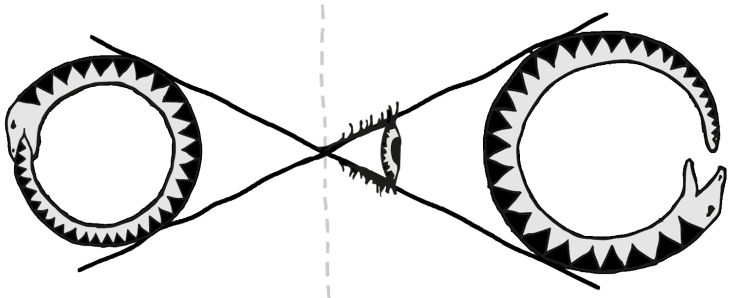
REINFORCEMENT LEARNING AND ARTIFICIAL CURIOSITY

Harry Songhurst

University of Manchester
May 2020



Advisor: Dr. Tingting Mu
Word count: 11,400

Contents

1 Introduction **1**

2 Background **2**

2.1 Markov Decision Processes (MDP's) 2

2.1.1 POMDP 3

2.2 Terminology of Reinforcement Learning 3

2.2.1 Reward Discounting 4

2.2.2 V-functions and Q-functions 5

2.2.3 Bellman Optimality Equation 6

2.3 Types of reinforcement learning algorithms 6

2.3.1 On-policy vs Off-policy 6

2.3.2 Monte Carlo vs Bootstrapping 7

2.3.3 Q-learning 8

2.3.4 Deep Q-learning 8

2.3.5 Policy Gradient Methods 9

2.3.6 Actor-critic methods 11

2.4 Credit Assignment 12

2.4.1 Generalized Advantage Estimate (GAE-λ) 12

2.5 Atari Games 14

2.5.1 OpenAI Gym 15

2.5.2 Wrappers 15

3 Curiosity Driven Learning and Intrinsic Motivation **17**

3.0.1 Count-based methods . . . 18

3.0.2 Next-state prediction . . . 19

3.0.3 Intrinsic Curiosity Module . . . 19

3.0.4 Random Network Distillation . . . 20

3.0.5 Go-Explore . . . 22

3.0.6 Agent57 . . . 23

3.0.7 Meta-Learning Curiosity Algorithms . . . 24

3.1 Limitations . . . 24

4 Experiments **26**

4.1 DQN . . . 28

4.1.1 Results . . . 28

4.2 REINFORCE . . . 28

4.2.1 Results . . . 29

4.3 Maximum Entropy . . . 30

4.4 A2C/A3C . . . 31

4.4.1 Results . . . 32

4.5 Proximal Policy Optimization (PPO) . . . 34

4.5.1 Results . . . 35

4.6 Random Network Distillation (RND) . . . 36

4.6.1 Results . . . 37

4.7 Intrinsic Curiosity Module (ICM) . . . 39

4.7.1 Results . . . 39

4.8 Next-state prediction . . . 40

4.8.1 Auto-Encoder . . . 41

4.8.2 U-Net . . . 41

5 Conclusion **44**

5.1 Reflections 44
5.2 Impact of COVID-19 45
5.3 Future Directions 45
5.4 Summary 45

Bibliography **46**

Chapter 1

Introduction

Reinforcement Learning (RL) is an active area of research in the field of Machine Learning (ML), which seeks to teach virtual agents to perform tasks in environments by rewarding the agent when it does something right and penalising the agent when it does something wrong. RL is a notoriously difficult area of ML, but in the pursuit of generally intelligent systems, it is one of the most promising.

The field had made much progress in the last four years after it received renewed interest in 2015 following DeepMind's publication (Mnih et al., 2015) "Human-level control through deep reinforcement learning". This paper popularised the use of Atari games as a test-bed for studying intelligent behaviour and showed that deep neural networks could solve simple games by just looking at pixels. Despite this progress, there is much dissatisfaction with the brute force nature of reinforcement learning. Humans require minutes, if not seconds, to understand most games, whereas most learning algorithms require hundreds of hours of gameplay to learn a winning strategy (Tsividis, Pouncy, Xu, Tenenbaum, & Gershman, 2017).

This thesis reviews six RL algorithms, two of which deal with a niche of reinforcement learning called "artificial curiosity", which attempt to make an agent behave intelligently in the absence of an extrinsic reward signal (for example, the score when playing a game). Learning with curiosity is appealing because it offers an approach to the sample inefficiency problem. If we can make an agent explore its environment in such a way that it is always encountering things it can learn from, then we should be able to make faster progress. Curiosity also offers a solution to the so-called "credit assignment problem", defined as the difficulty in programming a sensible metric for evaluating whether we're behaving intelligently or not.

Chapter 2

Background

2.1 Markov Decision Processes (MDP's)

Many real-world problems can be formalised as the interaction of an agent with an environment, where the learner/ decision-maker is called the agent, and the environment is the world which the agent lives in and can learn from. An agent can take actions in this world and does so in an effort to receive a reward/achieve a goal. The environment changes in response to the action that the agent takes [2.1]. This differs from standard supervised learning because the input is determined by the agent's actions, whereas with supervised learning the inputs are fixed examples.

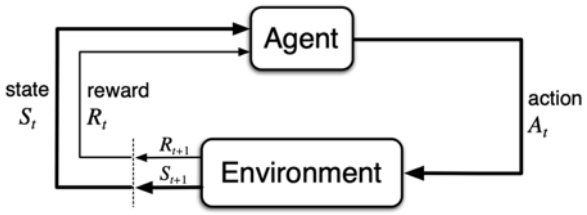


Figure 2.1: Interaction loop in an MDP – Sutton and Barto 1992

This complete process can be mathematically formalized as a "Markov decision process" (MPD) (Bertsekas, n.d.), called as such because the system respects the "Markov property" - all transitions between states depend only on the previous state and action. Mathematically an MDP is a 4-tuple (S, A, R, P), where:

- S is a finite set of states.

- A is a finite set of actions.
- $R : S \times A \to \mathbb{R}$ is the reward function, where $r_t = R(s_t, a_t)$. The reward associated with the current state–action pair.
- $P : S \times A \to \mathcal{P}(S)$ where P is the transition probability function. $\mathcal{P}(S)$ is the powerset of S, and $P(s' \mid s, a)$ is the probability of transitioning from state s into state s' when you take action a.

2.1.1 POMDP

Most interesting RL problems can be interpreted as Partially Observable Markov Decision Processes (POMPD). The distinguishing characteristic of a POMDP is that; whilst what we observe is ultimately determined by an MDP, we cannot directly observe it - it is hidden in some way.

Formally, a POMDP is a 6-tuple (S, A, R, P, Ω, O), where S, A, R, P are states, actions, rewards and transition probabilities as before, only now our agent receives observations $o \in \Omega$ which are selected from a pool of possible observations based on the new state and the action just taken, this observation is chosen according to the probability distribution $O(s', a)$.

To make this concrete, think of the game Atari Breakout. If our agent only observes one frame at a time (and all its decisions are made based on single frames) then it is a POMDP. This is because there is hidden information. Namely, the motion of the ball and paddle. The agent cannot tell which direction the ball is travelling. However, if we stack 4 frames on top of each other, the agent can now infer this motion, and hence we can say the game is an MDP. (Mnih et al., 2015; Hausknecht & Stone, 2017) found that all Atari 2006 games are fully observable MDPs when the states consist of at least 4 frames.

2.2 Terminology of Reinforcement Learning

A *trajectory* $\boldsymbol{\tau}$, also known as a *rollout*, is a sequence of states, actions and rewards $(S_0, A_0, R_0, S_1, A_1, R_1 ...)$. We select actions by following a *policy* $\boldsymbol{\pi_\theta}$, a function which maps states to actions $(\pi_\theta : S \to A)$. This function depends on some parameters $\boldsymbol{\theta}$, the modification of which leads to a change in the mapping $S \to A$. The policy can be either deterministic or stochastic. If it is deterministic, $\pi(S_t)$ maps directly to an action A_t. If it is stochastic, $\pi(S_t)$ maps to a probability distribution (a vector of real numbers that sum to one) over all possible actions, and an action is chosen randomly from this distribution.

This thesis is only concerned with policies in the form of deep neural networks.

The *return* $\boldsymbol{R(\tau)} = \sum_{t=0}^{T} r_t$ associated with a trajectory is a sum of all the rewards received over the trajectory. We say that the return is *finite horizon* if τ is finite, else it is *infinite horizon*.

With the former in mind, the objective of reinforcement learning is to discover the optimal policy $\boldsymbol{\pi^*}$ which maximises the expected return over any trajectory:

$$\boldsymbol{\pi^*} = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[R(\tau)\right]$$

The $\tau \sim \pi$ subscript means "over the trajectory, with respect to π". Hence, you start in state s_0, and select all actions with respect to the policy π, where π is selected from all possible π's such that it always maximises the expected return over the whole trajectory (regardless of the specific trajectory).

2.2.1 Reward Discounting

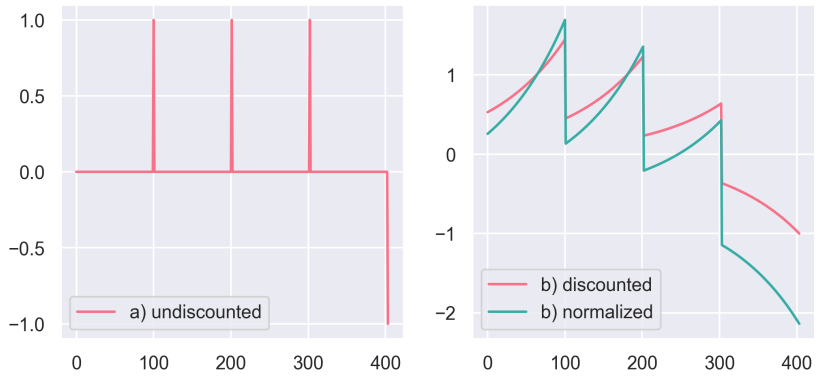


Figure 2.2: A: Undiscounted (raw) vs discounted rewards.

It is common to "discount" our rewards, which means that we distribute any rewards received across all transitions in a trajectory. This is demonstrated in 2.2. We do this because the exact point at which we receive a reward may not correspond to the exact action that led to it, rather it is more likely that an entire sequence of actions led to that reward, so we crudely distribute this reward across all the actions leading up to the reward spike, in the hopes that we can reinforce the general behaviour that leads to rewards.

2.2.2 V-functions and Q-functions

So now we know the problem that we want to solve. We want to find the optimal policy, $\boldsymbol{\pi^*}$, that maximises the expected return over any trajectory. To this end, it is necessary to introduce two functions, the *V-function* and *Q-function*, that allow us to determine how good any particular state or action is for the agent.

The state-value function, or V-function is defined thus:

$$\boldsymbol{V}^{\pi}(s) = \mathbb{E}_{\pi}\left\{\boldsymbol{G}_t | s_t = s\right\} \tag{2.1}$$

$$= \mathbb{E}\left[\sum_{t}^{T} \gamma^t r_t\right] \tag{2.2}$$

It tells us how good any particular state is, as measured by the expected discounted reward $\boldsymbol{G_t} = \sum_t^T \gamma^t r_t$ we receive by the end of the trajectory when starting from state s.

The action-value function, or Q-function is defined thus:

$$\boldsymbol{Q}^{\pi}(s,a) = \mathbb{E}_{\pi}\left\{\boldsymbol{G}_t | s_t = s, a_t = a\right\} \tag{2.3}$$

$$= \mathbb{E}_{\pi}\left[r_{t+1} + \gamma V\left(s_{t+1}\right)\right] \tag{2.4}$$

It tells us how good any available action is from any state, given that after taking the action we will take all subsequent actions by consulting our policy π. Q stands for quality.

Notice that these two functions can look very similar. The important difference is that the value function is the expectation/average over all available actions from state s_t following π, whereas the Q-function says that we must take a particular action a_t, then follow π.

Both the state-value and action-value function are said to be *optimal* when they are associated with an optimal policy.

$$\boldsymbol{V}^*(s) = \max_{\pi} \boldsymbol{V}^{\pi}(s) \quad \text{and} \quad \boldsymbol{Q}^*(s,a) = \max_{\pi} \boldsymbol{Q}^{\pi}(s,a)$$

The *optimal state-value function*, $\boldsymbol{V}^*$ tells us the maximum possible return from state s, for all $s \in S$. The *optimal action-value function*, $\boldsymbol{Q}^*$ tells us the maximum possible return from state s after taking action a, for all $s \in S$ and all $a \in A$.

2.2.3 Bellman Optimality Equation

With these definitions in place, we can introduce one of the most important equations in reinforcement learning, the Bellman optimality equation.

$$\boldsymbol{Q}^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} \boldsymbol{Q}^*\left(s', a'\right)\right] \tag{2.5}$$

This equation states that the value of taking action a in state s is equal to the expected reward received in the next state R_{t+1} (resulting directly from our chosen action), plus the cumulative *discounted* reward received after this by taking all subsequent actions following the optimal policy (keep playing until the end of the game, taking actions that leads to the maximum reward in each state). In this way, the equation is self-consistent, it is recursive.

2.3 Types of reinforcement learning algorithms

In normal supervised learning problems, we seek to maximise the probability of observing the correct labels y, given some inputs x (an image, for example). In reinforcement learning we do not have access to a big dataset of training pairs $[(a_t, s_t)...]$, describing the best action to take given some input state. Instead, reinforcement learning algorithms allow us to generate our own "dataset"; we consult our policy which tells us an action to take a given state policy $a_t \sim \pi_\theta(s_t)$, and receive some reward from the environment for doing so.

