Benchmarking Deep Reinforcement Learning in Scalable Environments

THE UNIVERSITY OF MELBOURNE

Submitted in partial fulfilment for the degree of
Master of Science (Computer Science)
COMP90070 - 75 Points

Kinsey Reeves
695705

Supervisor: Dr Miquel Ramirez Javega

June 2020

Abstract

Reinforcement learning is a framework where an agent learns to maximise reward in a given environment. Throughout our lives, humans interact with environments which result in them receiving both intrinsic and extrinsic rewards. A child will explore random actions, words or movements to learn cause and effect, quickly developing policies of language, movement or interaction to maximise their intrinsic and extrinsic reward. Reinforcement learning systems initially act randomly within an environment to learn cause and effect, or, a state-action mapping. It is a way of programming agents to learn an objective through reward and punishment only. Reinforcement Learning, a field in machine learning differs from classic Artificial Intelligence in that the agent learns purely through trial and error, with an extrinsic reward provided by the environment. Intelligence through iterative interaction with an environment is central to all animal intelligence.

Deep Reinforcement Learning, a relatively new field of research, focuses on approximating large state-action mappings with a neural network. The empirical results have been impressive, beating human performance at complex games such as Go [16], Atari or steering complex robots [12]. The types of environments are diverse, and their difficulty is not always apparent. Two Atari games, with seemingly similar state spaces and action spaces, could have very different challenges for a Deep Reinforcement Learning actor.

The objective of this Thesis is to measure the performance of standard Deep Reinforcement Learning algorithms against increasingly difficult environments with known optimal policies. These policies are based on known algorithms in robotics and AI planning. The designed benchmarks will measure the scaling effectiveness of each algorithm, precisely measuring how close the agent gets to the optimal policy while the complexity of the environment increases. This Thesis attempts to answer the question: how close can Deep reinforcement learning approximate known planned solutions or policies in Markov decision processes? The proposed environments, benchmarks and analysis also highlight the importance of MDP design.

Contents

Chapter 1

Introduction

1.1 Motivation of Research

The goal of this research is to answer a set of questions about reinforcement learning, its uses and the difficulty of problems it solves. Deep Reinforcement Learning has been applied to a wide range of environments. The inherent difficulty of a problem is not always clear to a human. For instance, solving simple mazes in approximately the least steps quite a simple task to a human given a clear goal and enough time, however this might not be the case for a RL agent. This thesis proposes a suite of benchmarking environments used to test the effectiveness of continuous and discrete action space reinforcement learning algorithms while scaling the environments complexity. With this framework, a developer will be able to asses exactly how far from the optimal policy their model is.

In typical reinforcement learning, the goal is to find a policy, that is a set of actions given the state which maximises the discounted future reward. Reinforcement learning algorithms have been proven to converge to optimal policies, such as Q-Learning [31] or TD-Learning. Reinforcement learning has excelled at solving time dependent control problems or specifically Markov Decision Processes, MDP's for short [18]. Most reinforcement learning algorithms are centered around improving some value function - how good it is to be in a state, or improving a policy function - which actions are best in a state. Such problems have their origin in dynamic programming [1] (DP). In many cases planning allows us to similarly solve these MDP's, at the cost of computational resources and programmatic complexity to the developer. Often, a heuristic search is too computationally expensive. Reinforcement learning allows us to create a general controller, that approximates an optimal policy. In domains where resources are limited, general reinforcement learning is a scalable approach at the cost of some policy optimality. This paper aims to measure the loss of optimality using Reinforcement learning on stochastic, scalable environments.

1.2 Problem Definition

The hype around reinforcement learning and its success in games like Go [**alpha˙go**], Starcraft [30] and various simulated environments has made it a popular approach to solving MDP's. The aim of this thesis is to measure, against a set of planning benchmarks, how close Reinforcement Learning agents come to approximating an optimal policy, found through planning. In many problems, the optimal policy is unknown, and cannot be computed due to the possible state size of the environment. Reinforcement Learning allows us to obtain an approximation of the optimal policy. How close to optimal a reinforcement learning algorithm reaches is not usually known. With knowledge of how close to a planned benchmark a RL approach is, we can be more confident in the implementation of an agent in large or complex environments that do not permit classical search.

Secondly, measuring the limitations of RL by scaling environments complexity allows us to decide with a structured approach when to apply reinforcement learning to a problem, with what reward and state mapping. It also allows us to measure the inherent difficulty of the problem. The aim of this thesis is to develop and test a set of benchmarks which will measure the convergence of reinforcement learning agents towards known heuristic search policies.

