techniques, as in ϵ-soft policies.

In [5], Schmidhuber et al. incorporated some of the ideas in [3], implementing a variation of standard

Q-learning with rewards based solely on exploration. In order to achieve this, the authors propose building a world model for a non-deterministic environment where the state space is discrete, by keeping count of the state-action pairs visited, as well as the resultant states to which they transition. Through keeping count, one is able to approximate state transition probabilities with increasing accuracy by sampling trajectories from the environment. The novel idea presented by the authors is an algorithm called *Reinforcement Driven Information Acquisition*, where the only reward for actions taken is a measure of difference between the approximated discrete probability distribution from one time step to the next (the authors propose entropy loss, Kullback-Leibler distance or a simple sum over absolute differences). In this way the agent is drawn to state-action pairs where it stands the best chance of learning something new - improving its knowledge of the world - in the sense that it's approximation of the discrete probability distribution representing possible transitions from the state-action pair is incorrect by some measurable amount, and may be improved through experience.

Although similar ideas were developed in other fields of study over the next decade, no explicit progress was made in the field of reinforcement learning that built on the existing theory after the papers by Schmidhuber [6]. In 2004 and 2005 Barto et al. wrote two papers ([6], [7]) based on a single large computational study of intrinsically motivated reinforcement learning, taking the ideas introduced by Schmidhuber and extending them, proposing a new framework which emphasises the utility of explorative behaviour for developing re-usable skills. The authors make the assertion that intrinsically motivated behaviour in animals leads to the development of broad sets of reusable skills necessary for successfully navigating their environment. They argue that such skills are developed due to an intrinsic reward system that favours "development of broad competence rather than being directed to more specific externally-directed goals", and that the skills are then used (and re-used) in an adaptive manner to solve more specific tasks over time. The authors then introduce a new model for learning where the agent is allowed to develop options, which are essentially new skills, and option models. An option is described by the authors as a subroutine, or a sequence of a combination of primitive actions and/or other options (a primitive action is simply an action as described thus far). Invoking an options triggers a sequence of actions, each of which may affect the environment. An option model models a portion of the environment dynamics, and enables the agent to predict what the result of exercising an option will be. In the toy problem used by the authors to illustrate the new framework, the agent exists within a grid world environment where, initially, the agent has only a set of primitive actions it may select from. In the environment, different sequences of primitive actions trigger what are called 'salient events'. When such a sequence of events is executed, the corresponding option and option model are created and made available to the agent. In response the salient event generates a stimulus for the agent (e.g. turning on a light), which is a response the agent fails to predict initially, but learns over time by improving the option model. To incentivise exploration, as proposed by Schmidhuber in [3], the agent is rewarded proportional to the error of its prediction. In the grid world environment, there is only one salient event that delivers explicit reward, and this event may only be triggered by a sequence of 14 primitive actions. The authors illustrate experimentally that the agent is able to learn options which combine sequences of actions and other options in order to learn how to trigger the salient event that delivers the explicit reward, significantly faster that an agent relying on extrinsic reward and ϵ-greedy action selection only. The results of this experiment, illustrated in figure 11, show that the agent equipped with intrinsic motivation and the ability to learn skills manages to find the explicit reward orders of magnitude faster than an agent relying only on stochastic action selection. This work has not been extended to applications in deep reinforcement learning as of yet.

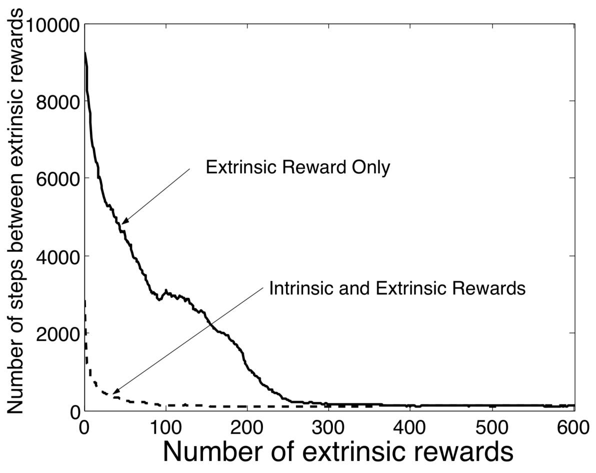

Figure 11: Results from [6], [7]. Intrinsically motivated agent learns how to generate extrinsic reward faster than an agent relying only extrinsic reward to learn the optimal policy

Attempts were then made in 2009 ([8]) and 2010 ([9]) to present a general, unified theory of intrinsic motivation in reinforcement learning, with [9] incorporating the ideas of Barto et al. in [7]. However, it appears that these ideas have yet to take hold in the mainstream. In the following years improvements in computation brought on the emergence of modern deep reinforcement learning (reinforcement learning methods reliant on deep neural networks, such as MCDP and DQN), and the initial ideas presented by Schmidhuber provided the basis for count-based (memory) and predictive methods for designing curiosity-driven learning algorithms.

In 2012 a paper ([10]) titled *Exploration in Model-based Reinforcement Learning by Empirically Estimating Learning Progress* published by Lopes et al. made robust improvements to count-based exploration algorithms for finite state and action spaces, R^{max} and Bayesian Exploration Bonus (BEB). The authors improve on the previous algorithms by estimating which parts of the state and action space are likely to yield significant improvement in knowledge about the dynamics of the environment empirically, without relying on prior estimations about visitation count thresholds or assumptions about the degree to which the environment is stochastic. They achieve this by defining a function $\zeta(s, a; k)$ which estimates to what extent the previous k visits to the state-action pair (s, a) improved the agents knowledge of the environment dynamics, i.e. the estimated learning progress. The function ζ is unique for each state-action pair, and is updated based off empirical knowledge gained from experience in the environment, making it able to adjust to changes in environment dynamics (non-stationary). The algorithm proposed by the authors uses the Q-Learning algorithm for finite state and action spaces presented by Sutton and Barto in [1], augmenting the reward function using ζ as follows. For the improved R^{MAX} algorithm, $\zeta - R^{\text{MAX}}$, the authors implement the reward function

$$R^{\zeta-R-MAX}(s,a) = \begin{cases} R(s,a) & \zeta(s,a;k) < m \\ R^{\text{MAX}} & \text{otherwise} \end{cases}$$

In words, the reward from the environment is given as per usual if nothing stands to be learned from taking action a in state s, but if something stands to be learned the reward is supplemented to encourage the agent to return to the state-action pair. Then, for the improved BEB algorithm, ζ-BEB, the authors implement the reward function

$$R^{\zeta-EB}(s,a) = R(s,a) + \frac{\beta}{1 + \frac{1}{\sqrt{\zeta(s,a;k)}}}$$

for some constant β. Both algorithms are shown to improve on the performance of the originals, the method of exploration under a standard ϵ-greedy policy, in a baseline setting (a version of gridworld).