We can use these transitions/interactions to discover better policies, and reinforcement learning algorithms allow us to do just this. These algorithms often belong to one of two camps; policy gradient methods, or Q-leaning methods. They can also be classified as on-policy or off-policy, Monte Carlo or bootstrapping.

2.3.1 On-policy vs Off-policy

The distinction between on-policy and off-policy RL algorithms is concerned with whether the policy used to generate actions, called the *behaviour policy*, is the same policy that we update whilst learning, called the *target policy*.

Off-policy methods behave according to one policy, whilst learning a different target policy. An example, discussed in 2.3.3, is Q-learning, where the target policy assumes we're behaving 100% greedily. However, the policy that we're behaving with in the environment,

might only be 80% greedy. That is, we select our action to be $a = \max_a Q(S, a)$ 80% of the time, but our update assumes that was the case *all* of the time. The advantage of this is that we can learn a 100% greedy policy, but afford ourselves the ability to explore the environment by occasionally taking random actions.

On-policy methods, on the other hand, are methods where the behaviour policy and the target policy are the same. We collect samples using the same policy that we update, and act using the same rule that our update rule assumes we use to act. Most policy gradient methods, discussed in 2.3.5, and actor-critic methods, discussed in 2.3.6, are on-policy methods.

As a rule of thumb, algorithms which store a "replay buffer" of transitions made in the past (using an old policy) are generally off-policy. Whereas algorithms that only update the policy using transitions made with the most recent rollout are on-policy. There are exceptions though, we can make policy gradient algorithms off-policy by incorporating importance sampling (Degris, White, & Sutton, 2013).

2.3.2 Monte Carlo vs Bootstrapping

Monte Carlo methods broadly refer to any algorithms which rely on random sampling to obtain numerical results; for example, to use random exploration and sampling to learn about perhaps deterministic environments (Wikipedia, 2020).

However, Sutton and Barto (Sutton & Barto, 1998) extend this definition in a way that is useful for distinguishing between modern RL algorithms. They define Monte Carlo methods explicitly as those which learn from averaged, *complete returns*. This distinction is important because many actor-critic (2.3.6) algorithms learn from only *partial returns*.

A Monte Carlo method may run through 200 time-steps until a game ends, storing the transitions and rewards encountered, then update the policy using these returns. Non-Monte Carlo methods, on the other hand, *bootstrap* an estimate of the expected reward for some much smaller number of time-steps (e.g. 5). What this means is that we have some function, usually a neural network called "the critic", that predicts the reward the agent will receive for the last 5 time-steps, and we use this prediction to calculate the actor policy update.

The term "bootstrapping" means just this - pretending that a predicted value is ground truth, and using it to update some other value/function. The "critic" in actor-critic architectures, is the thing we bootstrap from; it takes as input the state of the environment and outputs the expected reward for this state.

2.3.3 Q-learning

Q-learning is an off-policy algorithm which finds the long term expected reward (Q-value) associated with each state-action pair in some MDP. Traditionally this is accomplished by instantiating a two-dimensional table, called a "Q-table", that associates a learned Q-value with each state-action pair.

We initialize our Q-table randomly, or with all zeros. The Q-learning algorithm then iteratively updates the Q-values for each state-action pair, until we converge on the optimal Q-function $\boldsymbol{Q}^*$ as defined by 2.2.3. It does this by taking an action A in the world from state S , receiving a reward, and arriving in the next state S'. The action selected is sometimes (20% of the time) random, but usually the action with the highest Q-value as described by the Q-table. The following update rule is then used to update the Q-value associated with the state action pair (S, A):

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q\left(S', a\right) - Q(S, A) \right] \tag{2.6}$$

Intuitively, this rule says that the Q-value associated with the state-action pair we just encountered gets updated to its previous value plus some small (learning rate, α) multiplier of the reward received and the Q-value of the state we land in after taking the action, where that Q-value of that next state is given by the Q-table.

2.3.4 Deep Q-learning

Regular Q-learning only works for MDP's which are small enough to enumerate all possible state-action pairs of. This is not the case for most interesting environments. The state-space of environments such as Atari games is massive (all possible combinations of pixels). In 2015 (Mnih et al., 2015) introduced deep-Q-networks (DQN) that allow us to use Q learning to solve these more complicated environments. Instead of an explicit Q-table mapping state-action pairs to Q-values, we can approximate a Q-table using a neural network. The input to this neural network is states from the environment, and the output is the expected reward after taking action a then following the same policy until the end of the episode (the Q-value) (2.3 left).

We can use a very similar algorithm to that above in order to update our DQN, to more accurately predict the Q-values of each state, however, in this case, we make use of the bellman equation for optimality 2.2.3. We take the mean squared error between the value outputted

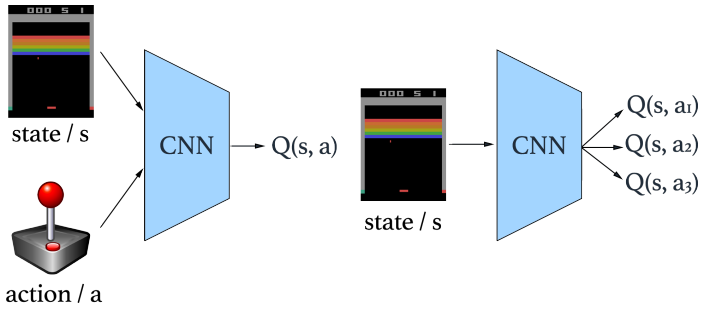


Figure 2.3: Deep Q-learning formulations

by our network and the target value given by the Bellman equation and backpropagate this loss through the network to make weight updates.

$$\hat{\boldsymbol{Q}}^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} \boldsymbol{Q}^*\left(s', a'\right)\right] \tag{2.7}$$

$$\boldsymbol{Q}(s,a) = \text{DQN Output} \tag{2.8}$$

$$\text{loss} = ||\hat{\boldsymbol{Q}}^* - \boldsymbol{Q}||^2 \tag{2.9}$$

The actual DQN algorithm, expanded on in the experiments section 4.1, is slightly different. In practice, we set up the DQN to output Q-values for all possible actions given the input state, instead of forwarding the network A times, where A is the number of possible actions (2.3 right). We then back-propagate through the neuron of our chosen action. This saves on compute by a factor of the number of actions. Another big difference is that we maintain a buffer of past transitions (S, A, R, S') from which we sample random batches to make weight updates This means that DQN is off-policy, because we can learn from transitions that were made using an old policy.

2.3.5 Policy Gradient Methods

Policy gradient methods are another type of RL algorithm. They are on-policy, meaning that we behave using the same policy that we update/learn. Our policy π_θ is a neural network that outputs a probability distribution over the actions available to us, and is fully differentiable with respect to its inputs, states from the environment. At each step we randomly sample an action from the output distribution. We would like to encourage actions that lead to rewards, and discourage actions that lead to penalties. We can do this by reinforcing our policy parameters θ if the randomly sampled action we took led to a

reward.

More formally, we can state the problem of taking actions that lead to rewards as the problem of maximising the objective function $J = \mathbb{E}\left[\sum_{t=0}^{T} R_t\right]$ (the expected return over some trajectory). We can use gradient ascent to perform this maximisation, which motivates the following derivation of the *policy gradient* $\nabla_\theta J$ - a high variance estimate of the affect our policy parameters θ have on W. First, recall that expectation is defined as a weighted average of events given their probabilities $\mathbb{E}[f(x)] = \sum_x f(x)p(x)$. Since our policy π_θ outputs probabilities over actions, we can substitute $\pi_\theta(a_t|s_t)$ for $p(x)$ and the reward function $R(a_t, s_t) = r_t$ for $f(x)$.

$$\nabla_\theta J = \nabla_\theta \sum_{a_t} r_t \pi_\theta\left(a_t|s_t\right) \qquad \textit{def. expectation} \tag{2.10}$$

$$= \sum_{a_t} r_t \nabla_\theta \pi_\theta\left(a_t|s_t\right) \qquad \textit{sum rule} \tag{2.11}$$

$$= \sum_{a_t} r_t \frac{\pi_\theta\left(a_t|s_t\right)}{\pi_\theta\left(a_t|s_t\right)} \nabla_\theta \pi_\theta\left(a_t|s_t\right) \qquad \times \frac{\pi_\theta\left(a_t|s_t\right)}{\pi_\theta\left(a_t|s_t\right)} \tag{2.12}$$

$$= \sum_{a_t} r_t \pi_\theta\left(a_t|s_t\right) \nabla_\theta \ln \pi_\theta\left(a_t|s_t\right) \qquad \nabla \ln f = \frac{\nabla f}{f} \tag{2.13}$$

$$= \mathbb{E}_{\pi_\theta}\left[r_t \nabla_\theta \ln \pi_\theta\left(a_t|s_t\right)\right] \qquad \textit{def. expectation} \tag{2.14}$$

The policy gradient tells us to update our policy parameters in accordance with:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha r_t \frac{\nabla \pi_\theta\left(a_t|s_t\right)}{\pi_\theta\left(a_t|s_t\right)} \tag{2.15}$$

This is intuitively satisfying because the numerator is the direction in the parameter space that most increases the probability of taking action a_t given state s_t. Multiplying this direction with the return means we decrease the action probability when the return is negative. The denominator makes updates inversely proportional to the magnitude of the action probability, which is good because otherwise actions with high probability would be disproportionately encouraged relative to actions that were less probable (but may have led to a higher reward) (Sutton & Barto, 1998).

2.3.6 Actor-critic methods

Policy gradient methods tend to be very unstable. This is because we always compute the policy gradient estimate using the latest batch of experience sampled from the environment. This latest batch is the direct result of our own actions, and because our actions are somewhat random, the samples we get back are highly variable. In other words, the policy gradient estimate we compute over some batch of trajectories is not the true policy gradient; but as out rollout length tends to infinity, our average policy gradient would approach the true policy gradient. Obviously, this is computationally infeasible.

In other words, the policy gradient has *high variance*. Because our trajectories are the function of probabilistic choices, one batch of trajectories may be radically different from another. Even transitions in the same batch may be very different from one another. This high variance means that we may be tugging the policy parameters all over the place with each weight update, leading to inefficient and sometimes degenerate learning (degenerate if we update the policy such that we never sample any interesting data again due to always choosing bad actions).

Reducing the variance of our policy gradient results in more stable and efficient learning. A common approach doing this is to subtract a baseline b from the reward (Williams, 1992), the baseline is also known as the critic:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} \left[(r_t - b) \nabla_\theta \ln \pi_\theta \left(a_t | s_t \right) \right] \tag{2.16}$$

This baseline b may be chosen as the average of reward over a batch of trajectories. This would mean that we end up encouraging roughly half the actions in each batch (actions that led to higher than average reward), and discouraging roughly half also (actions that led to lower than average reward). We hope that actions that genuinely lead to high reward are in the half that gets encouraged. This does improve learning, however, we can do better...

Ideally we would like to only encourage actions if they led to a higher than average reward compared to the reward we usually receive in the particular state we took it from. To this end, we can make use of the V-function and Q-function introduced in 2.2.2, to define the advantage function:

$$\hat{A}(a_t, s_t) = \hat{Q}(a_t, s_t) - \hat{V}(s_t) \tag{2.17}$$
$$= r_t + \gamma \hat{V}(s_{t+1}) - \hat{V}(s_t) \tag{2.18}$$

Expanding out both of the V-function terms leaves you with the advantage being equal to the reward received for performing action a_t in state s_t, minus the expected value for state s_t over all possible actions.

We don't have access to the true value function. Instead, we introduce another neural network which learns an approximation of it. We train it to predict the (discounted) reward we would usually receive in any given state, which allows us to then consult it as a substitute for $V(\cdot)$ and calculate the advantage of each transition. We hope that this neural network will generalize, such that it accurately predicts the value of *similar* states that it may have not been explicitly trained on.

Putting these things together gives a fairly robust learning algorithm. We collect experience from the world by sampling actions from the actor, discount the rewards we received, then subtract from those rewards the estimated value of the experiences predicted by the critic. This gives us a policy gradient that points in the direction of "actions which lead to higher than expected reward", and can be used to find very good behavioural policies.

2.4 Credit Assignment

We have already touched on reward discounting as a technique for distributing rewards to actions in a rollout 2.2. This is one of the simplest *credit assignment* schemes.

The actor-critic advantage function takes this this a step further by subtracting a baseline from each discounted reward. This results in lower variance. As we do not know $Q(S, A)$ or $V(S)$, we estimate the this baseline $V(S)$ using a "critic" neural network giving the final advantage 2.17. This is illustrated in 2.4 (middle).