1.3 Approach

In recent years, Deep Reinforcement learning has exploded in popularity with many new variations of a select core group of algorithms. Although there may be more state of the art algorithms, this paper will deal with the no-frills versions of the most common, and ground-breaking deep RL algorithms. This is due to them being most widely cited in various deep RL literature, and the availability of them in most deep RL libraries. Hyperparameters will be explored, however, the range of environement settings plus algorithmic hyperparameters is large. In an ideal world, all combinations of parameters, environment settings and environment scales would be tested. However, much of the testing was completed on a single CPU. Therefore, this thesis will explore the more interesting settings, parameters and results.

1.4 Significance and Contribution

This report offers an end-to-end framework for training and evaluating modern Deep Reinforcement Learning algorithms. Each environment allows a developer to scale various aspects to stress test their Deep RL model. Interestingly, the performance of an algorithm often does not scale with the complexity of an environment.

While providing a solution to test RL algorithms ability to approximate the optimal policy, the analysis of the proposed framework in this report highlights a number of considerations the developer should take in implementing a RL environment. Applying RL to a problem is as much about portraying the problem in an effective MDP setting as it is about parameter tuning or selection of specific algorithm. In a supervised learning setting, emphasis is often put on the feature engineering and data curation before applying the model. This report demonstrates how that is equally true in a RL setting - that an environments reward, state observations, and complexity are equally important in creating a well performing RL model.

1.5 Report Structure and Scope

This thesis will present a background and literature review on the main modern developments of Deep Reinforcement Learning, the types of problems it typically solves and an overview of the common Deep Reinforcement Learning algorithms. Secondly, proposed environments will be presented in which the algorithms will be evaluated alongside their respective benchmarks. The benchmarks will serve as the optimal policy of the deep RL algorithms. In cases such as pathfinding, the benchmark will be a proven optimal policy. In robotic control it will be a close approximation. Each environment will include its hyper-parameters, justifications for these and how each algorithm scales in complexity relative to the benchmark.

Next, the set of deep RL algorithms will be tested in the proposed environments with scaling difficulty. Each algorithm will be tested with a range of searched hyper-parameters to find the best solution to the scaling where possible. A number of modern approaches will be used to improve learning which will be discussed. A full set of results will be included in the appendix for full hyper-parameter and environment setting exploration to justify most hyper-parameters chosen.

Many modern Deep RL algorithms require large amounts of training time and are run on modern super computers. This report runs each algorithm for a fixed period of time, generally 30 minutes per parameter combination due to resource constraints. A complete grid search of all parameter options for all algorithms and environments would be interesting however did not fall within the scope of experimentation. A trade-off had to be made, to explore all possible parameter combinations for all algorithms, yet not look closely at any in particular, or choose certain algorithms and conditions to focus more specifically on. The latter was chosen to give a more detailed report on what are seemingly the interesting outcomes of the experimentation.

Chapter 2

Background

The background will serve as a literature review of modern RL, including the relevant terminology, algorithms and frameworks which are common in literature. The background follows various relevant chapters from [26] which is a common foundation of syntax and terminology throughout RL literature. First, the critical ideas in reinforcement learning are outlined, which are essential in understanding the different types and features of RL algorithms. This includes the general idea of reinforcement learning and in what domains it works. Next, the common types of algorithms are outlined. Finally, the technical details of each algorithm are described, outlining the main features and contribution of each.

2.1 Markov Decision Processes

Reinforcement Learning (RL) agents act within environments known as Markov Decision Processes (MDP). MDP's are a mathematical model used to shape an agent's behaviour through maximising the reward. The Markov Property states that the current state of an environment is independent of the past given the present. Simply put, an agent has sufficient information of the past to act, given the present state. A conclusion from this is that an agent always has enough information with only the present state observation to act optimally. Two main characteristics set Markov Decision Processes apart from other Markov Models, one, an agent has control over the environment with actions and two, states are completely observable. An MDP is an environment in which all states are Markov. An MDP is formulated by the tuple :

$$(S, A, P, R, \gamma) \tag{2.1}$$

Where S are the states, A, the action space, R the rewards, P the probabilities of action transitions, and γ the discount factor between states. The transition probabilities are in our case always 1, as each action transition is deterministic in both the Grid environment and reacher environment, discussed later. γ is generally defined within the RL agent, as we want to find a weighting which discounts future rewards and can be tuned depending on the task. The discount factor allows an agent to learn policies which delay immediate rewards for future rewards, a foundational idea in both reinforcement learning and psychology.