In 2016, Bellemare et al. (DeepMind) published *Unifying Count-Based Exploration and Intrinsic Motivation* [11] address the challenge of generalising previous count-based methods of exploration to continuous state-spaces, using non-tabular reinforcement learning methods. The observation is made in the paper that for continuous states, or in cases where the state space is very large, it becomes unlikely for a state to be visited more than once, if ever, hence the need for generalisation. The authors extend the simple idea of counting state-action pairs and state transitions, applying it to the case of continuous state spaces constructing a probability density model from what they call 'pseudo-counts'. The density model gives the probability of observing a new state x given a finite sequence of states $x_1 ... x_n$. The model is used to approximate information gain from one time step to another; the authors coin the approximation metric is 'prediction gain' which approximates the Kullback-Leibler Divergence (a measure of the difference between probability distributions). The prediction gain is used in a function which then yields an approximation for the pseudo count, which is in turn used to augment the reward function, giving the agent a bonus reward proportional to the amount of knowledge acquired from explorative behaviour. The authors used the pseudo-count / prediction gain mechanisms to augment two industry standard algorithms, Deep-Q Network (DQN) and Asynchronous Advantage Actor-Critic (A3C), on a set of Atari 2600 games, which are considered benchmark tests for modern reinforcement learning algorithms. The augmented algorithms improved on the originals moderately across all tested games, and dramatic improvement on one game in particular, MONTEZUMA'S REVENGE, which the authors assert is the most challenging and unforgiving of all the Atari 2600 games, as the environment is incredibly reward-sparse and riddled with traps; each level requiring a lot of exploration and complicated sequences of actions to pass. The augmented DQN algorithm (with exploration bonus) managed to explore 15 of 24 rooms in the first level of the game, as opposed to the 2 rooms managed by the original algorithm (without exploration bonus), setting an industry record at the time the paper was written.

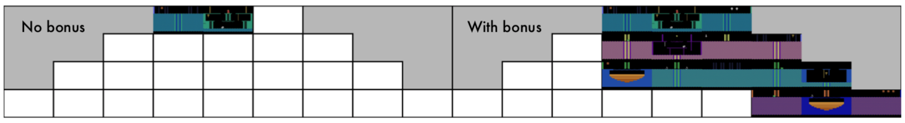

Figure 12: Record exploration by [11] on Montezumas revenge, using DQN with an exploration bonus.

There are several other papers which made contributions in the area of curiosity-driven reinforcement learning between 2010 and 2016 (e.g.[12], [13], [14], [15]), however for the sake of space and time they will not be covered here. Instead, in the following section, three of the most recently developed and promising algorithms are explored in detail.

6.3 Recent Developments

In this section, work from three recent papers is explored. The algorithms engineered by the authors have each achieved breakthrough performance in testing environments, all built on the ideas developed in the work described in the previous section. Each of the three algorithms share (at least) one thing in common; they rely on computation and the power of artificial neural networks as function approximators to generate results, rather than trying to devise a strategy that is mathematically or statistically elaborate.

6.3.1 Curiosity-Driven Exploration by Self-Supervised Prediction, 2017

The authors of [16] propose an Intrinsic Curiosity Module (ICM) which generates a reward bonus for explorative behaviour, r_t^i, in addition to any explicit reward received from the environment, r_t^e, based on the agents ability to predict the features of the next state given the current state and action. The authors used existing deep reinforcement learning algorithms in conjunction with the ICM to experiment with rewarding explorative behaviour in two video games, VizDoom and Super Mario Bros, as they presented challenges that had yet to be overcome by other methods at the time of writing. In formulating the problem, the authors propose that intrinsic reward may be formulated in two ways; (1) encouraging the agent to explore novel states, requiring, in continuous state spaces, a statistical model of the environments dynamics (as seen in [11]), or (2) encourage the agent to reduce uncertainty in it's ability to predict the consequent state from a state-action pair. When considering applying strategies of either form to reinforcement learning tasks involving high-dimensional, continuous state spaces, such as video games, both face two distinct challenges. First, the sheer number of possible permutations of pixels make the raw state space challenging to model and nearly impossible to predict at the pixel level, which may lead the agent to believe it is encountering novel states or acquiring significant new knowledge of environment dynamics when in fact it is not. Secondly, if the environment has stochastic elements in it's dynamics, the agent may become fixated on the source of randomness in the environment, believing it is exploring when it is not. The proposed solution, ICM, formulates intrinsic reward in the second way, next state prediction, and presents a solution that solves both of the challenges of next state prediction in both video game environments. The devised solution is tested on variations of the video game environments where the density of explicit rewards vary from very dense, to completely sparse (no explicit rewards), achieving promising results across the board.

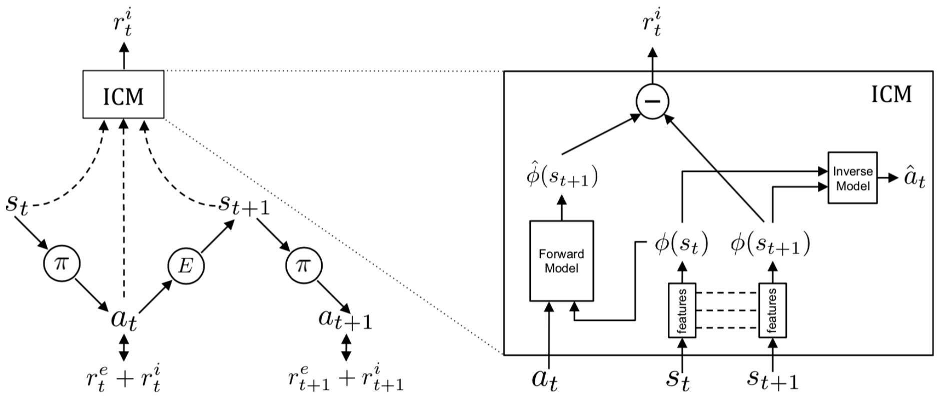

Figure 13: The ICM module proposed in [16]

As in in the MCPD algorithm, the authors make use of a policy gradient method during implementation, where the policy $\pi_{\vec{\theta}_P}(a|s)$ is an artificial neural network parameterised by $\vec{\theta}_P$ where the subscript P stands for "policy". As with before the policy gives a distribution of actions over states, and the goal of the learning algorithm is to maximise the expected reward, $\max_{\vec{\theta}_P} E_\pi[\sum_t r_t]$, through iteratively converging to the optimal policy. The difference is that, at each time step, the reward given to the agent is a sum of two parts, an extrinsic reward from the environment and an intrinsic reward generated by the agent, $r_t = r_t^e + r_t^i$. The intrinsic reward is generated by the ICM, which contains two components, each with a distinct function. The first component is the *inverse model*, which is an artificial neural network parameterised by $\vec{\theta}_I$. The inverse model has two sub-tasks. Firstly, it has the task of transforming the high-dimensional pixel representation of states s_t and s_{t+1} into lower-dimensional, latent feature vectors $\phi(s_t)$ and $\phi(s_{t+1})$, which we take from a hidden layer in the neural network, as pictured in figure 14.