2.4.1 Generalized Advantage Estimate (GAE-λ)

(Schulman, Moritz, Levine, Jordan, & Abbeel, 2018) introduces the "generalized advantage estimate" which is an even lower variance advantage function, which makes learning more stable. It is currently the most widely used credit assignment regime, and is described by the following equation:

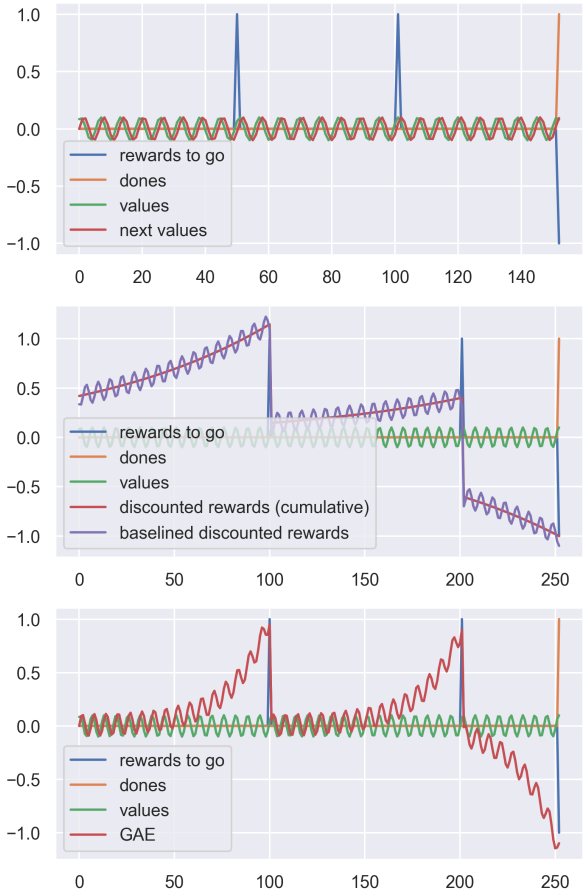


Figure 2.4: Credit assignment schemes. A: undiscounted "raw" rewards. B: actor critic advantage. C: generalized advantage estimate.

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.19}$$

$$\text{where } \delta_t = r_t + \gamma\hat{V}\left(s_{t+1}\right) - \hat{V}\left(s_t\right) \tag{2.20}$$

The intuition is that we take the advantage function from 2.17 and introduce a parameter λ which controls how much we care about earlier, more accurate, estimates of the advantage function. Notice that, because $\hat{A}_t$ is a sum of uncertain terms, it has high variance. λ trades off some of this variance by decreasing the weight of distant advantage estimates, in favour of bias towards earlier advantage estimates. If we choose λ to be 0, then $\hat{A}_t = \delta_t$ as in 2.17, and we have higher variance. As we increase λ, variance decreases for bias towards early advantage estimates. In practice, lambda is usually chosen to be about 0.95.

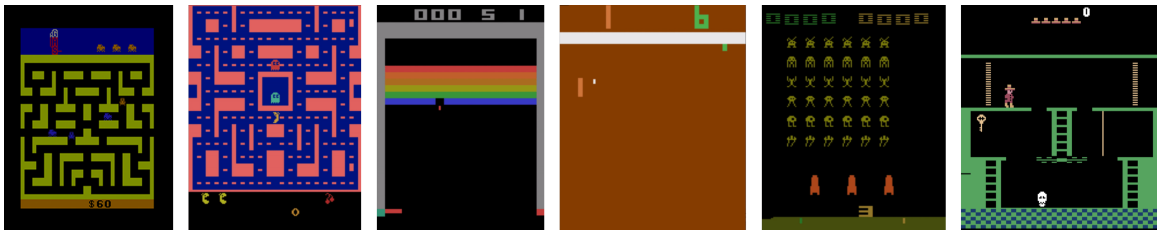


Figure 2.5: Bank Heist, Pacman, Breakout, Pong, SpaceInvaders and Montezumas Revenge.

2.5 Atari Games

Known as the "Arcade Learning Environment (ALE)", Atari 2006 games have served as a benchmark in the reinforcement learning literature since 2015. Games are an interesting thing to solve because they are often designed to be difficult for humans to master, and can sometimes mimic some interesting aspect of the real world. However, a lot of RL research focuses on beating "easy" Atari games - the ones which have a "dense reward signal", these are games such as Breakout and Pong where you get a point (reward) every 20 frames or so. This means that the agent doesn't have to learn any sort of long term plan or do any complex reasoning. It can rely on taking random actions to discover favourable behaviour. This makes these algorithms much less impressive.

With that said, there are a few games in the collection of 57 Atari games that are much more difficult to master. Games which are classed as "hard exploration". These are games where the agent may only receive a reward after 1000 frames. This means it is unlikely to discover a good behavioural policy by chance. Games in this class include Montezuma's Revenge, Pitfall, Solaris and Skiing. The work in my project focuses on 6 ALE games; Bank Heist, Pacman, Breakout, Pong, SpaceInvaders and Montezuma's Revenge 2.5, and the simple CartPole environment.

Ultimately we're not just trying to solve Atari games. Single systems that solve many games are a stepping stone to systems that can learn generally intelligent behaviour regardless of the environment. In fact, it seems likely that the field will move away from the arcade learning environment, towards procedurally generated games; where it is impossible to memorize specific trajectories or pixel patterns. Instead, procedurally generated environments systematically force the agent to understand the underlying idea behind what it's trying to achieve (Cobbe, Hesse, Hilton, & Schulman, 2019).

2.5.1 OpenAI Gym

OpenAI developed and maintain a Python package called Gym that makes it very easy to perform reinforcement learning experiments. We can use Gym to instantiate an Atari game environment by calling gym.make([game][mode]-[repeat_action_probability]).

- [game] represents the game we would like to play, for example, Breakout, or Pong.
- [mode] is either omitted, Deterministic, or NoFrameskip. This controls the "frame skipping", that is, the number of frames that the emulator runs our given action for per step. In [game]Deterministic, the emulator will always run our given action for a fixed number of frames (usually 4) before returning the new state. In [game]NoFrameskip the emulator will only run our action for one frame. And if the [mode] parameter is omitted, the frame skipping is stochastic; between two and five frames executed with the given action for each step. The reason for this is to speed up games and introduce some stochasticity into otherwise deterministic environments.
- [repeat_action_probability] is either v0 or v4. This parameter controls the probability that a step (*not* frame) ignores the given action a_t, and instead plays the previous action a_{t-1}. If v0 this happens with probability 0.25. If v4 the probability is 0.0, meaning the agent always performs the given action.

Each gym environment has an "observation space" and a "action space". If the action space is of type Discreet(n) then we are dealing with n mutually exclusive actions; we can only choose one of the n actions at each step, for example, GO LEFT. If the action space is of type Box(n) then the action space is a tuple of n real values, for example a valid action may be a 3-tuple $(0.4, 0.0, 0.6)$ and may represent (gas, brakes, steering angle).

2.5.2 Wrappers

DeepMind, OpenAI and Stable-Baselines (Hill et al., 2018) provide a collection of "wrappers" that modify gym environments in useful ways, making them easier to experiment with.

VecEnv

The frames of a single video game episode are likely all very similar to one another (one frame may differ from another only by a few pixels). In other words, they are highly correlated.

Data/feature correlation is a problem in machine learning generally. It is desirable in most machine learning tasks for training batches to be independent and identically distributed (IID). If we perform a weight update on a batch that is highly self-correlated, there just isn't that much our network can learn from. It can be very difficult for our network to figure out what in the sequence of observations was the predictor of success, and indeed, a single rollout may contain a sort of false-positive where we got a spurious reward for sub-optimal behaviour.

One way to prevent this is to run many parallel environments at once, e.g. 16 environments, and average weight updates across them. Doing this means that we should more frequently observe the "average" behaviours that lead to rewards. To this end, we can make use of the `VecEnv` wrapper, which spins up multiple parallel environments and allows you to step through all of them at once with a vector of actions instead of a single action. This utility makes use of OpenMPI to distribute environments over multiple CPU cores; however, I didn't manage to get this working, so just relied on Python's in-built multiprocessing.

FrameStack

The `FrameStack` wrapper simply stacks the last four environment frames on top of each other for each returned observation. This is necessary for capturing motion information; a single frame of Pong does not tell you enough information about which way the ball is flying to accurately play the game, but the last four do. This was first talked about in (Mnih et al., 2015).

Normalize

This wrapper maintains a running mean and standard deviation of previously observed environment frames. It subtracts this mean from each observation, then divides by the standard deviation. I use this in my implementation of Random Network Distillation 3.0.4.

Chapter 3

Curiosity Driven Learning and Intrinsic Motivation

This section introduces curiosity-driven learning and reviews a few existing approaches from the literature.

Different types of motivation have been studied in psychology since the 1970s (Ryan & Deci, 2000). An interesting question researched by many psychologists is why children engage in behaviour such as stacking blocks, or repeatedly throwing a toy at the ground and picking it up again. Psychologists study these phenomena through the lens of motivation and make the distinction between extrinsic and intrinsic motivation. If behaviour is extrinsically motivated, it is directed at gaining external rewards and avoiding external punishment. For example, in an Atari game, extrinsically motivated behaviour would be that which increases the game score and avoids losing lives. Intrinsic motivation, on the other hand, is not concerned with maximising the score or levelling up, but instead, intrinsic motivation occurs when we act because we simply enjoy an activity, find it interesting, or see it as an opportunity to explore and learn (Coon & Mitterer, 2007). It is highly desirable to develop algorithms that learn from intrinsic motives, because there are often no clear extrinsic motivations to human behaviour, especially during childhood development. In a very real sense we "learn our own reward functions". It seems reasonable to suggest that our faculty of intrinsic motivation is very well evolved, and is fundamental in directing our interest and attention.

Motivated by the desire to explore an environment with sparse extrinsic rewards (hard exploration games), we would like some metric which informs us when a phenomenon is interesting, what doesn't rely on the game score. We then want to reward our agent with a

"bonus" proportional to this metric, therefore encouraging it to explore interesting things further. But how do we quantify whether something is interesting?

Schmidhuber (J. Schmidhuber, 2009) says that only data with still unknown but *learnable* statistical or algorithmic regularities is truly interesting and thus deserves attention. (Oudeyer & Smith, 2016) say we should be "engaged by what is just beyond current knowledge, neither too well known nor too far beyond what is understandable". Many curiosity metrics (aka intrinsic bonuses) rely on a self-supervised process of predicting something in the future, then observing "reality" and calculating a reward based on the difference between this prediction and what actually happened. In this way, our actions are guided by stimuli that allow us to predict the future.

However, it is possible for something to be unpredictable and new, but at the same time wholly inconsequential and uninteresting. Take, for example, white noise on a TV screen; if we were not careful in our definition of curiosity, we might build an agent that fixates on this random noise because it is a source of infinite new states/novel information. But this is undesirable because there is nothing to learn from a noisy TV screen. What we want instead is to be curious about states which are both new and learnable, that is, states which we can represent, compress and build predictive models of. With this, an agent can be coaxed into playing with new parts of the environment, with the promise that by doing so, it will be able to learn about and master something new. Of course, this is easier said than done, and indeed, much of reinforcement learning is concerned with this problem of writing smart reward functions such that our agents 'care' more about the learning process and aren't just accumulating reward through brute-force.

To better define the problem, (Pathak, Agrawal, Efros, & Darrell, 2017) identify three different types of environmental change that may occur: (1) changes that the agent is in control of, (2) changes the agent is not in control of, but are important (e.g. the ball moving in pong) and (3) changes the agent is not in control of, that are inconsequential (e.g. white noise on a TV screen). We want an intrinsic bonus that is influenced by (1) and (2) but not (3). There are many suggestions for how to do this, but it seems that no technique yet devised is particularly satisfying.

3.0.1 Count-based methods

A simple suggestion (Strehl & Littman, 2008) is to reward an agent when it discovers some state of the environment not yet visited. We could formally define an intrinsic bonus as a function of the visitation count $n(S_t)$ of each state. These are called count-based bonuses; $1/n(S_t)$ is an example. However, count-based bonuses are only really useful in constrained

MDPs that are fully specified (we have a table of all the possible states), allowing us to keep track of how many times we've visited each state. But we are interested in solving more complicated, unbounded problems, and besides, approaches relying on counting states are highly susceptible to the noisy TV problem mentioned earlier because we will associate a visitation count of 1 with each unique frame on the TV, and thus receive a relatively high bonus for each TV frame, meaning we end up being inappropriately curious.

3.0.2 Next-state prediction

Early suggestions of Schmidhuber (J. U. Schmidhuber, 1991) proposed an intrinsic bonus called 'adaptive curiosity' based on next-state prediction. The idea is that we have two different neural networks, one is our behaviour policy, and the other is a "world model"; a neural net that predicts the next state $\hat{S}_{t+1}$ of the environment given the current state S_t, and the planned action a_t. We then reward the agent with a bonus proportional to the mean squared error between the predicted state and the actual observed state (reality) $\sum (S_{t+1} - \hat{S}_{t+1})^2$. Any sensible agent will now take actions that tend to increase this error - putting the agent in states from which the world model struggles to predict the next state.

The world model is continuously updated so that it becomes better at predicting any part of the environment it was previously unfamiliar with. However, we can see that the problem of dealing with truly stochastic parts of the environment remains; the world model will never learn to predict any part of the environment that is a true source of randomness, and therefore our policy will collapse and fixate on any true source of randomness since it provides constant intrinsic reward. For this reason, this bonus only works in deterministic environments.

Methods that use next-state prediction are very attractive from a 'biological plausibility' standpoint. There is a well-established theory in neuroscience called predictive coding (Brodski, Paasch, Helbling, & Wibral, 2015) which suggests that the brain is constantly generating predictions of its next sensory inputs, and that these predictions are compared to actual sensory inputs to give prediction errors that are used to make the predictive model better. It seems reasonable to suggest that this self-supervised process of "predicting the future" is one of the primary learning signals in the brain.