Reinforcement learning agents interact with their MDP environment at each state $s_t \in S$ in discrete time steps t choosing an action $a \in A$, influencing their transition to the next state s and receiving possible reward r_t at each state. In an MDP, the goal is to find a policy $\pi(a|s)$ for choosing the best actions in states, or ideally the optimal policy $\pi^*(a|s)$, which is a solution to the MDP. A solution to an MDP is the set of actions which maximise the cumulative reward throughout an episode. An episode is a set of states over which an agent acts. Consider a single game of tic-tac-toe or one life in Pacman. A policy is a mapping for each state to a specific action or action probability.

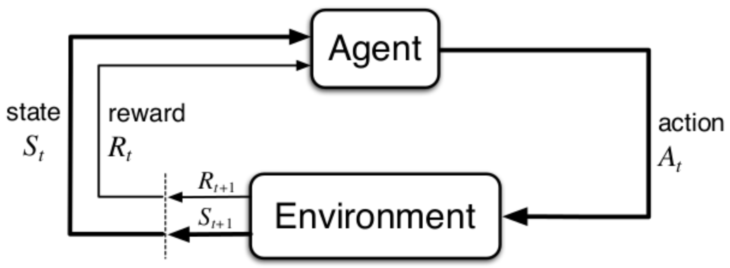

Figure 2.1: Diagram of an MDP

2.2 Reinforcement Learning

Reinforcement learning is a field of machine learning concerned with an agent learning what to do, or how to map observations to actions in a set of sequential states. The goal of the agent is to converge towards an optimal policy. It is known that a deterministic, optimal policy exists for all MDP's [4].

Reinforcement learning is used in problems in many fields where the goal is optimising behaviour over time, such as robotic control, scheduling, game-theoretic environments and so on. Reinforcement learning works regardless of whether we have a model of the environment and can learn through interaction only. It is this that sets reinforcement learning apart from typical supervised learning - it generates its own training data. A reinforcement learning agent observes its state, performs a chosen action according to the policy and receives a reward. These rewards are often sparse, and time-delayed, meaning agents must choose actions which maximise expected discounted reward. An example might be reaching the end of a complex maze, or moving a robotic arm over multiple steps to the desired goal. A foundational idea in learning is represented in RL, such that an agent can delay immediate rewards to gain potential larger future rewards. Therefore, RL Agents attempt to maximise the expected value of G_t with discount factor γ.

$$G_t = \sum_{t=1}^{n} \gamma^k r_{t+k+1} \tag{2.2}$$

Reinforcement learning differs from supervised learning in that we are maximising some policy through interaction, using partial feedback from an environment, rather than learning from a set of examples picked by some knowledgeable provider. In many cases, we can tabulate a direct state-action mapping when the problem is simple, and this works exceptionally well in finding the optimal policy. An example might be simply mapping each board combination to the optimal action in tic-tac-toe. Often, the complete set of unique observations is too large to be simply mapped to actions. Examples of massive state domains might be road images in a self-driving car or the combinations of a chessboard. It is for this reason that Reinforcement learning uses function approximation to find a good policy. Function approximation and sampling over an extremely large, unknown and stochastic data set to create a competent agent is what makes reinforcement learning powerful.

Two main branches of RL algorithms exist; model-based and model-free algorithms. A model is the complete set of states, actions and reward probabilities. Model-free approaches use the model to iterate over states (value iteration) or iterate over a policy to find the optimal policy (policy iteration). If an accurate model of the environment is not provided, model-free approaches as outlined in this paper are used. Most state of the art RL deals with model-free algorithms, as many interesting environments don't have accurate models, therefore policies or value functions must be approximated. Within model-free algorithms, the approaches are either on-policy or off policy. On-policy algorithms learn from the actions carried out by the agent, whereas off-policy algorithms learn the policy independently of the agent's policy actions. An on-policy algorithm uses the next best action and learns from it while an off-policy algorithm updates the policy given the next best action, but doesn't necessarily select it.