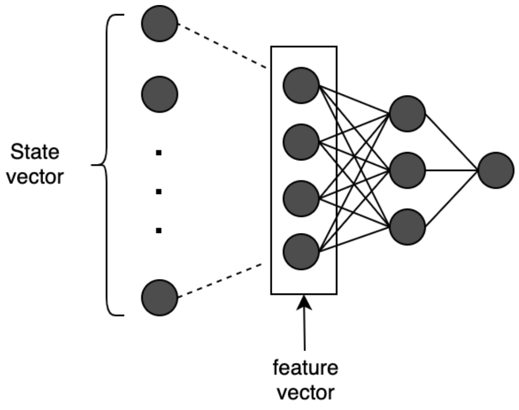

Figure 14: Example of a low-dimensional feature vector extracted from a hidden layer in a deep neural network

Secondly, it uses the feature vectors in order to inversely predict a_t, the action taken in state s_t, resulting in s_{t+1}. The predicted action $\hat{a}_t$ is, as usual, taken from the output layer of the neural network. The inverse model amounts to learning a function

$$\hat{a}_t = g(s_t, s_{t+1}; \vec{\theta}_I)$$

In training the neural network belonging to the inverse model the loss function optimised is

$$\min_{\vec{\theta}_I} L_1(\hat{a}_t, a_t)$$

where L_I computes a measure of difference between the actual and predicted action a_t. If a_t is discrete, the output of g is a soft-max distribution across all possible actions, in which case minimizing L_I amounts to computing a maximum likelihood estimation of θ_I under the produced multinomial distribution. In optimising the neural network in this way, the inverse model achieves the goal of encoding in the latent feature space **only state information which is affected by the agents actions in some way**. In other words, the neural network will filter out any elements of the environment captured in the raw pixel vector that are stochastic in nature. This property is crucial for constructing a good intrinsic reward for encouraging exploration, which is the task fulfilled by the second component of the ICM, the forward model. The forward model is also a neural network, parameterised by $\vec{\theta}_F$, which takes as input the action a_t and the latent feature vector $\phi(s_t)$, extracted from the inverse model, and attempts to predict the next-state latent feature vector $\hat{\phi}(s_{t+1})$. The forward model amounts to learning the function

$$\hat{\phi}(s_{t+1}) = f(\phi(s_t), a_t; \vec{\theta}_F)$$

where the loss function being optimised is the square of the L-2 norm of the difference between the predicted and actual latent feature representation of the state s_{t+1}

$$L_F(\phi(s_{t+1}), \hat{\phi}(s_{t+1})) = \frac{1}{2}||\phi(s_{t+1}) - \hat{\phi}(s_{t+1})||_2^2$$

The error produced by L_F will then be large if the agent fails to predict elements of the resulting states affected in some way by its actions. In this way, both the challenge of prediction in the high-dimensional state space and the challenge of avoiding getting caught in the trap of mistaking stochastic features for meaningful novelty are overcome. The intrinsic portion of the reward function fed to the agent at each time step is then proportional to L_F, giving the full reward function at time t

$$\begin{aligned} r_t &= r_t^e + r_t^i \\ &= r_t^e + \eta L_F \\ &= r_t^e + \frac{\eta}{2}||\phi(s_{t+1}) - \hat{\phi}(s_{t+1})||_2^2 \end{aligned}$$

where $\eta > 0$ is a small scaling factor. Constructing the reward function in this way encourages "good" explorative behaviour of the state space, and allows the agent to get "bored" as the forward model becomes better at predicting the latent features of the next-state.

The ICM was used in conjunction with the Advantage Actor Critic (A3C) algorithm, notated as ICM+A3C, and tested against two baselines algorithms in the video game settings; ICM(pixels)+A3C, in which the forward model learnt to predict raw pixel state vectors instead of latent feature vectors, and vanilla A3C. Results from testing with varying degrees of reward density in the VizDoom environment, see figure 15, show that even in the "very sparse reward" variant of the environment, the intrinsic reward encourages sufficient exploration to seek out sources of extrinsic reward that may exist. Additionally, the baseline Vanilla A3C method fails completely in the "sparse" and "very sparse" setting.

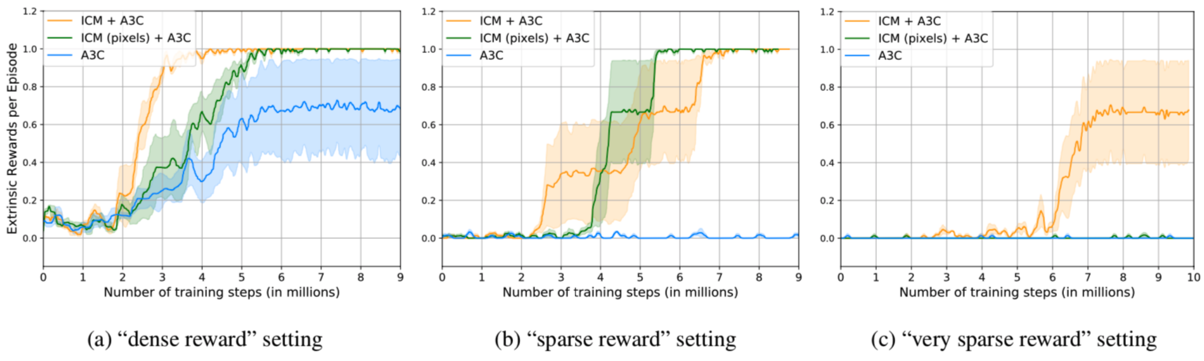

Figure 15: Results of ICM-A3C in the VizDoom environment with varied reward densities, dense(left) to very sparse(right), tested against two baseline algorithms, ICM(pixels)+A3C and Vanilla A3C

One shortcoming of the formulation of curiosity as implemented in the ICM is that in some situations an agent may be able to cause stochastic responses from the environment to manifest as a result of its own actions. The authors noted the problem as it was encountered in further research following the paper. This problem is addressed in both solutions described below.

6.3.2 Exploration by Random Network Distillation, 2018

In 2018, Burda et al. [17] proposed a simple method for encouraging exploration which demonstrably solved two problems experienced by methods proposed previously. First, the potential for being 'trapped' by stochastic elements - especially those controlled by the agent, a problem experienced by [16] - and second, the issue of scalability which becomes a problem for methods such as pseudo counts (e.g. [11]) or information gain (e.g. [5]), which becomes a significant challenge when the state space, or state-action space, is very large. The method proposed in [17] titled *Random Network Distillation* augments the reward function by generating a numerical reward for the agent at each time step t proportional to difference between the outputs of two artificial neural networks, given the an input vector representing the current state s_t. The two networks are called the *predictor network*, $\hat{f}_{\vec{\theta}} : \mathcal{S} \rightarrow \mathcal{R}^k$, and the target network $f : \mathcal{S} \rightarrow \mathcal{R}^k$. Before training, the weights for both the target network and the predictor network are randomly initialised. During training, the weights of the target network remain fixed, while the weights of the predictor network are iteratively updated (usually by means of a gradient computation as in MCDP) to better mimic the target network. If the state information for s_t is captured (either completely or partially) in the vector $\vec{x}$ at time t, then at each time step the loss function is the MSE of the output of the two networks:

$$L_{RND} = ||\hat{f}_{\vec{\theta}}(\vec{x}) - f(\vec{x})||^2 \tag{33}$$

This process distills a randomly initialised neural network into a trained one. The optimization problem is then exactly a supervised learning problem where the labels are simply produced by the static target network. This method works well in practice since neural networks typically have lower prediction errors on examples similar to those they have been trained on [17]. The authors demonstrate this distillation mechanism using the MNIST data set, where they train a predictor network to mimic a randomly intialised target network given training data consisting of the instances of the zero digit class, as well as instances from a second target class of digits from 1-9. During training the number of images of the zero digit are held constant, serving as the 'frequently observed state', and the number of the images from the second digit in the training set, representing the lesser-visited state, are varied from 0 to ≈5500. The network performance of the predictor network is then validated using the remaining instances of the second, lesser-seen digit. The results, illustrated in figure 16, show the MSE between the outputs of the predictor and target networks decreases with the number of lesser-seen digit instances added to the training set, asymptotically approaching 0.