3.0.3 Intrinsic Curiosity Module

A suggestion very similar in spirit to next-state prediction is at the core of the "Intrinsic Curiosity Module" (ICM) proposed by (Pathak et al., 2017). This approach, however,

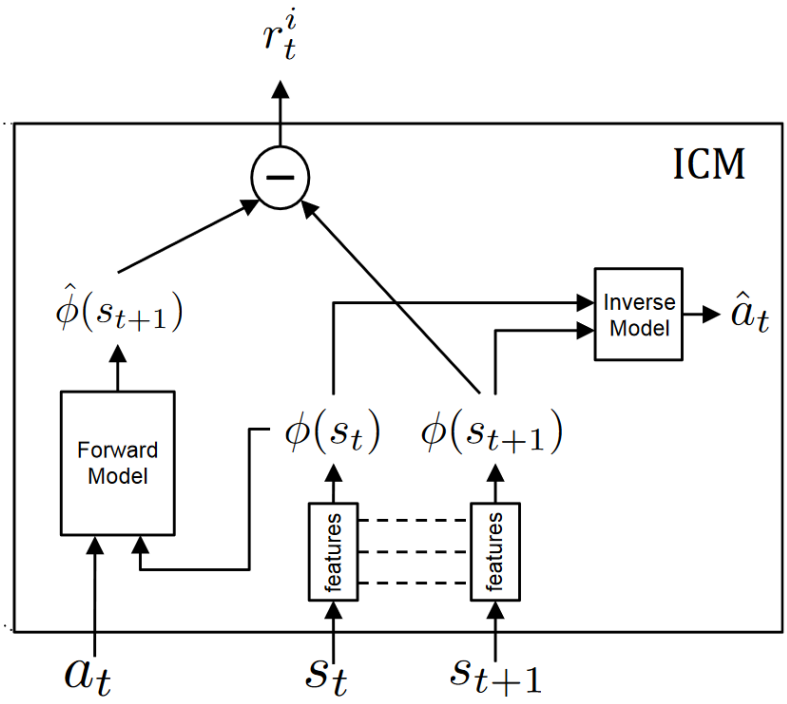


Figure 3.1: Intrinsic Curiosity Module (Pathak et al., 2017)

relies on predicting the action a_t that was taken, given latent encodings of the pair of states (S_t, S_{t+1}). The latent encodings (ϕ in 3.1) are 128-dimensional vectors that are trained such that they encode the features of the environment that are most relevant to predicting which action was taken. The authors justify this approach by arguing that this makes the agent focus more on changes in the environment that are deterministic with respect to its own actions, that is, changes that it is in control of and can (hopefully) learn from. Consequently, this somewhat mitigates the "curiosity trap" mentioned before (where the agent fixates on a source of true randomness) as we care less about things we can't control. Also, it stands to reason that our actions have a relatively predictable effect on the environment.

Pseudocode for calculating the ICM bonus is given in my experiments section 5.

3.0.4 Random Network Distillation

Random network distillation, introduced by OpenAI in 2018 (Burda, Edwards, Storkey, & Klimov, 2018) offers yet another slightly different intrinsic bonus. Instead of deriving a bonus from the ability to predict how the pixels of the environment changes with respect to our own actions (as with next-state prediction), we instead derive a bonus from the ability

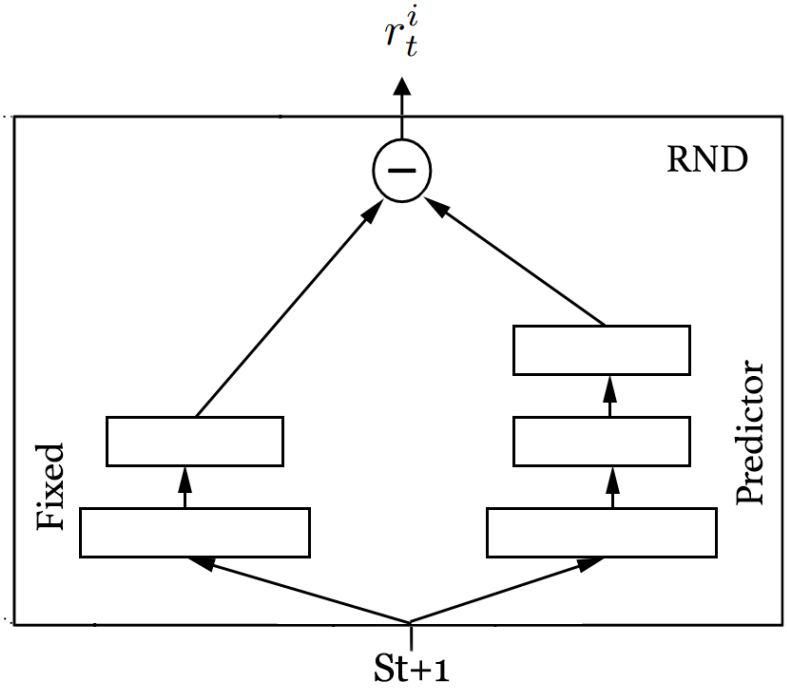


Figure 3.2: Random network distillation

of one neural network to predict the output of another neural network, giving the one S_t and the other S_{t+1}, where the output is, for example, a 128-dimensional vector. Importantly, we only train one of these networks, whereas the other is fixed for the lifetime of the agent. The idea being that if the predictor network can accurately predict the output of the fixed network, it is likely that the agent has already explored the part of the environment it is currently in, so there is nothing left to learn from it. In this case, the bonus calculated will be low. However, if the predictor network cannot predict the output of the fixed network, the prediction error will be high, and therefore so will the intrinsic bonus, hence, encouraging exploration.

RND mitigates the 'noisy TV problem' somewhat as well, because the 128-dimensional output is somewhat stable in the face of minor fluctuations in the input. Buried away in this paper, however, is the admission that it will not solve the noisy TV problem if the TV is a truly stochastic source of infinitely many of RGB values.

Again, pseudocode and implementation details for computing the RND bonus are given in my experiments 4.6.

RND Prioritized Oversampled Experience Replay

All methods discussed so far are very sample inefficient. In order for something to become "uninteresting" from the perspective of RND, the "random network" must distil; that is, the neural network must learn to predict what is going to happen in the environment accurately. This can take millions of frames of experience (years of gameplay at 30 FPS) and hundreds, if not thousands of weight updates. This means solving a game such as Montezumas Revenge is very inefficient.

A simple extension to RND proposed by (Sovrano, 2019) reduces the amount of time needed to solve games by incorporating "Prioritized Oversampled Experience Replay" (POER). This is essentially just storing a buffer of experiences/states that were associated with a high intrinsic or extrinsic reward, and incorporating the old samples from this buffer into each new training batch ("replaying" the old samples), where we are more likely to include (prioritize) experiences which were associated with higher intrinsic or extrinsic reward.

There are a few implementation details that must be taken care of to allow policy gradient methods such as RND to make use of old experiences collected with an old version of our policy - we have to turn it into an off-policy algorithm 2.3.1. Ultimately, this amounts to scaling our weight updates, for some sample, by the ratio between the old probability of the action (computed with the old policy) and the new probability of the action (computed with the new policy) (*Policy Gradient Algorithms*, 2018).

3.0.5 Go-Explore

(Ecoffet, Huizinga, Lehman, Stanley, & Clune, 2019) theorise that one problem with the aforementioned intrinsic motivation approaches is, what they call, "detachment". This is the phenomenon by which an agent becomes familiar with some part of the environment, and therefore loses interest in it. However, if it had persisted in exploring that part of the environment a while longer it would have encountered the reward. It becomes "detached" from the rest of the environment because it is too familiar with the space between where it is currently, and where the reward lies. The problem is illustrated in 3.3.

The algorithm they propose to solve this problem relies on caching the emulator state (literally the state of the RAM), then revisiting any states which had high reward associate with them until they stop seeing progress; at which point another emulator state is loaded and the process repeats. Whilst this algorithm solves the games very well, it has been criticised because it seems more like a brute force game solver than anything intelligent. It is contingent on being able to use a deterministic emulator that is fully observable and

restorable (through RAM). The real world is not like this. While the detachment problem is interesting and valid, the solution to it they propose is not.

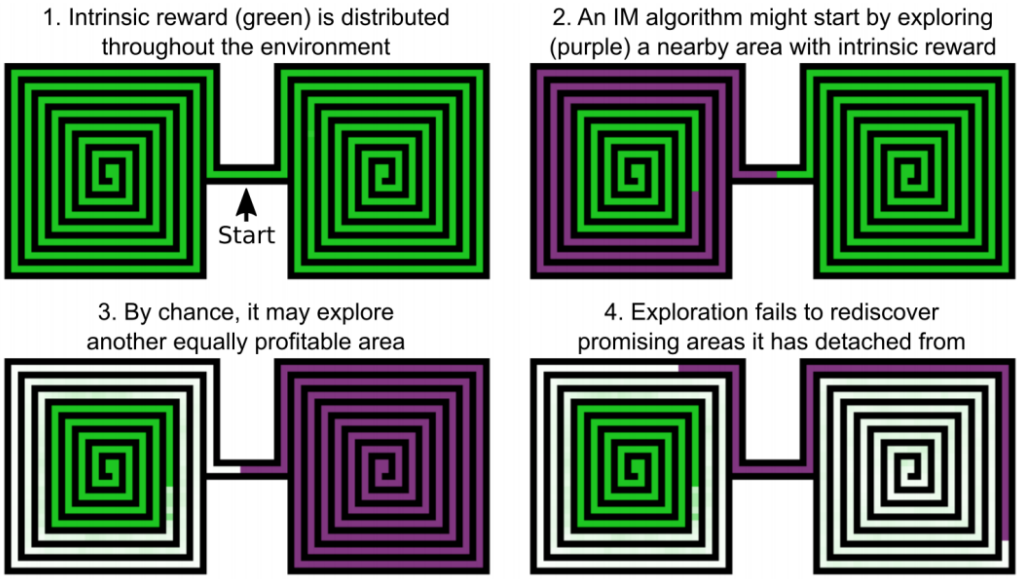


Figure 3.3: Detachment (Ecoffet et al., 2019)

First Return Then Explore

The criticisms of GoExplore are addressed in a recent paper (Ecoffet, Huizinga, Lehman, Stanley, & Clune, 2020) by the same authors. In this work, they do not rely on resetting the emulator to high-potential states. Instead, they "learn to return" to the high-potential state by maintaining a buffer of transitions leading to them, then condition a policy on relevant items in this buffer.

3.0.6 Agent57

(Badia et al., 2020) is the first to outperform humans on *all* 57 Atari 2006 games using the same algorithm (this is *not* multi-task learning, they still train on each game separately). Thought it solves every game, it still requires a lot of training for hard exploration games such as Skiing. Consequently, they train on each game for 100 billion frames (equivalent to $\tilde{1}06$ years of constant gameplay at 30 FPS).

Agent57 is quite complicated, so this summary won't do it complete justice. With that being said, it uses RND as its source of intrinsic motivation/curiosity. However, it introduces

an additional contribution to RND, which allows it to perform better. This contribution stems from the observation that some games are easy to solve by exploiting a strategy that is discoverable almost immediately (Pong is an example of this sort of game), however, there are some games where it helps to be far more curious, such as Pitfall or Montezumas Revenge. This realisation led them to incorporate a *meta-controller*, which dynamically controls the trade-off between exploration and exploitation. This meta-controller is itself a neural network, which outputs a "strategy", e.g. [20% explore, 80% exploit], which in turn controls the reward being maximised (0.2×intrinsic reward + 0.8×extrinsic reward)

They utilise one further development which I didn't explore in-depth in my project - short term memory. This was first discussed for reinforcement learning in (Hausknecht & Stone, 2017). The input state (screen pixels) is initially processed by a convolutional "torso" which compresses the state into a latent space of 512 units. These 512 units then feed into LSTM (Hochreiter & Schmidhuber, 1997) cells, which allows for information from the past to be registered with the current state. The output of these cells are then fed to the actor network which output action Q values, where this actor is trying to maximise both the extrinsic and intrinsic reward gained, as prescribed by the meta-controller.

3.0.7 Meta-Learning Curiosity Algorithms

Meta-learning is an interesting area of research which attempts to automatically discover neural network architectures, learning algorithms and programs generally. (Alet, Schneider, Lozano-Perez, & Kaelbling, 2020) explores meta-learning of curiosity algorithms. The authors take an evolutionary approach, where an outer loop is generating candidate curiosity algorithms, and an inner loop is training a PPO agent to maximise the expected reward plus a proposed curiosity bonus. They trained over 50,000 agents (meta-learning is very computationally expensive), and discover a few algorithms that outperform all the other curiosity bonuses mentioned in this section. The most effective algorithm discovered, which they call "Fast Action-Space Transition", derives its curiosity bonus from continually training a neural network to predict the action that was just taken from a given state. It then computes the L2 distance (MSE) between this neural network ran on s_t, and it ran on s_{t+1}, to give the bonus.

3.1 Limitations

Curiosity is very much an open area of research. All methods explored in this project, and seemingly all methods that exist at the moment are quite dissatisfying. They are

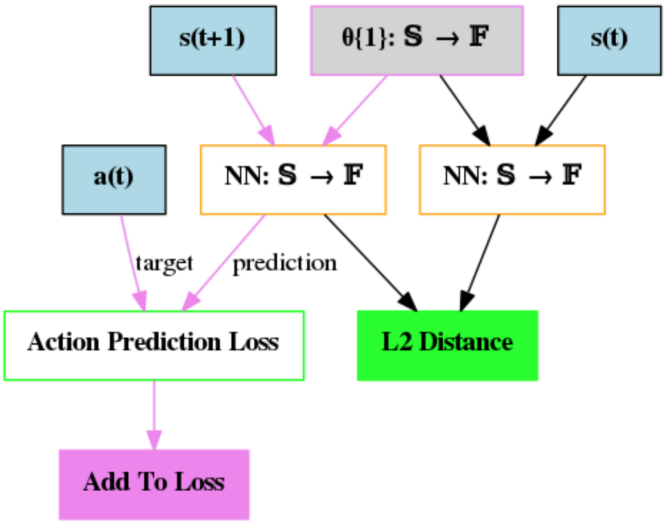


Figure 3.4: Fast Action-Space Transition Curiosity Bonus. "L2 Distance" is the bonus. (Alet et al., 2020)

incredibly sample-inefficient. In the original RND paper, they train the agent for a total of 2 billion frames of experience. It takes them about 1 billion frames to reach average human performance. At 30 FPS, this is equivalent to about one calendar year of constant play. In the Agent57 paper, they train on each game for 100 billion frames, which at 30 FPS is 106 years.