2.2.1 Actions, States and Rewards

How an MDP is formulated by design is essential in creating an effective Reinforcement Learning agent. States and actions may be a complete representation of the problem, or they may be a higher-level feature selection. This project will deal with a finite MDP, where there is a limited number of states from $t = 0$ to $t = n$. Each environment has an overall goal, which, when accomplished, terminates the episode. Secondly, states action transition probabilities are always 1, the action selected by the agent deterministically determines the next state. High-level feature representations of states may speed-up learning in simpler environments; however, it may no longer be obvious what is transient and what remains unchanged in more complex problems. The full complexity of the problem may be lost, and the agent is unable to move towards better policies. A highlighted example is in [17] using the full-pixel representation of 2D Atari game environment, passed through a CNN with stacks of 4 frames caused significantly slower training than using the raw memory array. While the full pixel array makes more sense to a human observer, it contains large amounts of unnecessary information. Continuous action spaces are useful in robotic control when the movements can't be discretised easily over time. However, they are a much harder problem for deep RL agents to solve.

In RL, the agent's purpose or goal is to receive a signal passed from the environment to the agent. The agent's goal is to maximise the total cumulative reward it receives throughout the full episode. Consequently, this means the agent should delay immediate rewards for larger future rewards. The use of a reward function makes RL distinct and interestingly analogous to animal learning with external stimuli. Rewards should be designed such that for the agent to maximise its reward means solving the MDP optimally. A sub-goal which the agent may exploit repeatedly should not be rewarded at a premium to completion of the full task. An example, discovered during testing demonstrated this: The robotic arm was rewarded when reaching the goal, upon which the episode terminated after a number of steps on target. If the goal was not reached, the episode would time out. The agent, therefore, learnt to move on and off the goal to increase the episode's cumulative reward.

Two reward functions which, when maximised, result in optimal control can have very different learning outcomes. Assigning rewards for an agent is a difficult task. If rewards are too sparse, an agent may fail to learn. If we give too many intermediate rewards, hoping our agent learns the outcome, it may exploit this for unexpected results. Take a maze where the agent is rewarded for collecting spawning goals and reaching some exit, both rewarded. Depending on the magnitude of each reward, the agent could just spend its time collecting goals or going straight for the exit. As outlined in [26], we can punish until the overall completion of the task. However, this may cause the rewards to be very sparse in long, complex tasks. Another example may be just collecting goals in a grid environment terminating with all goals found. We can punish the agent -1 per step as long as the task is incomplete, or, we can reward upon picking up the points of interest with a +1 reward as shown in [14] in a similar GridWorld. If we reward goal capture, these are intermediate rewards and can potentially be exploited. However, if we punish until completion, or conversely reward only at completion, the reward signal is very sparse in large domains. It then becomes difficult for the agent to infer which exact intermediate actions, now many steps in the past, that led to this result. The importance of temporal assignment of rewards is known as the credit assignment problem introduced by Sutton [25], and is a topic of research in itself. We want our reward function to be not too sparse, while also not easily exploited. When designing a reward function, the magnitude of reward is not of importance, only the relative difference between rewards. This is also shown in [26].

Similarly, the states encoding has an impact on how an agent map states to actions. States should be unique such that every unique positioning of the state is mapped to only one state encoding. A simple encoding might be the agent's position and enemy positions on a grid or, we could use the full 2d grid encoded to integers for different grid cell occupants (agents, empty, goals etc.). However, as later shown, this creates the problem of representing categorical data with numerical order. This can be fixed by converting original representations to 1-hot vectors, as is common throughout Machine Learning.

2.2.2 OpenAI Gym

OpenAI gym [6] provides a framework for implementing MDP's for Reinforcement Learning. This framework allows for the disjoint development of algorithms and environments. A new algorithm can be easily implemented against existing or new environments which fit the Gym interface. The gym interface is simple and requires only four functions, which provide all interaction with, and information from the MDP.

- Initialize - Initialisation of environment variables, the shape of action and observation space
- Step - Takes an input action and returns a tuple (s_t, r_t, d) of the new state, the reward and boolean if the environment is done
- Reset - Resets the environment to its initial state, not necessarily deterministic
- Render - Rendering of the environment at each step for debugging or evaluation.