Computing an exploration bonus in this way yields two distinct benefits. First, it is efficient because the computation of the prediction error is nothing more than a forward pass through a neural network. The forward pass together with the gradient updates give an algorithm that is able to scale approximately linearly with the size of the state space depending on how the neural network is constructed. The second benefit is that any stochastic elements in the environment cease to present the same danger as with previous prediction based algorithms, even in the case where the agent is able to cause a stochastic manifestation in the environment. This is because it does not matter whether the next state follows in an 'expected' way from the current state-action pair or not since the predictor network is not making next-state predictions using current state vectors, it is trying to mimic the feature vectors produced by the target network. Therefore even if the agent finds or causes a stochastic source within the environment, it will very quickly get 'board' with observing its effects.

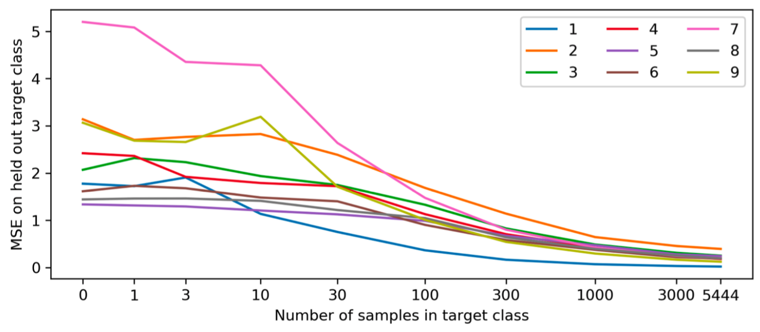

Figure 16: The MSE between the outputs of the predictor and target networks trained on different classes from the MNIST data set, as a function of the proportion of instances from the lesser-seen digit class (ranging from digits 1-9)

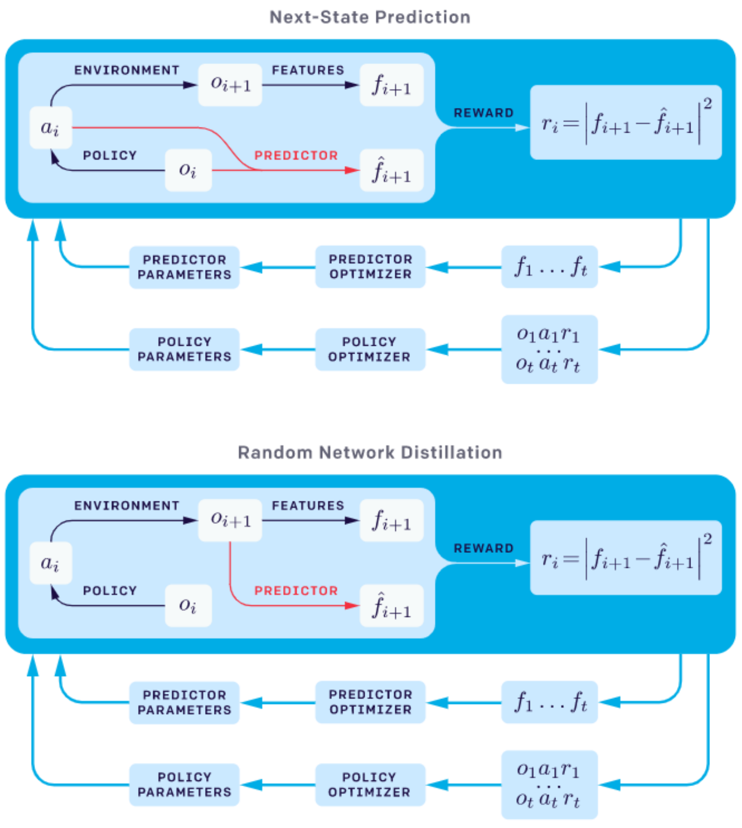

Figure 17: Visual representation of the RND algorithm versus next-state prediction (image sourced from openai.com/blog/reinforcement-learning-with-prediction-based-rewards)

The authors of [17] tested the RND algorithm on various Atari 2600 games including MONTEZUMA'S REVENGE where the agent managed to explore all 24 rooms in the first level, as well as pass the first level. The algorithm was tested against two baseline algorithms; Proximal Policy Optimization (PPO), as well as PPO augmented with a next-state prediction bonus which the authors call 'forward dynamics'. RND shows comparable or improved performance in all reinforcement tasks on which it was tested, and far superior performance in tasks where the environment is reward sparse.

Commenting on possible future work, the authors state that even with the exploration bonus generated by RND the agent struggles to learn long or complicated sequences of actions, and that the PPO + RND agent

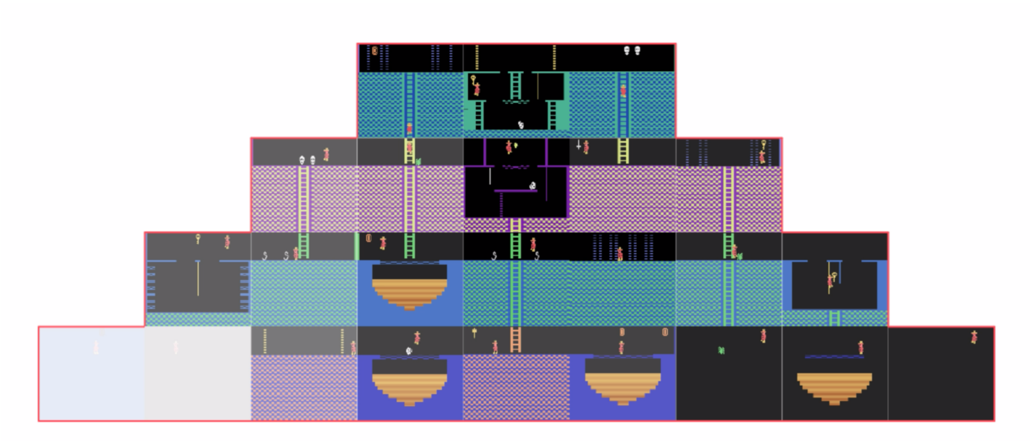

Figure 18: Rooms in the first level of MONTEZUMA'S REVENGE visited by the reinforcement learning agent trained using PPO and RND

only passed the first level rarely through chance exploration. Future research, they propose, should investigate how to equip agents with the ability to learn these sequences more effectively. This is similar to the thesis of the work in [6] by Barto et al. which could serve as a basis for further research.