This inefficiency is because it takes many weight updates, and many samples to get a neural network to predict the next state accurately. I explore this in greater depth in 4.8.1, by training various autoencoders to predict future states of the environment, given the current state and the taken action. In these experiments, I show that it is quite challenging to train a neural network to predict the next states of even the simplest environments accurately.

Chapter 4

Experiments

In this chapter, I summarise my efforts in implementing six reinforcement learning algorithms, two of which are state of the art in curiosity-driven reinforcement learning. All code was written by myself, much of it from scratch (but drawing on literature). I trained most algorithms to play five Atari games each (the ones which I didn't are not powerful enough to play these games). Each trained model and algorithm is available on both GitHub and Google Colab in their own Python notebook; the idea being that everything needed to verify and repeat each experiment is provided in a self-contained notebook.

Source code and 30 pre-trained models are available on GitHub:
https://github.com/indrasweb/rl-and-curiosity

The following figures aggregate my results for all algorithms tested on the games Pong, Space Invaders, Pacman, Breakout, BankHeist and Montezumas Revenge. All games were of the NoFrameskip-v4 variety, meaning that the agent's actions are always executed (no sticky actions), and frame of the game is seen (NoFrameskip). Each experiment ran for at least five million emulator steps. For each of the following algorithms, I provide a link to a Colab notebook showing my results on an example environment.

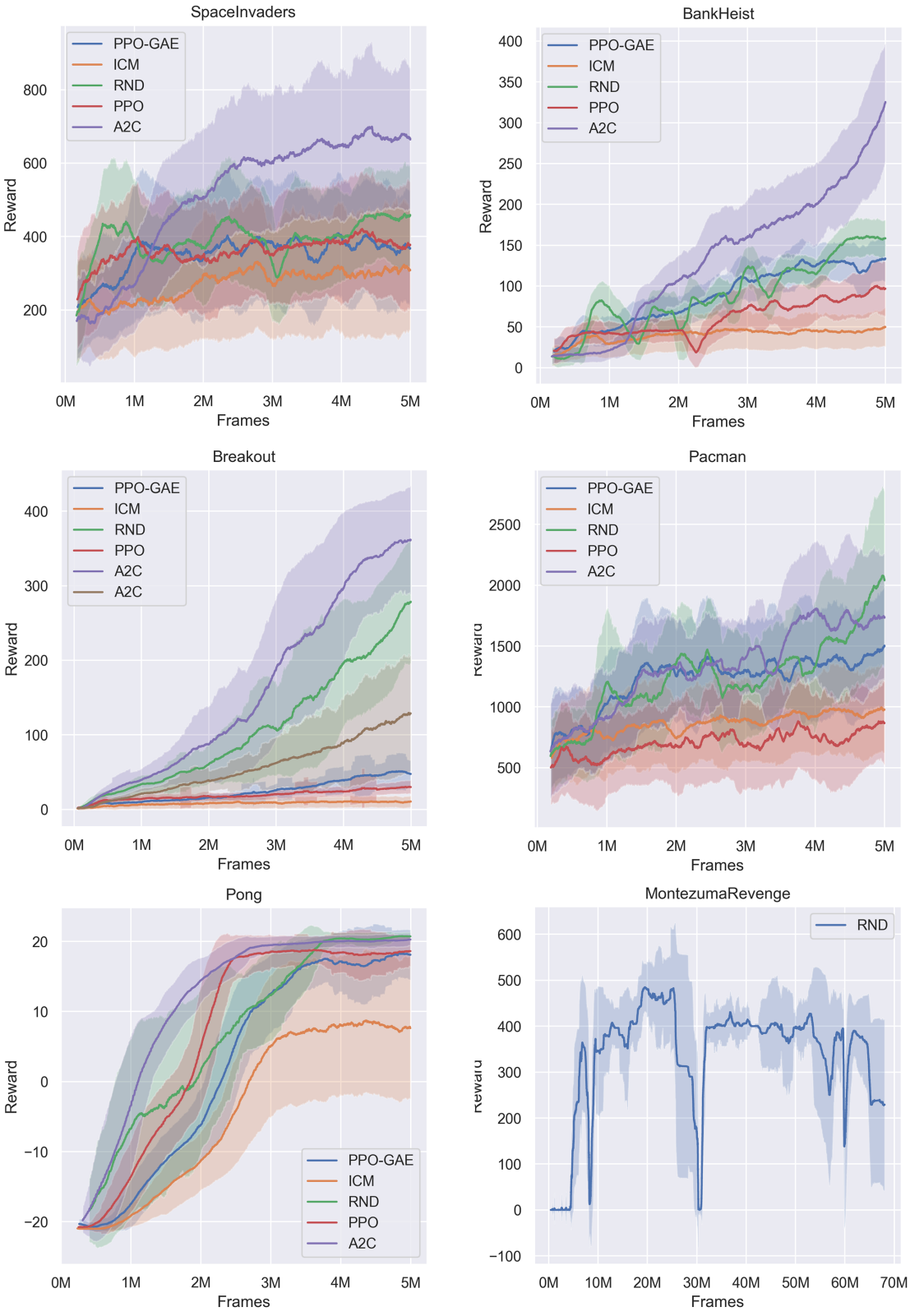


Figure 4.1: Average reward vs frames (environment steps) for all algorithms

4.1 DQN

I implemented a simple version of DeepMind's Deep Q Network and trained it on the Cartpole-v1 environment. This environment has only four features, compared with the hundreds/thousands of features seen in Atari games, making it much easier to solve. The form of DQN follows naturally from the bellman equation discussed in 2.2.3. Pseudocode for the one-step Q-learning algorithm is listed in 1. It is called one-step because we update Q-values for state S using the Q-values one step ahead in state S_{t+1}.

Algorithm 1: One-step Q-learning (Sutton & Barto, 1998)

Initialize $Q(s, a)$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ε-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q\left(s', a'\right) - Q(s, a) \right]$
    $s \leftarrow s'$
  until s is terminal

4.1.1 Results

My results, shown in figure 4.2 show that the DQN algorithm quickly solves the environment (balances the pole for more than 500 frames), however, it then becomes worse. As mentioned before, reinforcement learning algorithms are famously unstable, and DQN is no exception. One observation worth making that that Q-learning algorithms make far more weight updates than policy-gradient methods since they perform a weight update for every environment step (whereas policy gradient methods only do so every few hundred steps).

4.2 REINFORCE

The REINFORCE algorithm is the Hello World of policy gradient methods. Introduced in 1992 by (Williams, 1992), and later expanded on by Sutton (Sutton, McAllester, Singh, & Mansour, 2000), it has served as the foundation of many other algorithms. It's form follows naturally from the definition of the policy gradient discussed in 2.3.5. It allows us to update the weights of a policy in the direction of expected higher returns.

Recall the policy gradient derivation $\nabla_{\boldsymbol{\theta}} J = \mathbb{E}_{\pi} \left[\hat{A}_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \right]$.

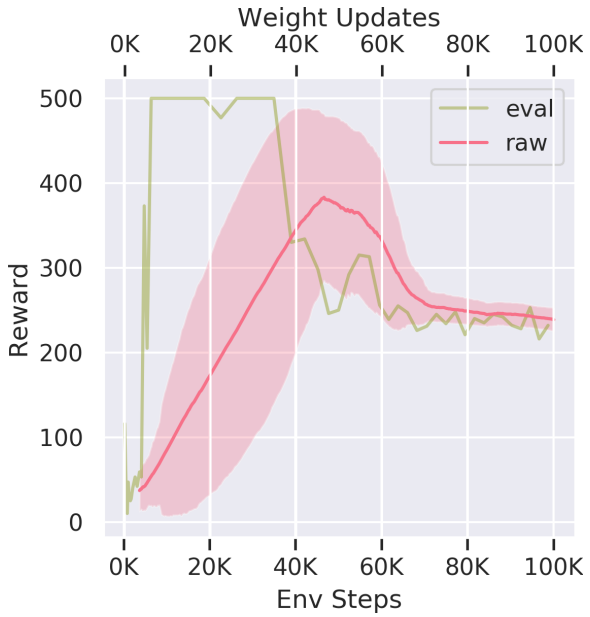


Figure 4.2: DQN CartPole-v1 Results

Open in Colab

In REINFORCE, $\hat{A}_t$ is simply the complete discounted return starting at time t, summing to the end of the episode $\sum_{t'=t}^{T} \gamma^{t'} r_{t'}$. This means that classic REINFORCE is a Monte Carlo algorithm, because we rely on complete episodes to make our updates. Below is pseudocode describing the algorithm.

Algorithm 2: Vanilla REINFORCE (Sutton et al., 2000)

Initialize differentiable policy π_θ
Repeat (for each episode):
collect rollout from environment $(S_0, A_0, R_0, ...S_T, A_T, R_T)$ following π;
$A \leftarrow$ discounted and normalized rewards;
$\boldsymbol{\theta} \longleftarrow \boldsymbol{\theta} + \alpha \frac{1}{T} \sum_t^T R_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)$;

4.2.1 Results

I wrote and ran vanilla REINFORCE on the `CartPole-v1` environment. REINFORCE takes a bit longer to solve the environment than DQN, but does not collapse in the same way DQN did (though this is just luck). The policy network used was a simple multi-layer perceptron (MLP) with two hidden layers and less than 1000 neurons. Figure 4.3 plots

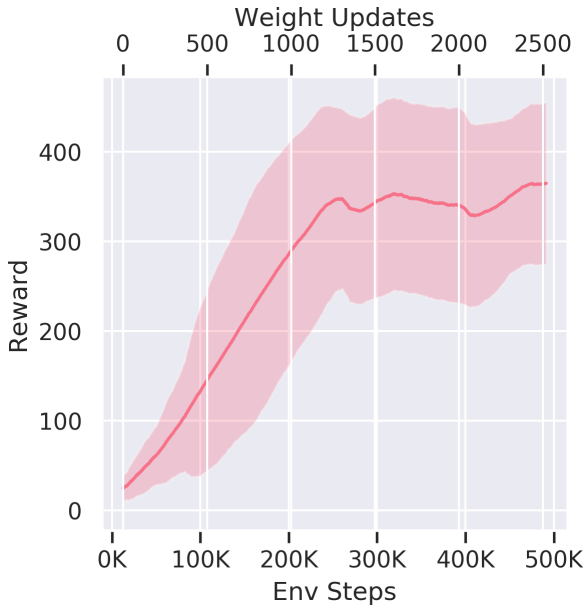


Figure 4.3: REINFORCE CartPole Results

Open in Colab

performance against environment steps. As you can see, policy gradient methods do far fewer weight updates, but still consume the same amount of environment steps (which is usually the preferred measure of algorithm performance).

In practice, it is necessary to decorrelate batches of rollouts for more robust training (expanded on in 2.5.2). This is accomplished by only computing weight updates across many rollouts. Faster training is also achieved by standardizing (subtracting mean and dividing by standard deviation) returns R before computing gradients; this is a crude form of variance reduction, detailed in 2.3.6. Vanilla REINFORCE is often not powerful enough to solve complex environments like Atari games. To solve Atari games, we need to introduce better variance control...

4.3 Maximum Entropy

One problem all reinforcement learning algorithms is the trade-off between exploration and exploitation. We want our agent to exploit whatever it can in the environment to accrue as much reward as possible. However, we would not like it to do this at the cost of perhaps discovering an even better policy for behaviour (which in the long run leads to even more reward).

We would like the agent to explore the environment to discover the global best policy, instead of settling for something that may be locally optimal. We have discussed curiosity-based methods for achieving this. However, a more common and simpler method is to introduce an "entropy penalty" over action predictions into the loss (Haarnoja, Zhou, Abbeel, & Levine, 2018).

Entropy is, informally, a measure of how unpredictable a random variable is. For example, it's harder to predict the outcome of a dice roll than a coin toss. Therefore, a dice roll has higher entropy (~ 1.79) than a coin toss (~ 0.69). Formally, entropy is defined as follows, where $P(x_i)$ is the probability of x_i occurring:

$$H(X) = -\sum P(x_i) \log P(x_i)$$

Recall, the actor outputs action probabilities. Maximum entropy reinforcement learning introduces a negative entropy penalty over these action probabilities into the loss function. This means that we are jointly optimising for increased rewards but penalising the agent if it becomes very sure of itself. This duality effectively encourages the agent to get as much reward as it can, whilst not committing too strongly, hence, giving us the flexibility to explore the environment.

All of the experiments after this section make use of the maximum entropy penalty, as do nearly all recent papers in the field. All it amounts to in practice is adding the following line to our loss: `loss -= mean([prob * ln(prob) for prob in action_probs])`.

4.4 A2C/A3C

(Mnih et al., 2016) popularised an extension to the REINFORCE algorithm, originally introduced by (Sutton & Barto, 1998). This algorithm is known as "asynchronous actor-critic" (A3C) or "synchronous actor-critic" (A2C) and is the foundation for many state of the art algorithms. Actor-critic methods are discussed in section 2.3.6.