A gym environment wraps an MDP process in a set of callable functions which are used by the RL algorithm. First, the environment is initialised, and the agent receives the state shape, action shape and the initial state observation. The agent uses the observation shape to create the shape of its value function table or neural network input. The action shape is used to create the output of the network or the second dimension of the value table. The agent then chooses an action based on the observation and the action-functions output. The environment receives the agent's action, updates its inner state and then returns the tuple with a possible reward $r > 0$ or punishment $r < 0$. The agent uses the state-reward feedback to update its policy. Most modern RL literature uses this approach due to the easy interchangeability between both environments and algorithms - a single algorithm is easily tested against multiple environments and vice versa.

2.2.3 Optimal Policies

A value function, denoted $v_\pi(s)$ is a learnt function, which measures the utility of the agent being in a given state. It measures the agent's expected return, starting in s and following policy π onward. An optimal policy defines the solution to a finite MDP which maximises the cumulative reward throughout every possible episode. A policy π is defined to be better or equal to another policy π' if its expected return is greater than or equal to that of pi'. There is always at least one policy that is better than all other policies. This policy is known as an optimal policy, denoted pi_*. All optimal policies, share the same state value function $v_*(s)$ or action-value function $Q_*(s,a)$.

The bellman optimality equation for Q_* describes that the value of a state, following the optimal policy is equal to the expected return for the best action from that state and onwards.

$$Q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max Q_*(s',a')\right] \tag{2.3}$$

$Q_*(s,a)$ here is often more useful than $v_*(s)$ as this is a function of both states and actions, meaning our agent can know the best action without looking forward to future states values. The idea behind 2.3 is that our optimal policy is found by following the current reward, plus the discounted future reward saved in some memory. This recursive nature means we can build a mapping by considering our current and the next states we move to, in a tabular setting, we can use an algorithm such as Q-Learning to learn this. Q-Learning has been shown [31] to converge to an optimal policy.

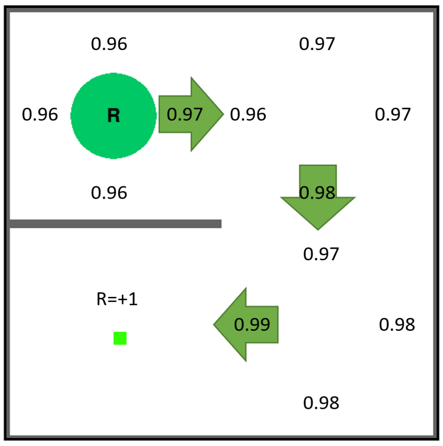

Figure 2.2: Tabular Q-learning with an optimal policy on a 2x2 maze (table mapped to corresponding states). Each grid coordinate has all values of $Q(s, a)$, calculated using 2.7

Generally, such an optimal policy takes longer to find in more complex settings. To solve a reinforcement learning problem means finding a policy which maximises the Bellman optimality equation 2.3. In the tabular setting, parallels can be drawn with an exhaustive search over all possible states multiple times. To be able to solve this, we must have an environment which is easily encoded to a table and that we have enough resources to run multiple trials. Often, an optimal solution will not be vital. When this is the case, we can use function approximators to approximate the optimal policy.

2.3 Deep Neural Networks in RL

Previously divergence was shown to be an issue in training neural networks [5]. However, recent advancements and impressive results in modern reinforcement learning have proven neural networks the approximator of choice [17] denoted as a parameter vector θ. Deep networks allow reinforcement learning algorithms to train on large sets of data by approximating value functions or parametrising a policy. A neural network is a set of neurons containing inputs, input weights, bias and an activation function. Weights and biases sum together, and the neuron's output passes through a non-linear activation function. The activation function allows the network to predict large non-linear mappings. The network θ is a function of the trainable weights and biases of each neuron, of which there may be multiple in each layer. In a multi-layered deep network used for Deep RL, there are typically thousands of trainable weights and biases. While this seems like a lot, this value is likely much smaller than the total number of states in the MDP, meaning it is a much more efficient representation than using one to one table mapping. Neural networks have been used in various reinforcement learning methods [8] to varying levels of success, and are the de-facto method of function approximation in modern Deep Reinforcement Learning.