6.3.3 Episodic Curiosity Through Reachability, 2019

Most recently, in 2019, Savinov el al. from Google published a paper presenting a model-based method for encouraging exploration where the agent is able to compute a measure of distance between new observations and existing observations stored in memory, rewarding the agent for reaching novel states which, in their words, 'take some effort to reach'. The authors conceptualise curiosity as novelty seeking behaviour but make the observation that there is a difference between novelty that arises simply from an inability to predict something which hasn't been seen before, as is the problem when an agent may cause a stochastic response from the envirnoment, and the novelty that arises from observing things that are unfamiliar from things you have experienced previously. The method of *episodic curiosity through reachability* as outlined in the paper is designed to reward the agent for the latter.

The Episodic Curiosity (EC) module described in [18], and illustrated in figure 19, is designed to take as input an observation $\mathbf{o}_t \subseteq s_t$ at time t, outputting a numerical exploration bonus b which may be negative or positive. The bonus is then added to the reward from the environment, r_t, to create the augmented reward $\hat{r}_t = r_t + b$. The components inside the module are: an embedding network $E : \mathcal{O} \rightarrow \mathcal{R}^n$, a comparator network $C : \mathcal{R}^n \times \mathcal{R}^n \rightarrow [0, 1]$, an episodic memory buffer $\mathbf{M}$, a reachability buffer and a reward bonus estimation function B.

The embedding and comparator network function jointly to estimate the so-called *within-k-reachability* of one observation $\mathbf{o}_i$ to another $\mathbf{o}_j$, together forming a reachability network $R(\mathbf{o}_i, \mathbf{o}_j) = C(E(o_i), E(o_j))$. At each time step an observation $\mathbf{o}_i$, in the form of a vector, is fed into the embedding network which produces a latent-space feature vector $\mathbf{e}_i = E(\mathbf{o}_i)$ (as with the ICM). The comparator network then estimates the probability $c_i = C(\mathbf{e}_i, \mathbf{e}_j)$ that $\mathbf{o}_i$ is within k-steps from each $\mathbf{o}_j$ for $j = 1, ..., |\mathbf{M}|$ stored previously in the memory buffer $\mathbf{M}$. Each c_i is stored in a reachability buffer until the M^{th} (final) within-k-reachability probability is computed. Then, an aggregation function $F(c_1, ..., c_{|\mathbf{M}|})$ takes as input all computed probabilities in the reachability buffer, and uses the 90^{th} percentile to compute the similarity score. The reason the 90^{th} percentile are used as opposed to the $\max(c_1, ..., c_{|\mathbf{M}|})$ is because, in practice, taking the maximum score as representative of the most similar state was observed to be susceptible to outlier-related errors. Finally, the curiosity score is computed as $b = B(\mathbf{M}, \mathbf{e}_i) = \alpha(\beta - F(c_1, ..., c_{|\mathbf{M}|}))$ where $\alpha, \beta \in \mathcal{R}$ are hyperparameters of the method. If the computed score b is greater than some threshold $b_{novelty}$, the feature vector $E(\mathbf{o}_i)$ is added to the memory buffer $\mathbf{M}$. Additionally, to avoid memory and performance problems, when a new observation's feature vector is added to the memory buffer, a previously stored vector is selected at random and deleted.

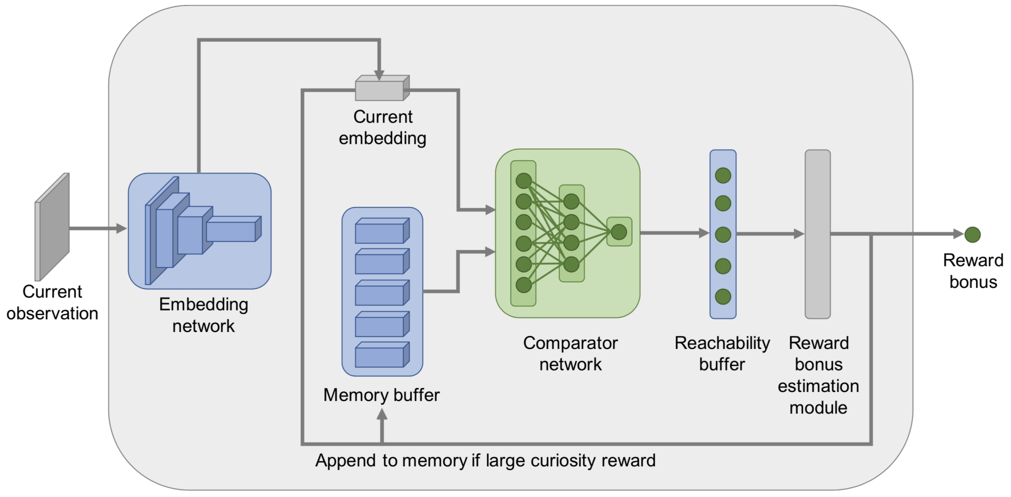

Figure 19: An illustration of the episodic curiosity module as described in [18]

Notice that in a finite state space one could compute the steps required to move from one state to another by representing the state space as a weighted graph, in which case there would be no need to use a neural network to estimate the distance between pairs of observations. In large, continuous state spaces however, a function approximator is required. In order to train the reachability network to accurately approximate steps between observations the authors found that one of two methods may be used. One may allow the agent to sample trajectories from the environment before playing the game and create a labelled data set for training the network. Alternatively, one may train the network online while the tasks is being performed by optimising iteratively every finite number of steps.

The EC module was used in conjunction with PPO, and tested in various 3D, visually rich environments including *VizDoom* and *DMLab*. As in [16] the tests were conducted with varied reward-density settings, from very-sparse to dense, against several baseline algorithms including the PPO + ICM algorithm developed by [16] as well as a standard PPO algorithm, which relied only ϵ-greedy exploration only. The EC + PPO agent exhibited extensive explorative behaviour, and in the total absence of explicit rewards was found to cover 4 times more area (measured in discrete (x, y) coordinates) than the baseline PPO + ICM agent. Results of experiments for different reward densities are illustrated in figure 20.

PART C: Experiment

In this final section, results of an attempt to use the random network distillation method along with DQN to solve the classic toy problem *Mountain Car* are presented.

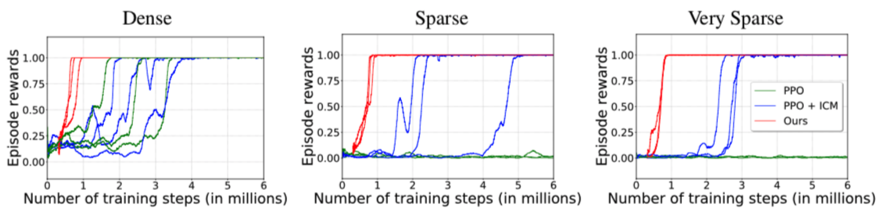

Figure 20: Results from testing the EC module with PPO. Explicit rewards as a function of training steps in *VizDoom*.