The idea is that we have two neural networks. An actor, which outputs actions to take in the environment, and a critic, which predicts the return for any given state. This predicted return is known as the baseline $V(S_t)$, and is subtracted from the actual (discounted) return received R_t to give the *advantage* $A_t = R_t - V(S_t)$. We can interpret the advantage as the amount of unexpected reward we received. A_t is then multiplied with the policy gradient, to perform a weight update on π. We then also make a weight update to the critic, whose loss is the mean square error between predicted returns and actual returns.

A *very* important detail, and indeed one of the main contributions of the actor-critic method is that we may update our policy with only partial rollouts. We may play only part of a game, then bootstrap the expected return from the critic/value function for the next state, and use this bootstrapped return to update our actor. In practice, this means the last advantage in the trajectory becomes $A_t = V(S_t) - V(S_{t+1})$. I highlight this detail in the rollout_generator() function of my code. It is worth noting that the actor and critic are usually two head of the same neural network; they share some common "backbone" compute, then branch off from one another to predict action probabilities and value separately.

The algorithm is "synchronous" (A2C) when the process of updating the policy and critic happen in tandem; we have an update loop which is collecting a batch of rollouts from the environment using the current policy, then updating the policy and critic one after the other using these rollouts. The other variant of this algorithm, the "asynchronous" (A3C) variant, creates multiple copies of the actor and critic and has them each play in a separate environment (in a separate thread/process). Then every few updates, each separate process reports back to a global process which averages all the weight updates to update some global policy. Both variants allow for the agent to be trained with experience from parallel environments; it's just that A3C parallelises both weight updates and environment steps, whereas A2C only parallelises environment steps. Using experience from multiple environments is vital for data-correlation reasons (expanded on in 2.5.1).

Algorithm 3: A2C (Mnih et al., 2016)

Initialize differentiable policy π_θ
Initialize differentiable value function V
Repeat (for each episode):
collect rollout from environment $(S_0, A_0, R_0, ...S_T, A_T, R_T)$ following π;
$R_T = \begin{cases} 0 & \text{for terminal } S_T \\ V(S_{T+1}) & \text{for non-terminal } S_T \text{// Bootstrap from last state} \end{cases}$
discount rewards ;
$A \leftarrow R - V$;
normalize advantages A;
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \frac{1}{T} \sum_t^T R_t \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t \mid S_t)$
update critic V using mean squared error $||V - R||^2$;

4.4.1 Results

I implemented and ran A2C on five environments (Bank Heist, Breakout, Pacman, Pong, Space Invaders). Each experiment consumed 5 million frames of experience. I used eight

parallel environments, and policies were updated every five environment steps (a rollout length of 5), meaning that each policy underwent 125,000 weight updates (5,000,000/(5*8)). In each case, the policy was a convolutional neural network whose architecture is summarised in the appendix 5.1. Rewards were discounted with a gamma of 0.99, but did *not* use GAE-lambda.

This was the highest performing algorithm out of all I experimented with, which is quite surprising since it is one of the simpler algorithms. The fact that such short rollouts (5) didn't impede performance proves that the value function can accurately learn to predict the return of most states, allowing us to bootstrap from it. In fact, when I increased the rollout length to 128 for the Breakout environment, the model performed about 60% worse over the 5 million training steps (this can be seen in my results for Breakout 4.4).

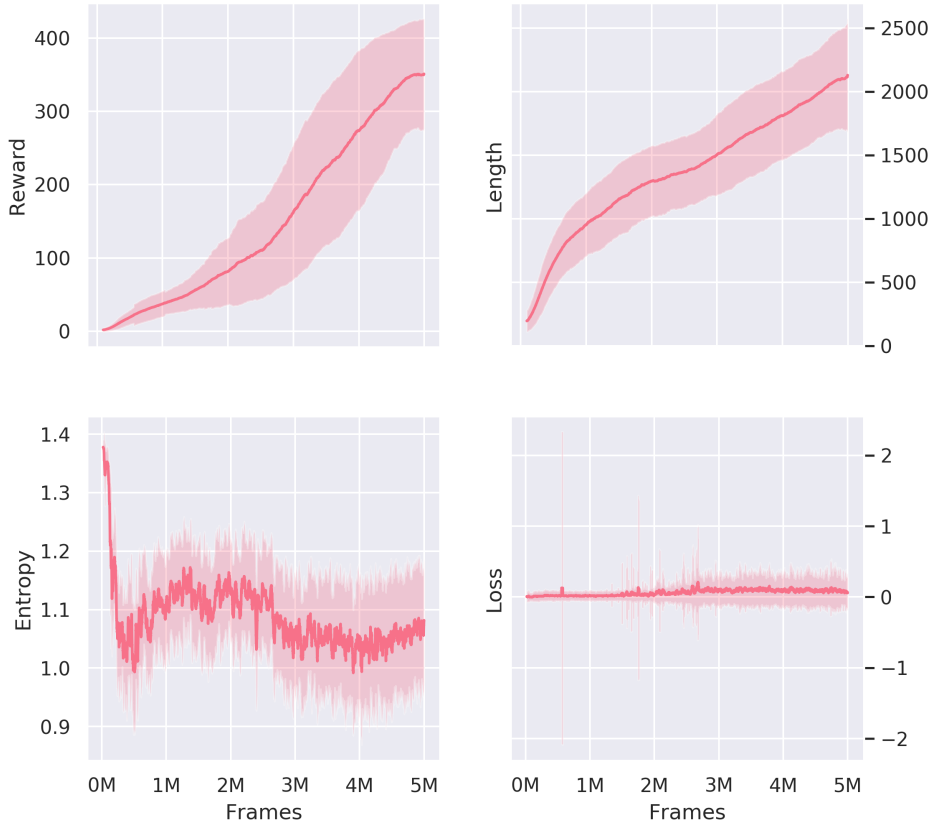


Figure 4.4: A2C Breakout Results

Open in Colab

4.5 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017) is one of the most popular RL algorithms today. It is an elegant policy gradient method based on the actor-critic dichotomy. It introduces a few improvements over A2C that can make it more robust and data efficient in certain circumstances.

The main improvement PPO relies on, "Trust Region Policy Optimization" (TRPO), was introduced by (Schulman, Levine, Moritz, Jordan, & Abbeel, 2017). This paper describes how it is beneficial to constrain all weight updates made to our policy such that no individual parameter moves too far away from its previous value. The motivation behind this is that when using regular policy gradient methods, we may accidentally sample a spurious bad batch of trajectories which, for example, are highly self-correlated. Updating with this batch of trajectories may push the policy into a region of the parameter space that means we keep sampling bad actions henceforth (remember, reinforcement learning algorithms are effectively generating their own training data by interacting with the environment. Garbage in, garbage out). I observed this happen many times when first experimenting with A2C; training would progress to a level, then suddenly performance would plummet.

Formally, what TRPO and PPO amount to is replacing the regular policy gradient update with a "surrogate objective". This replacement objective adds two additional components to the regular policy gradient loss (4.1); (1) a ratio $r_t(\theta)$ and (2) a constraint.

The ratio is computed between an old policy and a new policy, where the new policy has been updated using some samples the old policy hasn't seen yet.

$$r_t(\theta) = \frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta\text{old}}\left(a_t|s_t\right)}$$

This ratio tells us how much more (or less) likely an action is under the new policy compared to the old policy. Dropping this into the regular policy gradient objective (4.1) leaves us with 4.2.

$$L^{PG}(\theta) = \mathbb{E}_t\left[\log \pi_\theta\left(a_t|s_t\right)\hat{A}_t\right] \tag{4.1}$$

$$L^{CPI}(\theta) = \mathbb{E}_t\left[\frac{\pi_\theta\left(a_t|s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t|s_t\right)}\hat{A}_t\right] = \mathbb{E}_t\left[r_t(\theta)\hat{A}_t\right] \tag{4.2}$$

But what if an action is 100x more likely under the new policy than the old policy? This

would lead to a massive weight update which may lead to policy collapse. We therefore place a per-parameter *constraint* on this ratio by clipping it between $(1-\epsilon, 1+\epsilon)$. This prevents overly large weight updates, leaving us with the final PPO optimization objective:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right] \tag{4.3}$$

This setup allows you to perform multiple weight updates with one batch of rollouts, with 16 updates being typical. This sometimes leads to quicker and more stable convergence. Pseudocode for PPO is given in 4, where we follow the same actor-critic dichotomy given in 3.

Algorithm 4: PPO Actor Critic Style (Schulman, Wolski, et al., 2017)

Initialize differentiable policy π_θ
Initialize differentiable value function V
Repeat (for each episode):
    collect rollout from environment $(S_0, A_0, R_0, ...S_T, A_T, R_T)$ following π;
    Compute advantage estimates A from R and V (bootstrapped or using GAE-λ);
    $\pi_{\theta old} \leftarrow \pi_\theta$;
    For update in range(n):;
        Sample a random sub-batch from our batch of rollouts;
        Update π_θ w.r.t $L^{CLIP}(\theta)$, $\pi_{\theta old}$ and A;
    update critic V using mean squared error $||V - R||^2$;

4.5.1 Results

I ran experiments with two versions of the PPO algorithm. One using the simple discounted reward credit assignment scheme (PPO in 4.1). The other using generalized advantage estimate credit assignment (PPO-GAE in 4.1) with a lambda of 0.95. The generalized advantage estimate 2.4.1 outperforms regular reward discounting in most experiments, as expected.

Through my experiments, I discovered that rollout lengths must be much longer for PPO based algorithms than for A2C. I settled on 128 for all experiments. I also found that it's crucial to ensure that the initial learning rate is set at a value that results in less than 20% of the parameters being clipped per update, otherwise no learning occurs. In my code, the proportion of clipped parameters is printed as 'clip frac'. I was unable to get PPO to outperform A2C on most tasks, and I suspect this is because PPO requires longer rollouts,

meaning it makes fewer weight updates.

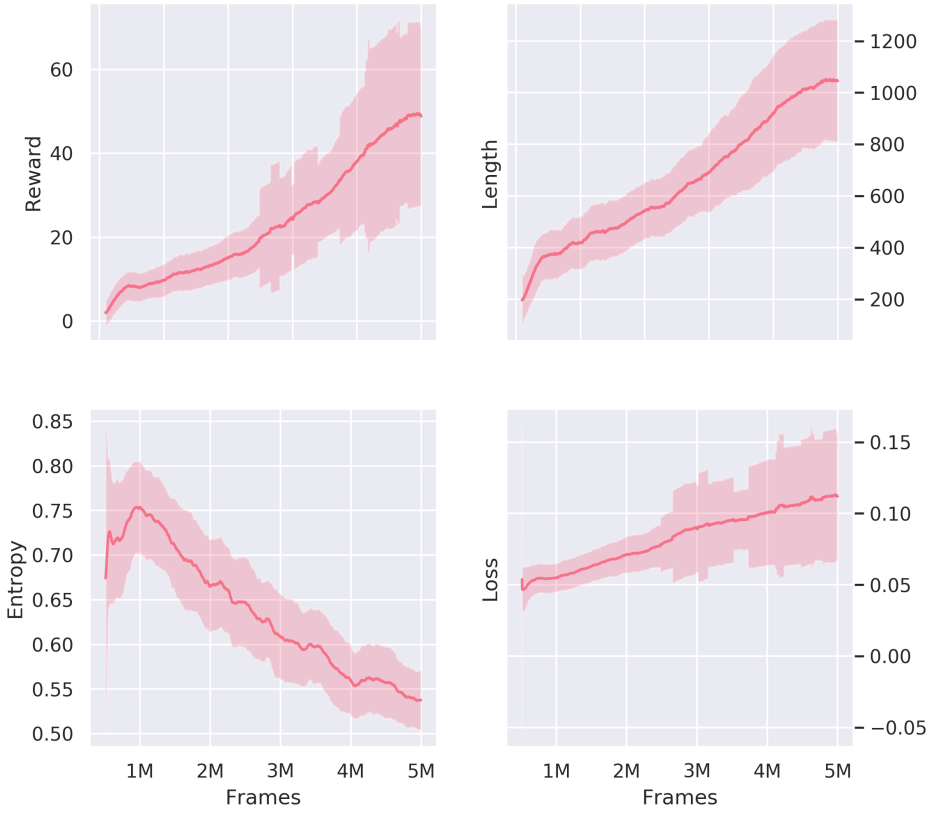


Figure 4.5: PPO-GAE Breakout Results

Open in Colab

4.6 Random Network Distillation (RND)

The curiosity bonus offered by RND is explained in 3.0.4. With reference to 3.2, this algorithm relies on five neural networks.

1. An actor which outputs actions to take in environment.
2. The "fixed" RND network, which is initialised randomly, but never trained.
3. The "predictor" RND network, which is trying to learn to imitate the fixed network. Our intrinsic bonus for each step is the mean squared error $||\text{fixed}(S_{t+1}) - \text{predictor}(S_t)||^2$

4. A critic, which predicts the extrinsic reward (points from the game).

5. Another critic, which predicts the intrinsic reward.

RND trains an agent using a similar procedure to PPO 4. The only differences are that it maximises both the intrinsic bonus computed between the fixed and predictor networks as well as the extrinsic reward from the environment. It also trains an intrinsic reward critic as well as an extrinsic reward critic, allowing us to baseline both sources of reward separately.