Training neural networks involves two main steps: Forward propagation to calculate the outputs, and therefore the loss. And backpropagation, to propagate the errors back throughout the network to calculate the error with respect to each weight and bias in order to decrease the overall model's loss. The neural network is a differentiable function of functions (of weights, biases and the activation functions), and therefore the gradient can be calculated. In backpropagation, the error gradient is calculated to move the weights such that we minimise some error function. Training neural networks is typically done in a supervised learning setting. This means, all the data is present, with known labels which we are trying to predict. We can minimise some loss function such as mean squared error, with multiple sampled training epochs over the whole dataset. This is not as simple in an RL setting as there is not, and may never be a complete data set of every action, state, reward mapping. Therefore, each deep RL algorithm requires some sampling strategy to collect samples of states and actions taken. These samples are then used to move the network towards a set of more desirable state action mappings. Strategies include sampling batches of recently seen state transitions, known as experience replay as in [17] or calculating a full episodes expected reward for each state until the end as in PPO [22]. Such sampling requires either the agent or network to implement an exploration strategy such that a wide breadth of actions and states are visited to guarantee convergence to the optimal policy at a global minima.

2.4 Value Based Methods

As outlined in Sutton and Barto [26], a method behind many Reinforcement Learning algorithms to learn value functions. Value functions are a models estimation of how good a state is $v(s)$, or how good an action, given the state is $Q(s, a)$. Solving a Reinforcement Learning task involves finding, or approximating, an optimal value function which leads to an optimal or approximately optimal policy. We often make a trade-off when state spaces are large for an approximately optimal policy. There may be more than one optimal policy. However, all optimal policies will share one state-value function, as two equally good paths to a goal will be equally as good in the total state-value function. The optimal state value function in reinforcement learning is defined as :

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.4}$$

Similarly, optimal policies also share an optimal action-value function, known as Q-values as in Watkins Q-learning [31]

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{2.5}$$

Simply put, taking the action with the highest Q-value in each state leads to the optimal policy. The Optimal Q-values satisfy the recursive Bellman Optimality Equations [2]

$$Q^*(s, a) = \mathbb{E}\left[R_{t+1} + \gamma \max_a Q^*(s_{t+1}, a')\right] \tag{2.6}$$

Where a' is the next states best action. The idea here is that optimal value function is equal to the expected return of taking action a in s and following states by following the optimal policy. Watkins Q-Learning [31], a foundational algorithm uses the Bellman Equation to update the Q-values. This algorithm uses off-policy updates, such that the update action used to train the Q table is not necessarily the action taken at step t. The idea behind the algorithm is that it selects actions from the table of values greedily, with some random ϵ chance of exploration. This action is then performed to receive the next state. The table is updated with the immediate reward and the Q-value of the next states best action, or the discounted reward. Values in the table are initialised to an arbitrary value and updated using the Bellman equation. We can iteratively approximate $Q*$ using the Q-learning update.

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.7}$$

Where α is a constant learning rate. Keeping track of state values or state-action values, $Q(s, a)$, can be done using a table, with an entry for each state or each state action. $Q(s, a)$ in 2.7 represents an element in the row s at column a. As the environment grows in complexity, the limitations of a table are obvious. In large state domains or without a complete model of the environment, finding a good approximate tabular value function is not possible.

For value-based methods to work, they rely on both following their policy and exploring. Too much exploration and the policy doesn't learn anything useful, too little, and new actions and states are not found. This is common ϵ-greedy approach, as in Q-learning, where a random action is taken with some small probability at timestep. Over time, this can be annealed to a lesser value or kept constant.

2.4.1 Deep Q Learning

The 2013 paper [17] released by DeepMind highlighted the power of using Deep Neural Networks to approximate $Q(s, a)$. This off policy, value based method is known as Deep Q Learning, or Deep Q Networks (DQN). In the 2013 paper, agents were trained using a state space of the Atari 2D pixel array, using a convolution neural network as the approximate value function which was very successful. DQN learnt policies which were above human performance in most of the Atari environments. The algorithm, evident in the name, is founded in Watkins Q-Learning [31] and is a value-based approach, which trains off policy. The algorithm attempts to find the optimal value-action function $Q(s, a)$. Many variations of the Q-Learning algorithm have been introduced; such as prioritised experience replay sampling [21] or two twin networks to reduce state value overestimation [28]. Prioritised experience replay uses the TD error, where more informative states with higher TD error are sampled for learning at a higher rate. Deep Q learning obeys the identity outlined in 2.6, which is the maximum expected reward following $Q*$.