7 Problem: Mountain Car

Mountain car is a simple reinforcement learning task that is notoriously tricky to solve without re-engineering the reward function or employing an exploration method beyond stochastic action. The task is simple, a car sits in a 2-dimensional valley with a mountain on either side. The agent must get the car to reach a flag at the top of a hill on the right using only three actions: push left, push right, do nothing. The agent receives -1 reward for each time step that elapses before it reaches the top of the hill, and a reward of +0.5 when it does. If 200 time steps pass without the car reaching the top, the game is over. The tricky part is that the car does not have enough power to drive straight up the either of the two mountains, so it must perform a coordinated sequence of actions in order to roll the car back and forth, up and down each of the slopes until it gains enough momentum to reach the top of the rightmost peak. The state space for the task consists of two continuous variables $v \in [-0.07, 0.07]$ and $x \in [-1.2, 0.6]$, representing horizontal position and velocity, respectively. Since the environment is reward sparse, solving the task will require intrinsic motivation to explore the state space, which should encourage the agent to manoeuvre the car further and further up the slopes, ultimately reaching the flag.

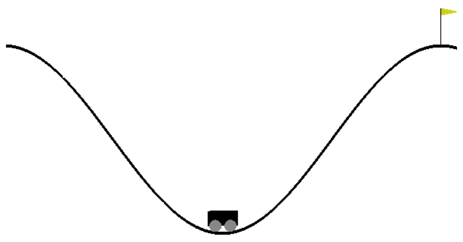

Figure 21: An illustration of the Mountain Car problem

7.1 Algorithm: DQN + RND

In order to solve the Mountain Car task, a simplified variation of the DQN algorithm was used with RND. At each time step t the agent takes an action in the environment (left, right or nothing) and observes a new state s_{t+1} and extrinsic reward r_t^e. The intrinsic reward r_t^i is then computed as the difference between the output of a predictor network and a target network, according to equation (33). The total reward is computed as $r_t = r_t^e + r_t^i$ and the transition (s_t, a_t, r_t, s_{t+1}) is added to a memory buffer B as per the standard DQN algorithm. Then each time step a random minibatch of samples is drawn from B and for each sample drawn, a gradient descent step is performed for the action-value function approximator Q_{θ_i} according to equation (32) and a a gradient descent step is performed for the RND predictor network according to $\theta_P \leftarrow \theta_P + 2\alpha||\hat{f}_{\theta_P}(s_t) - f(s_t)||\nabla_{\theta_P}\hat{f}_{\theta_P}(s_t)$. The algorithm is given below in full.

Algorithm Deep-Q Learning + Random Network Distillation

```
initialise replay memory buffer $B$ to capacity N;
initialise action-value function approximator $Q_{\theta_i}$ with random weights;
create a deep copy of $Q_{\theta_i}$ to use as the target model; $Q_{\theta_{i-1}}$;
initialise RND target network $f$ with random weights;
initialise RND predictor network $\hat{f}_{\theta_P}$ with random weights;
for episode=1, M do
    reset environment and set initial state $s_0$;
    for t=1,T do
        with probability $\epsilon$ select random action $a_t$;
        otherwise select $a_t = \max_a Q(s_t, a_t; \theta_i)$;
        execute action $a_t$ in the environment and observe extrinsic reward $r^e_{t+1}$ and state $s_{t+1}$;
        compute intrinsic reward as $r^i_t = ||\hat{f}_{\theta_P}(s_t) - f(s_t)||^2$;
        set combined reward as $r_t = r^e_t + r^i_i$;
        store transition ($s_t$, $a_t$, $r_t$, $s_{t+1}$) in $B$;
        sample random minibatch of transitions ($s_j$, $a_j$, $r_j$, $s_{j+1}$) from $B$;
        for each transition in minibatch do
            set $y_j = \begin{cases} r_j & \text{if} s_{j+1} \text{is terminal} \\ r_j + \max_{a'} Q(s_{j+1}, a'; \theta_{i-1}) & \text{if} s_{j+1} \text{is non-terminal} \end{cases}$;
            perform a gradient descent step $(y_j - Q(s_j, a_j; \theta_i))$ according to equation (32);
            perform a gradient descent step for the RND predictor network:
             $\theta_P \leftarrow \theta_P + 2\alpha||\hat{f}_{\theta_P}(s_t) - f(s_t)||\nabla_{\theta_P}\hat{f}_{\theta_P}(s_t)$
        end
        reset $Q_{\theta_{i-1}} = Q_{\theta_i}$ after every $K$ time steps;
    end
end
```

7.2 Results

The algorithm was run for 300 episodes with the intrinsic reward, as well as without the intrinsic reward. The results are shown in figure 22. In the case where the intrinsic reward was used, the agent explores without attaining any explicit reward for approximately 100 episodes initially, attaining only intrinsic reward. At about the 100th time step a rise in the average intrinsic reward preempts a rapid increase in the average extrinsic reward as the car makes it to the top of the hill for the first time. Where after both the intrinsic and extrinsic rewards shoot up rapidly. In the case where no intrinsic reward is used, the agent relies only on ϵ-greedy action selection and eventually figures out how to climb to the top of the hill after approximately episode 230.

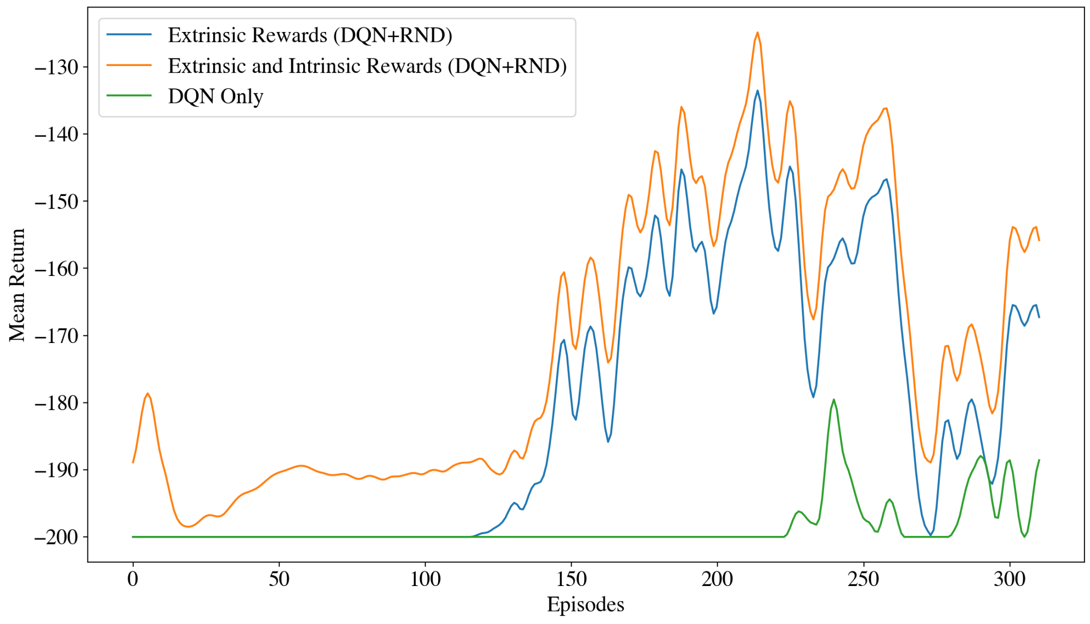

Figure 22: Mean returns generated from training an agent on the Mountain Car task, using the DQN algorithm augmented by Random Network Distillation

The intrinsic reward generated by the RND module has a positive effect in encouraging curious behaviour, even in the simple Mountain Car environment and clearly outperforms the standard DQN algorithm. The drop in extrinsic and intrinsic reward at approximately episode 250 is perhaps the agent trying to figure out a more efficient way of attaining the goal through further exploration. The average reward quickly increases again at around episode 300 - if left to continue the agent would eventually learn the most efficient action sequence for ascending the mountain.