There are a few important implementation details that make or break RND. One is that we have to normalise all inputs to the fixed network using a running mean and standard deviation of prior inputs. Because the network parameters are fixed, they are unable to adapt to the scale of the inputs, and hence the output embedding can have very low variance, and carry little to no information about the input distribution. Normalisation ensures that information from the input is still amplified throughout the network. Normalisation is realised as a wrapper around the environment (expanded on in 2.5.1), which must be initialised for a few thousand frames before training commences.

Another detail that can help speed up training a lot is to ensure that the actor-critic networks are initialised correctly, using something like Kaiming initialisation; a dodgy initalisation can prevent the fixed network from encoding anything useful.

4.6.1 Results

Along with the five games I trained all other algorithms on, RND is the only algorithm that I trained for a substantial amount on frames (70 million) on Montezuma's Revenge, with GAE being the credit assignment regime. The progress made in these 70 million frames proves that my implementation works, and my results are plotted in 4.6/ For comparison (not shown on the graph) I trained A2C on Montezumas Revenge for about 15 million frames, but the algorithm made no progress. I would have like to have run it for 2 billion frames as they do in the original paper but lacked the compute resources to do so. I feel if I had ran it for longer it may have been possible to see clearer spikes in intrinsic reward when the agent discovers a new part of the map.

When I initially implemented this algorithm, I forgot to discount the intrinsic rewards. This had extremely detrimental effects on performance. Another error I made was having a too higher learning rate. This had the effect of making it such that a high proportion of gradients were clipped, thus stifling learning. In the end, I settled for a learning rate of 1e-4 (using the Adam optimizer) which improved training.

As you can see from my results 4.1, RND outperforms all other algorithms aside from A2C; however, as we've established this comparison is not apples to apples because A2C trained networks receive many more updates (due to the sorter rollout length). Whilst this paper/algorithm has been highly acclaimed, it is not a magic wand. Any implementation must have carefully balanced hyperparameters and initialization must be somewhat lucky; but I suppose this is the case with most RL, it is famously brittle.

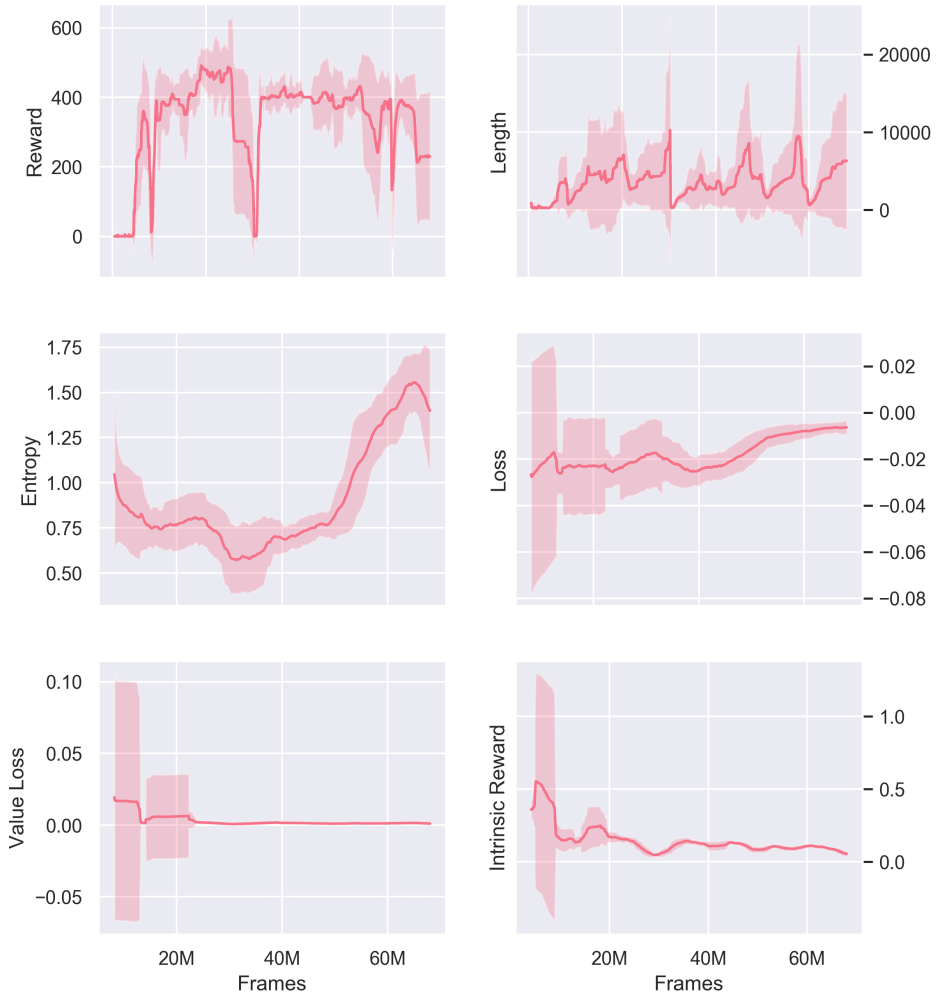


Figure 4.6: RND Montezuma's Revenge Results

Open in Colab

4.7 Intrinsic Curiosity Module (ICM)

The curiosity bonus offered by ICM (Pathak et al., 2017) is explained in 3.1. ICM uses A2C (3) to maximise a combination of both intrinsic and extrinsic reward. I chose 10% intrinsic and 90% extrinsic for the experiments I ran. Credit is distributed using GAE with a lambda of 0.95.

With reference to figure 3.1, the ICM intrinsic bonus is computed using the procedure outlined in 5. This bonus is plugged into an A2C which attempts to maximise it, as well as the extrinsic reward.

Algorithm 5: ICM Bonus

input : Policy π_θ, state encoder ϕ, state prediction model F, action prediction model I

Take an environment step under $\pi_{\boldsymbol{\theta}}$ for (S_t, A_t, R_t, S_{t+1}).;
Encode S_t and S_{t+1} into latent representations $\phi(S_t), \phi(S_{t+1}) \in \mathcal{R}^{128}$;
Compute $f_t = F(\phi(S_t), A_t)$, which predicts $\phi(S_{t+1})$;
Compute $i_t = I(\phi(S_t), \phi(S_{t+1}))$ which predicts A_t;
Intrinsic bonus $r_t^i = ||\phi(S_{t+1}) - f_t||^2$;
Backprop through f_t and i_t;

4.7.1 Results

Unfortunately I could not get ICM to work very well. I suspect this is because I either initialise the forward model and inverse model incorrectly, or there is a bug that is causing backprop to proceed through parts of the graph that it shouldn't. I was reluctant to submit my report with this algorithm, however my implementation of it is very clean, and I'm sure when I revisit it soon I will find what's wrong with it.

You can see from my results that it does make a little bit of progress in most games, however it is extremely slow. The fact that the intrinsic bonus rapidly decreases is expected (since ICM should quickly learn the pixel distribution of the game).

When I revisit this algorithm, I will try and get it to play the game of Mario using just intrinsic reward (no extrinsic reward). They show that this is possible in the original paper (Pathak et al., 2017).

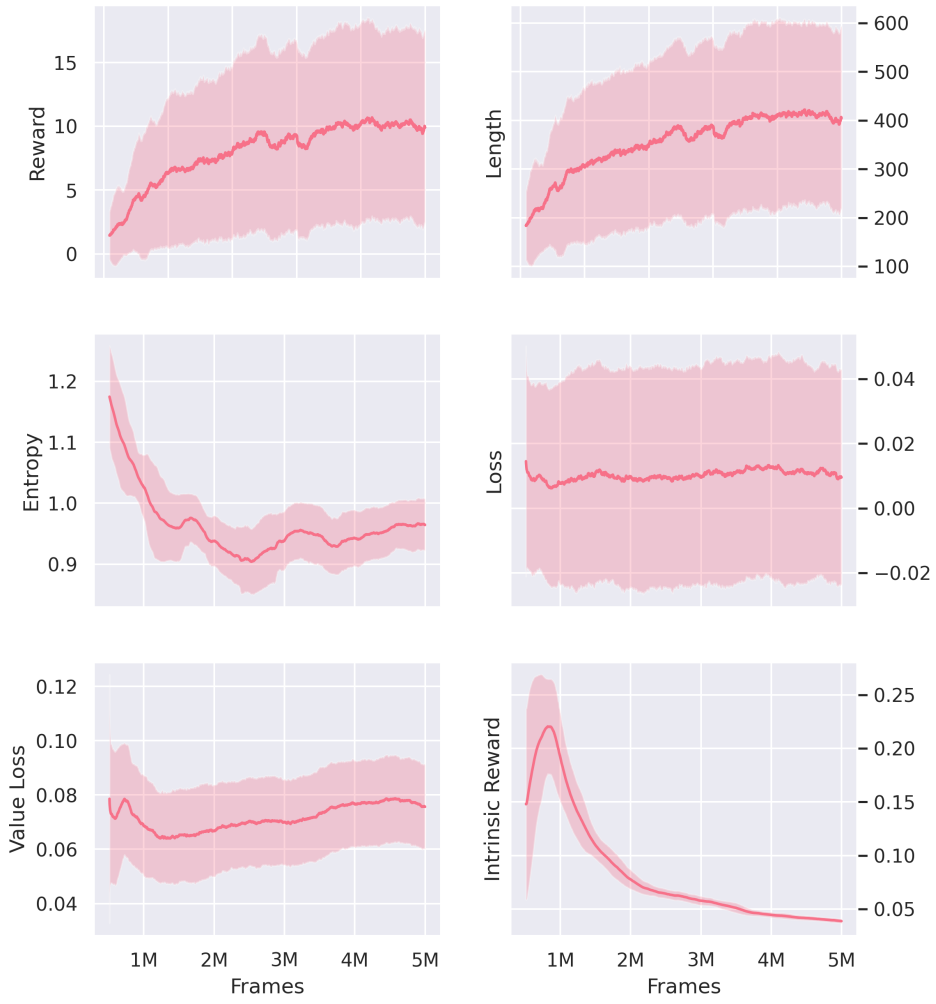


Figure 4.7: ICM Breakout Results

Open in Colab

4.8 Next-state prediction

Towards the end of my project, I became interested in the idea that we can do reinforcement learning on a learned representation of the environment. That is, we can use an auto-encoder to learn a compressed representation of the world, then use reinforcement learning to learn a behavioural policy inside this latent space.

This has been explored in a few papers, most notably (Ha & Schmidhuber, 2018; Hafner et al., 2019), which train a variational auto-encoder (Kingma & Welling, 2014) to predict

S_{t+1} given S_t, after passing it through some bottleneck of 64 latent neurons. After training this auto-encoder, they use an evolutionary algorithm to learn a policy that maximises the expected survival time inside the environment by just looking at these 64 neurons. They are able to use an evolutionary algorithm, instead of deep learning, because the input dimensionality is so small.

I experimented with various architectures for next-state prediction, to see if I could learn a model that accurately encodes future states of the environment, given the current state and the action to be taken.

4.8.1 Auto-Encoder

As a sanity check, I trained a regular auto-encoder with 256 latent neurons to reconstruct one image. 4.8 shows training progress over about 800 weight updates.

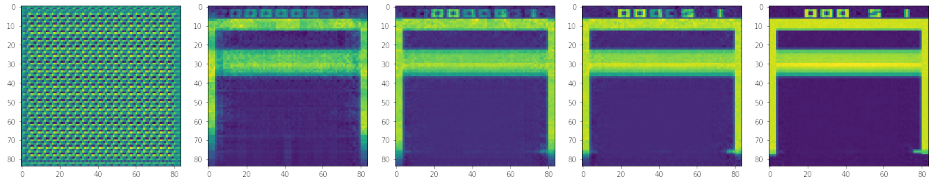


Figure 4.8: Training Progress of Simple Auto-Encoder on Frame of Breakout

I wanted to find a model which was able to accurately predict the next state given the current state and action to be taken. I hoped I could find a model that was at least able to accurately identify the pixels that are directly changed by the action (e.g. the paddle in pong).

I trained a regular auto-encoder on random episodes of gameplay, where the input to the auto-encoder was the current state (pixels) plus a one-hot encoding of the action to be taken, but, despite ample training, this didn't work so well. This may have been to a sub-optimal selection of hyperparameters.

4.8.2 U-Net

I tried again using the U-Net architecture (Ronneberger, Fischer, & Brox, 2015). This has the same hourglass shape most auto-encoders have, but also contains skip-connections which allow information from earlier layers in the encoder to 'skip over' into their mirrored layers in the decoder. I hoped this would force the latent space to emphasise information

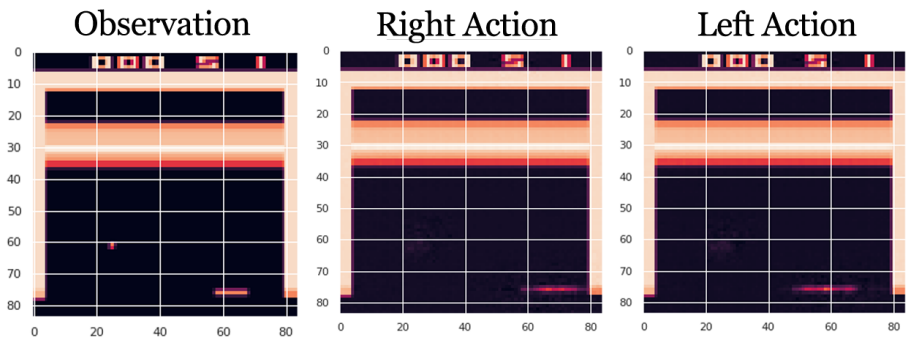


Figure 4.9: Left and right action predictions of U-Net trained on Breakout

relevant to the difference between the input frame and the next frame, as it wouldn't have to learn to reconstruct every single pixel.