Comparable to tabular Q learning, the algorithm instead uses the neural network to compute the current Q-value and the next states best Q-value as the target. Similarly, it uses an ϵ-greedy approach to exploration, which often decays throughout the training. Choosing moves at randomly with some small probability ϵ, otherwise, selecting the action with the highest Q-value. The action is applied, and then the state action, reward and next state (s, a, r, s') are stored in the experience replay buffer. Then, a minibatch of transitions is sampled which are used for training the Q-network. Stochastic gradient descent is applied to this minibatch to minimise the mean squared error. The networks input is the state s and outputs a set of action values A, from which the best action is chosen if not exploring.

$$L(\theta) = [(y_i - Q(s, a))^2] \tag{2.8}$$

where $y_i = E(r + \gamma \max_{a'} Q(s', a'))$. We note here that using $Q(s', a')$ makes the algorithm offline, as the target values of the network are not necessarily those of the action it takes. Further, both the target and Q value depend on the neural network, which is an interesting difference from typical supervised learning. We can think of this as moving towards better, already known actions, updated with the reward of the current state. At each step of the algorithm, an action is selected and then performed, observing the (s_t, a_t, r_t, s_{t+1}) these transitions are stored in memory called experience replay. The experience replay is first initialised to some capacity with the arbitrary initial policy. Once full, the actions are selected based on the policy and all-new transitions are stored in the fixed-length queue. Optimisation of the loss function is performed using stochastic gradient descent over a mini-batch sampled evenly from the experience replay. From here, a new action is performed, and the process repeats.

The results of Deepmind's 2013 publication are significant, as it was shown that reinforcement learning could perform on very complex state spaces. The results also showed how an algorithm could learn as would a human, with only the pixel state observation. The distribution of possible states is very large when using pixel only values. However, this algorithm fails to learn in certain environments with sparse rewards where episodes are long, and there is a reward after a set of complex tasks. Deep Q learning is the only value-based method considered in this paper. Adaptations of Deep Q learning use features such as two Q-networks to reduce estimation bias [28] or prioritised experience replay[21].

2.5 Policy Gradient Methods

In policy gradient methods, a policy is searched for directly without the intermediate measurement of the value of a state. The agent observes, and our parametrised function θ outputs actions directly. The goal is that we want our agent to take actions which maximise some objective function $J(\theta)$. The objective function is like a loss function, used in the optimisation of θ. To get the gradient of our function, it must once again be a differentiable neural network. By taking the gradient of our policy function, $\pi_\theta(a|s)$ parametrised by a neural network θ, we can move our policy towards better actions. We note here that that the steepest direction of change of function is the gradient. Therefore, by performing gradient ascent on an objective function, we are constantly moving the policy towards better actions. These gradients are easily calculated in most neural network libraries.

Policy gradients are particularly useful over value-based methods for multiple reasons. One, the policies output are stochastic. Value-based methods can't learn stochastic policies and must rely on an e-greedy approach to exploration. In some cases, stochastic policies may be optimal, take the game of paper, scissors, rock, for example, where the best strategy is to pick moves at random such that the opponent can't learn your strategy. Secondly, the outputs of our function approximator can be applied directly to actions in continuous action spaces, something which is not possible in value-based approaches. In policy gradients we want to maximise some performance measure, which is often the performance from the start state of the episode, as in [26]

$$J(\theta) = v_{\pi_\theta}(s_0) \tag{2.9}$$

Where v_θ is the true value function for π_θ the policy. It is also useful to know that the value function for a given state is the sum of its actions by its action probabilities.

$$V_\pi(s) = \sum_a \pi(a|s) Q_\pi(s,a) \tag{2.10}$$

Where $\pi(a|s)$ is the probability of taking action a in state s. And $Q_\pi(s,a)$ is a value function similar to that of (2.7). Using this, and the policy gradient theorem as outlined in [26](Chapter 13) we can arrive at the general update rule for the network.

$$\theta_{t+1} = \theta_t + \alpha G_t \nabla \ln \pi(a_t|s_t,\theta) \tag{2.11}$$

This update rule pushes the network to update the probability of actions $\pi(a|s,\theta)$ proportional to the guesses of these actions. The idea here is that we cannot only update actions we are confident about as there may be better actions which are less likely to be picked. A detailed derivation of how equation 2.11 is derived from 2.10 is shown in [26]. This is known as the policy gradient theorem and is an important idea in policy gradient Deep RL algorithms. Policy-based methods work well, and simply applying the update rule is the basis for the vanilla policy gradient algorithm (VPG). However, modern algorithms use a mix of both policy and value-based methods.