8 Conclusions & Possible Future Work

This report has outlined the mathematical framework for the reinforcement learning problem, as well as the algorithmic approach to solving it using Generalised Policy Iteration (GPI). Two modern algorithms used to solve the problem of generalisation to continuous state spaces were covered, along with a mathematical analysis of the first method, Monte Carlo Policy Differentiation (MCPD). Section B covered a brief history of the development of curiosity-driven reinforcement learning algorithms, followed by a detailed examination of the three most recent algorithms: the Intrinsic Curiosity Module (ICM) which made use of next-state feature prediction, Random Network Distillation (RND) which indirectly modelled environment dynamics by distillation a randomly initialised neural network, and finally the Episodic Curiosity (EC) module which estimated the within-k-step reachability of new observations from observations in memory. The latter two methods solved the problem of agent-elicited stochastic response from the environment, and all three were robust under tests in high-dimensional state spaces under varied reward densities. Finally, Section C demonstrated the effectiveness of RND, used in combination with DQN to solve the classic reward-sparse Mountain Car Problem.

In order to build on the work explored in this paper, and the three main algorithms in particular, it appears that solutions to the problems arising from an agents inability to learn sequences of actions, what Barto et al. (2004) called ’skills’ or ’options’, should be explored. If an agent is unable to learn actions that are not directly made available to it by the programmer, there will naturally be a limit to solutions involving intrinsic reward that are limited in this way - this was demonstrated by Burda et al. (2018) in [17] when the agent was tested in MONTEZUMA’S REVENGE. Additionally, in taking direction from biological systems and nature, it appears that the argument made by Barto et al. (2004) is supported by what we observe in the natural world, which is that animals and human beings, intrinsically motivated to explore, gradually develop skills made up of sequences of smaller actions. A young animal or child will continue to develop more complicated skills and learn how to adapt them to various categories of tasks in order to achieve specific goals, or to satisfy the urge for further exploration. The foundations for implementing such ideas in the domain of reinforcement learning have been laid in [6] and a fruitful avenue of further research may be to attempt to develop those ideas and extend them to the case of continuous state spaces and deep reinforcement learning.

References

[1] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1998.

[2] Mnih et al. Playing Atari with Deep Reinforcement Learning. Deep Mind, 2013.

[3] J. Schmidhuber. A possibility for implementing curiosity and boredom in model-building neural controllers. In From Animals to Animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior, pages 222–227, Cambridge, MA, 1991. MIT Press.

[4] J. Schmidhuber. Curious Model-Building Control Systems In Proc. International Joint Conference on Neural Networks, Singapore, volume 2, pages 1458-1463. IEEE, 1991.

[5] J. Schmidhuber and J. Storck. Reinforcement driven information acquisition in non-deterministic environments Technical report, Fakultat fur Informatik, Technische Uni- versit at Munchen, 1993.

[6] A.G.Barto, S.Singh, and N.Chentanez. Intrinsically motivated learning of hierarchical collections of skills. In Proceedings of the 3rd International Conference on Developmental Learning (ICDL '04), LaJolla CA, 2004.

[7] A.G.Barto, S.Singh, and N.Chentanez. Intrinsically Motivated Reinforcement Learning. In Advances in Neural Information Processing Systems 17: Proceedings of the 2004 Conference, Cambridge MA, 2005. MIT Press.

[8] P.-Y. Oudeyer and F. Kaplan. What is intrinsic motivation? a typology of computational approaches. Frontiers in neurorobotics, 2009. 1, 9.

[9] Satinder Singh, Richard L. Lewis, Andrew G. Barto, Fellow, IEEE, and Jonathan Sorg. Intrinsically Motivated Reinforcement Learning: An Evolutionary Perspective. IEEE TRANSACTIONS ON AUTONOMOUS MENTAL DEVELOPMENT, vol. 2, no. 2, June 2010.

[10] Lopes, M., Lang, T., Toussaint, M. and Oudeyer, P.Y. Exploration in Model-based Reinforcement Learning by Empirically Estimating Learning Progress. In Advances in neural information processing systems (pp. 206-214), 2012.

[11] Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D. and Munos, R. Unifying count-based exploration and intrinsic motivation. In Advances in Neural Information Processing Systems (pp. 1471-1479), 2016.

[12] Houthooft, R., Chen, X., Duan, Y., Schulman, J., De Turck, F. and Abbeel, P. VIME: Variational Information Maximizing Exploration. In Advances in Neural Information Processing Systems (pp. 1109-1117), 2016.

[13] Stadie, B.C., Levine, S. and Abbeel, P. Incentivizing exploration in reinforcement learning with deep predictive models. arXiv preprint arXiv:1507.00814, 2015.

[14] Ostrovski, G., Bellemare, M.G., van den Oord, A. and Munos, R. Count-Based Exploration with Neural Density Models. In Proceedings of the 34th International Conference on Machine Learning-Volume 70 (pp. 2721-2730), 2017.

[15] Burda, Y., Edwards, H., Pathak, D., Storkey, A., Darrell, T. and Efros, A.A., 2018. Large-Scale Study of Curiosity-Driven Learning. arXiv preprint arXiv:1808.04355, 2018.

[16] Pathak, D., Agrawal, P., Efros, A.A. and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops (pp. 16-17), 2017.

[17] Burda, Y., Edwards, H., Storkey, A. and Klimov, O. Exploration by random network distillation. arXiv preprint arXiv:1810.12894, 2018.

[18] Savinov, N., Raichuk, A., Marinier, R., Vincent, D., Pollefeys, M., Lillicrap, T. and Gelly, S. Episodic curiosity through reachability. arXiv preprint arXiv:1810.02274, 2019.