I tried training three different U-Net setups. One where the action was given as a one-hot 3D tensor concatenated to the image input, another where the one-hot action was concatenated only to the latent space, and one more where the action was concatenated to all layers of the network. Though I thought these choices would have a significant effect, they didn't; the version where the action was concatenated to all layers performed marginally better than the rest.

After training on just on 100,000 frames of experience (1 hour at 30 FPS) the model was able to just about figure out that the left action means the paddle will move left, and the right action means the paddle will move right. This can be seen in 4.9, on the left is the input state, the middle and right are the U-Net predictions of the next state given the left and right actions, respectively.

I next instantiated two of the same network, with the same initial weights, and trained one to predict S_{t+1} and the other to predict S_{t+2}. 4.10 plots the respective losses (mean squared error) for each over the course of training. The high correlation between the two is interesting, and shows something that is intuitively true; it is harder to predict two steps in the future than one step.

I ran out of time during my project, but in the future, I will see if I can train an RL agent to play something like Breakout by only looking the latent representation of the U-Net developed in this section.

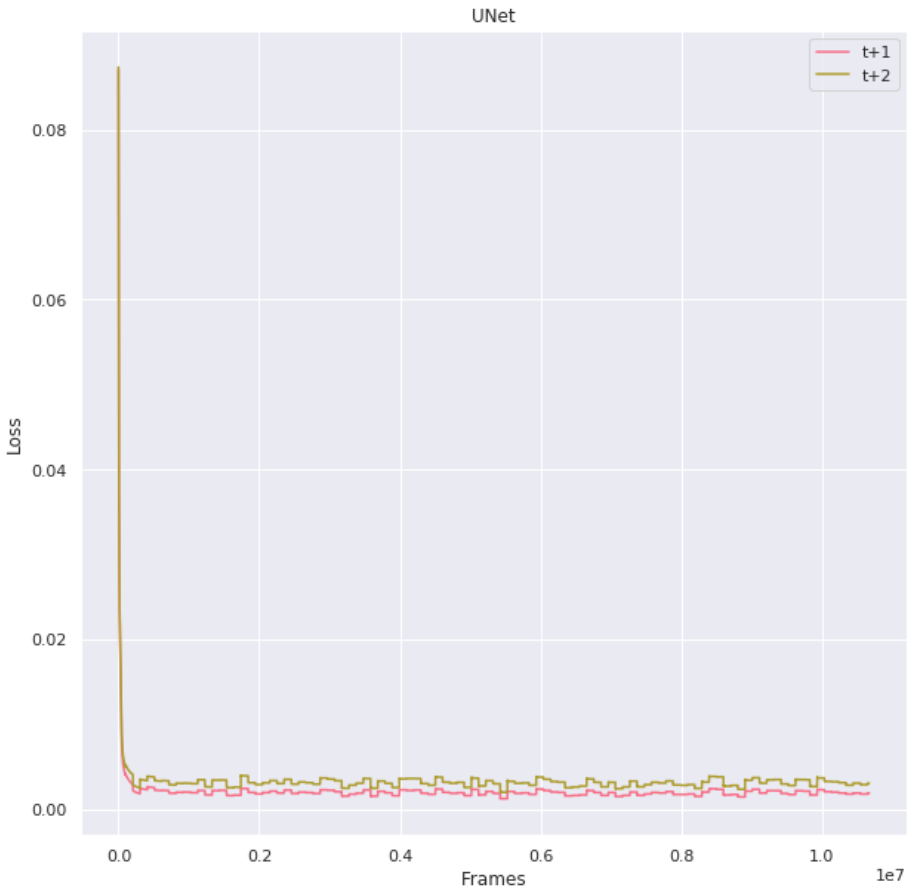


Figure 4.10: U-Net MSE for future predictions (S_{t+1}, S_{t+2})

Chapter 5

Conclusion

5.1 Reflections

The aim of this project was to familiarise myself with the landscape of reinforcement learning algorithms and to bring me up to speed with research in curiosity-driven RL. I have accomplished most of what I set out to, implementing (mostly from scratch) and comparing the following algorithms: REINFORCE, DQN, A2C/A3C, PPO, PPO-GAE, ICM, RND. All algorithms are implemented as self-contained experiments in Python notebooks, and trained models accompany each instance.

It took me a few weeks longer than I anticipated to get to grips with the simple REINFORCE and actor-critic algorithms because I managed to write a few bugs that were very difficult to hunt down. I learned that when I get stuck implementing a paper, I should find a reputable implementation written by someone else, and step through it in a debugger to learn how it works. I feel this is the best technique for understanding RL algorithms because there are *a lot* of implementation details that make or break certain algorithms, which are easy to miss, or omitted, from papers.

I encountered some self-inflicted technical debt; the code that I wrote initially while playing around with the algorithms was very poor. I went back and re-wrote a lot to make it something I was proud of and happy to share on GitHub.

If I had done a more thorough literature review, I would have encountered the "Procgen benchmark", which I would have used in place of Atari games. This is much better for assessing algorithm generalization because all games are procedurally generated, and therefore unique, meaning that an agent really has to learn the underlying rules, and not rely on memorizing pixels.

I would love to have implemented the world-models paper (Ha & Schmidhuber, 2018) fully and experimented with different techniques for environment compression; however, I only managed to explore simple auto-encoders and U-Nets for this purpose and didn't try any reinforcement learning inside the learned latent spaces. I decided it was better to focus on curiosity-driven learning and save this for another project.

It would have also have been great to implement Agent57 (Badia et al., 2020) and the curiosity bonus "FAST" discovered in (Alet et al., 2020), however, they were published in March 2020, and Agent57 seems very complicated to implement.

5.2 Impact of COVID-19

Fortunately, Covid-19 had little to no impact on my project.

5.3 Future Directions

Now I have a firm grasp on the modern foundations of deep reinforcement learning and artificial curiosity, I am in a position to contribute to the field. I plan to investigate a novel connection between "neuromodulation" (Beaulieu et al., 2020) and curiosity-driven reinforcement learning. The high-level idea is to design a system which *directs* its curiosity toward something specific in the environment. I hope to accomplish this by using a modulatory process that selectively gates neurons in the curiosity/intrinsic bonus network. This directed curiosity should hopefully allow for more efficient exploration and more goal-oriented behaviour.

5.4 Summary

This thesis has explored the fundamentals, as well as some more sophisticated aspects of reinforcement learning. I have reviewed implemented and compared six different algorithms on the Atari 2006 benchmark. I also explored curiosity-driven learning, which seeks to develop algorithms that reward agents when they encounter novel, but *learnable* regularities in their environments. Methods developed thus far are dissatisfying because they are highly sample inefficient, but this is exciting because it means there is ample room for research and improvement!

Bibliography

Alet, F., Schneider, M. F., Lozano-Perez, T., & Kaelbling, L. P. (2020, March). Meta-learning curiosity algorithms. *arXiv:2003.05325 [cs, stat]*.

Badia, A. P., Piot, B., Kapturowski, S., Sprechmann, P., Vitvitskyi, A., Guo, D., & Blundell, C. (2020, March). Agent57: Outperforming the Atari Human Benchmark. *arXiv:2003.13350 [cs, stat]*.

Beaulieu, S., Frati, L., Miconi, T., Lehman, J., Stanley, K. O., Clune, J., & Cheney, N. (2020, March). Learning to Continually Learn. *arXiv:2002.09571 [cs, stat]*.

Bertsekas, D. P. (n.d.). Dynamic Programming and Optimal Control 3rd Edition, Volume II. , 233.

Brodski, A., Paasch, G.-F., Helbling, S., & Wibral, M. (2015, June). The Faces of Predictive Coding. *Journal of Neuroscience*, *35*(24), 8997–9006. doi: 10.1523/JNEUROSCI.1529-14.2015

Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018, October). Exploration by Random Network Distillation. *arXiv:1810.12894 [cs, stat]*.

Cobbe, K., Hesse, C., Hilton, J., & Schulman, J. (2019, December). Leveraging Procedural Generation to Benchmark Reinforcement Learning. *arXiv:1912.01588 [cs, stat]*.

Coon, D., & Mitterer, J. O. (2007). *Introduction to psychology : Gateways to mind and behavior* (11th ed ed.). Belmont, CA ; Australia : Thomson/Wadsworth.

Degris, T., White, M., & Sutton, R. S. (2013, June). Off-Policy Actor-Critic. *arXiv:1205.4839 [cs]*.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2019, May). Go-Explore: A New Approach for Hard-Exploration Problems. *arXiv:1901.10995 [cs, stat]*.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., & Clune, J. (2020, April). First return then explore. *arXiv:2004.12919 [cs]*.

Ha, D., & Schmidhuber, J. (2018, March). World Models. *arXiv:1803.10122 [cs, stat]*. doi: 10.5281/zenodo.1207631

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, August). Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor.

arXiv:1801.01290 [cs, stat].

Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., & Davidson, J. (2019, June). Learning Latent Dynamics for Planning from Pixels. *arXiv:1811.04551 [cs, stat]*.

Hausknecht, M., & Stone, P. (2017, January). Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv:1507.06527 [cs]*.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., ... Wu, Y. (2018). Stable baselines. *GitHub repository*.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. doi: 10.1162/neco.1997.9.8.1735

Kingma, D. P., & Welling, M. (2014, May). Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]*.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., ... Kavukcuoglu, K. (2016, June). Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Hassabis, D. (2015, February). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. doi: 10.1038/nature14236

Oudeyer, P.-Y., & Smith, L. B. (2016). How Evolution May Work Through Curiosity-Driven Developmental Process. *Topics in Cognitive Science*, *8*(2), 492–502. (_eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/tops.12196) doi: 10.1111/tops.12196

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017, May). Curiosity-driven Exploration by Self-supervised Prediction. *arXiv:1705.05363 [cs, stat]*.

Policy Gradient Algorithms. (2018, April). https://lilianweng.github.io/2018/04/08/policy-gradient-algorithms.html.

Ronneberger, O., Fischer, P., & Brox, T. (2015, May). U-Net: Convolutional Networks for Biomedical Image Segmentation. *arXiv:1505.04597 [cs]*.

Ryan, R. M., & Deci, E. L. (2000, January). Intrinsic and Extrinsic Motivations: Classic Definitions and New Directions. *Contemporary Educational Psychology*, *25*(1), 54–67. doi: 10.1006/ceps.1999.1020

Schmidhuber, J. (2009). Simple algorithmic theory of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes (–)..

Schmidhuber, J. U. (1991). Adaptive Confidence and Adaptive Curiosity..

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017, April). Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*.

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018, October).

High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv:1506.02438 [cs]*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017, July). Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*.

Sovrano, F. (2019, August). Combining Experience Replay with Exploration by Random Network Distillation. *2019 IEEE Conference on Games (CoG)*, 1–8. doi: 10.1109/CIG.2019.8848046

Strehl, A. L., & Littman, M. L. (2008, December). An analysis of model-based Interval Estimation for Markov Decision Processes. *Journal of Computer and System Sciences*, *74*(8), 1309–1331. doi: 10.1016/j.jcss.2007.08.009

Sutton, R. S., & Barto, A. G. (1998). *Introduction to Reinforcement Learning* (1st ed.). Cambridge, MA, USA: MIT Press.

Sutton, R. S., McAllester, D. A., Singh, S. P., & Mansour, Y. (2000). Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. A. Solla, T. K. Leen, & K. Müller (Eds.), *Advances in Neural Information Processing Systems 12* (pp. 1057–1063). MIT Press.

Tsividis, P. A., Pouncy, T., Xu, J. L., Tenenbaum, J. B., & Gershman, S. J. (2017). Human learning in Atari. In *2017 AAAI Spring Symposium Series.*

Wikipedia. (2020). *Monte Carlo method — Wikipedia, The Free Encyclopedia.*

Williams, R. J. (1992, May). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*(3), 229–256. doi: 10.1007/BF00992696

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 20, 20]           8,224
            Conv2d-2          [-1, 32, 20, 20]           8,224
              ReLU-3          [-1, 32, 20, 20]               0
              ReLU-4          [-1, 32, 20, 20]               0
            Conv2d-5            [-1, 64, 9, 9]          32,832
            Conv2d-6            [-1, 64, 9, 9]          32,832
              ReLU-7            [-1, 64, 9, 9]               0
              ReLU-8            [-1, 64, 9, 9]               0
            Conv2d-9            [-1, 64, 7, 7]          36,928
           Conv2d-10            [-1, 64, 7, 7]          36,928
             ReLU-11            [-1, 64, 7, 7]               0
             ReLU-12            [-1, 64, 7, 7]               0
          Flatten-13                [-1, 3136]               0
          Flatten-14                [-1, 3136]               0
           Linear-15                 [-1, 512]       1,606,144
           Linear-16                 [-1, 512]       1,606,144
             ReLU-17                 [-1, 512]               0
             ReLU-18                 [-1, 512]               0
           Linear-19                  [-1, 18]           9,234
           Linear-20                   [-1, 1]             513
================================================================
Total params: 3,378,003
Trainable params: 3,378,003
Non-trainable params: 0
----------------------------------------------------------------
Input shape: (-1, 4, 84, 84)
Input size (MB): 0.11
Forward/backward pass size (MB): 0.71
Params size (MB): 12.89
Estimated Total Size (MB): 13.70
----------------------------------------------------------------
```

Figure 5.1: Actor Critic (A2C) CNN Architecture