9 Appendix A: Summary of Notation

t	discrete time step
T	final time step of an episode
a_t	action at t
s_t	state at t (dependant on a_{t-1} and s_{t-1})
r_t	reward at t (dependant on a_{t-1} and s_{t-1})
R_t	return (cumulative, discounted reward) following t
R_t^n	n-step return
π	policy
$\pi(s,a)$	probability of taking action a under a *stochastic* policy
$\mathcal{S}$	set of all non-terminal states
$\mathcal{S}^+$	set of all states, including the terminal state
$\mathcal{A}(s)$	set of all possible actions in state s
$\mathcal{P}_{ss'}^a$	probability of transitioning from state s to s' under action a
$\mathcal{R}_{ss'}^a$	expected immediate reward on transition from s to s' under action a
$V^\pi(s)$	value of state s under policy π
$V^*(s)$	value of state s under an optimal policy
V, V_t	estimates of $V^\pi(s)$ or $V^*(s)$
$Q^\pi(s,a)$	value of taking action a in state s under policy π
$Q^*(s,a)$	value of taking action a in state s under an optimal policy
Q, Q_t	estimates of $Q^\pi(s)$ or $Q^*(s)$
$\vec{\theta}$	vector of parameters underlying V, Q or π
$\vec{\phi}_s$	vector of features representing state s
γ	discount factor parameter
α, β	step-size parameters
ϵ	probability of selecting a random action under an ϵ-greedy policy

```python
import torch
import torch.nn
import torch.nn.functional as F

class Network(torch.nn.Module):
    def __init__(self,n_input,n_output,n_hidden):
        super(Network, self).__init__()
        self.n_input = n_input
        self.n_output = n_output
        self.n_hidden = n_hidden

        self.layer_1 = torch.nn.Linear(n_input,n_hidden,'linear')
        self.layer_2 = torch.nn.Linear(n_hidden,n_hidden,'linear')
        self.layer_3 = torch.nn.Linear(n_hidden,n_output,'linear')

    def forward(self,x):
        y = F.relu(self.layer_1(x))
        y = F.relu(self.layer_2(y))
        y = self.layer_3(y)
        return y


class RND:
    def __init__(self,n_input,n_output,n_hidden, learning_rate=0.0001):
        self.target_network = Network(n_input,n_output,n_hidden)
        self.model_network = Network(n_input,n_output,n_hidden)
        self.optimizer = torch.optim.Adam(self.model_network.parameters(),lr=learning_rate)

    def get_reward(self,x):
        y = self.target_network(x).detach()
        y_pred = self.model_network(x)
        reward = torch.pow(y_pred - y,2).sum()
        return reward

    def update(self,Return):
        Return.sum().backward()
        self.optimizer.step()
```

```python
import copy
from collections import deque
import numpy as np
import random
from RND import RND
import torch
import torch.nn.functional as F


class Q_Network(torch.nn.Module): # Set up Q network
    def __init__(self,n_input,n_output,n_hidden):
        super(Q_Network, self).__init__()
        self.n_input = n_input
        self.n_output = n_output
        self.n_hidden = n_hidden

        self.layer_1 = torch.nn.Linear(n_input,n_hidden,'relu') # Input and hidden layer
        self.layer_2 = torch.nn.Linear(n_hidden,n_output,'linear') # Hidden layer and output layer

    def forward(self,x): # forward pass through the network
        y = F.relu(self.layer_1(x))
        y = self.layer_2(y)
        return y


class DQN_Agent:
    def __init__(self, env, gamma, buffer_size):
        # Environment setup
        self.env = env # Environment
        acts = env.action_space
        obs = env.observation_space

        # Q Network
        self.model = Q_Network(obs.shape[0],acts.n,64) # action-value function
        self.target_model = copy.deepcopy(self.model) # copy of action-value function for training
        self.optimizer = torch.optim.Adam(self.model.parameters(),lr=0.001) # Adam optimiser for gradient descent
        self.gamma = gamma # discount factor
        self.batch_size = 64 # Size of minibatch
        self.buffer_size = buffer_size # Size of buffer for storing transition tuples
        self.Qupdate_target_step = 500 # Number of steps to reset copy of Q network
        self.replay_buffer = deque(maxlen=buffer_size) # Buffer for storing transition tuples
        # RND Network
        self.rnd = RND(obs.shape[0],64,124) # random netowork distillation module

        # Hyperparaeters
        self.Q_step_counter = 0 # keep track of steps to reset copy of Q network
        self.steps = 0 # keeping track of total steps
        self.epsilon_low = 0.05 # minimum value for epsilon
        self.epsilon_high = 0.9 # maximum value for epsilon
        self.epsilon = self.epsilon_high # Initial value for epsilon
        self.decay = 200 # Decay rate for epsilon

    def run_episode(self):

        obs = self.env.reset() # Reset the environment and get the initial state
        sum_r = 0 # keep track of total explicit return
        sum_tot_r = 0 # keep track of combined reward (implicit + explicit)
        loss_t = 0 # keep track of loss

        for t in range(200):
            self.steps += 1
            self.Q_step_counter = self.Q_step_counter + 1

            state = torch.Tensor(obs).unsqueeze(0) # current state
            new_state, reward, done, info, action = self.step(state) # take a step in the environment
            sum_r = sum_r + reward # add to explicit reward total
            reward_i = self.rnd.get_reward(state).detach().clamp(-1.0,1.0).item() # compute intrinsic reward
            combined_reward = reward + reward_i # compute combined reward
            sum_tot_r += combined_reward # add to combined reward total

            self.replay_buffer.append([obs,action,combined_reward,new_state,done]) # add tuple to buffer
            loss_t += self.update_model() # update the Q_Network and RND module
            obs = new_state # next state

            if (self.Q_step_counter > self.Qupdate_target_step):
                self.target_model.load_state_dict(self.model.state_dict()) # reset copy of Q network
                self.Q_step_counter = 0 # reset Q step counter
                print('updated target model')
            if done:
                break
        return sum_r, sum_tot_r, loss_t/self.steps


    def step(self, state):
        Q = self.model(state) # Get Q values for actions given current state
        num = np.random.rand() # select a random float in [0,1]
        self.epsilon = self.epsilon_low + (self.epsilon_high-self.epsilon_low) * (np.exp(-1.0 * self.steps/
self.decay)) # update epsilon
        if (num < self.epsilon):
            action = torch.randint(0,Q.shape[1],(1,)).type(torch.LongTensor) # take a random action
        else:
            action = torch.argmax(Q,dim=1) # take action with max Q-value

        new_state, reward, done, info = self.env.step((action.item()))
        return new_state, reward, done, info, action

    def update_model(self):
        self.optimizer.zero_grad() # reset gradients to zero (pytorch accumulates graidents otherwise)

        # Get training instances from buffer
        num_tuples_in_buffer = len(self.replay_buffer) # number of tuples stored in buffer so far
        num_samples = np.min([num_tuples_in_buffer,self.batch_size]) # number of samples to optimise with
        samples = random.sample(self.replay_buffer, num_samples) # minibatch of samples for optimisation

        S0, A0, R1, S1, D1 = zip(*samples)
        S0 = torch.tensor( S0, dtype=torch.float) # states at t
        A0 = torch.tensor( A0, dtype=torch.long).view(num_samples, -1) # actions
        R1 = torch.tensor( R1, dtype=torch.float).view(num_samples, -1) # rewards
        S1 = torch.tensor( S1, dtype=torch.float) # states at t+1
        D1 = torch.tensor( D1, dtype=torch.float) # terminal (boolean)

         # Update predictor network in RND module
        implicit_reward = self.rnd.get_reward(S0) # implicit reward
        self.rnd.update(implicit_reward) # optimise RND

        # Update Q network
        target_q = R1.squeeze() + self.gamma*self.target_model(S1).max(dim=1)[0].detach()*(1 - D1)  # target for Q
update
        policy_q = self.model(S0).gather(1,A0) # Q values for all states and actions chosen in sample
        L = F.smooth_l1_loss(policy_q.squeeze(),target_q.squeeze()) # loss function for Q update
        L.backward() # compute gradients
```

```
self.optimizer.step() # update gradients
return L.detach().item() # return Q loss
```