2.5.1 Actor Critic Methods (A2C)

Actor critic methods mix both policy and value-based methods and are a set of popular off-policy algorithms in deep reinforcement learning. Both the actor and the critic are separate parametrised functions. The actor computes the policy $\pi_\theta(a|s)$ and chooses actions in the environment based on its parametrised and learnt policy. The critic computes the value functions, $Q(s,a)$ and $V(s,a)$ which evaluate the state action or state value or advantage functions. Many Deep reinforcement learning algorithms make use of these separate functions, including A3C, DDPG, PPO, TRPO, A2C among others.

The new policy update involves two functions which must be optimized separately. The policy update is similar to a the standard policy gradient update.

$$\theta_{t+1} = \theta_t + \alpha(\ln \pi(a_t|s_t,\theta) Q(s,a)) \tag{2.12}$$

Where the Q function is the temporal difference target as in Q-learning or other value-based approaches. In order to compute this, we need to constantly update our estimate of the state value function by optimising the weights of our value function.

$$\Delta w = \beta(R_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s,a)) \nabla Q(s_t, a_t) \tag{2.13}$$

Where θ and w are the respective weights of the policy network and value network parameters. With α and β their respective learning rates. We note here that one is a vaue network and one a policy network. The second equation is the TD error by the gradient of our current state action value estimate, which is similar to the Q-learning update in 2.7. This equals the change in our weights. As in regular policy gradient methods, the critic replaces the r_t value in our policy update, which is the update proportional update with respect to the probability of the actions, with an advantage function. The advantage function tells us the improvement of the action, compared to the average action taken in that state.

$$A(s,a) = Q(s,a) - V(s) \tag{2.14}$$

Both $V(s)$ and $Q(s,a)$ are computed using the temporal difference target

$$A(s,a) = r_{t+1} + \gamma V(s_t + 1) - V(s_t) \tag{2.15}$$

Which is equivalent to the equation 2.14. With this, we can use a single parametrised network for both $Q(s,a)$ and $V(s)$ for the critic. This is trained using the error between the current TD target of a state and the current parametrised state value during an episode rollout.

This critic Asynchronous Advantage Actor critic, A3C [15] from Deepmind, executes a number of agents in parallel on separate environments to increase sample diversity and the number of gradient updates. The policy gradient updates use an advantage function, which is usually $Q(s,a) - V(s)$ using the n-step returns on prior complete rollouts of the environment. The synchronous equivalent, OpenAI's A2C [32] simply does these updates sequentially and is found to have similar performance despite a slower training rate.

2.5.2 Proximal Policy Optimization (PPO)

Proximal policy optimization (PPO) [22] is a simpler, yet equally powerful version of Trust Region Policy Optimization (TRPO) [23]. Its simplicity and strong performance, both continuous and discrete action space make it a popular modern RL algorithm. Both algorithms attempt to push a policy by the largest possible amount towards the desired action. PPO is shown to perform as well as TRPO while being much simpler to implement. Both algorithms are on policy and can output both discrete and continuous actions by outputting a probability distribution. PPO introduces two main contributions on top of a vanilla policy method, which are the clipped surrogate objective and the use of multiple epochs of stochastic gradient ascent for each policy update.

The clipped surrogate objective is used to improve training stability by limiting the update at each step to the policy. From the vanilla policy gradient, as in [26], the network is pushed towards better actions. The VPG algorithm uses the log probability of the action to lower the impact of an actions update on the network. If an action is much more probable, the update could be too large and ruin the current policy. PPO, for this reason, introduces the clipped surrogate objective.

$$L_{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right] \tag{2.16}$$

where $r_t(\theta)$ is the objective term by the advantage function. The objective term, equal to the log probability of an action as in vanilla policy gradients 2.11.

$$r_t = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \tag{2.17}$$

Which is the probability of taking an action in the current state, divided by the probability of taking that action under the previous policy. The advantage function: A_t, as in 2.14 is used here. This is so more probable actions under the new policy generate larger updates, and less probable actions generate smaller ones. This is also a learnt network. This can lead to a problem, if the action is much more probable under the new policy, a large gradient update can ruin the current policy, causing divergence. Therefore, PPO uses a clipped surrogate objective, the second part in 2.16. If an action was good, and $A > 0$, the gradient should only be updated so much, as the r value is clipped when too high. That is we do not want the algorithm to act too greedily, having found a great action. If the action was bad $A < 0$, and it also became less probable $r_t < 1$ then do not make it too much less probable, i.e. the update is clipped. If it was bad and became more probable, then there is no clipping, and the damage is undone.

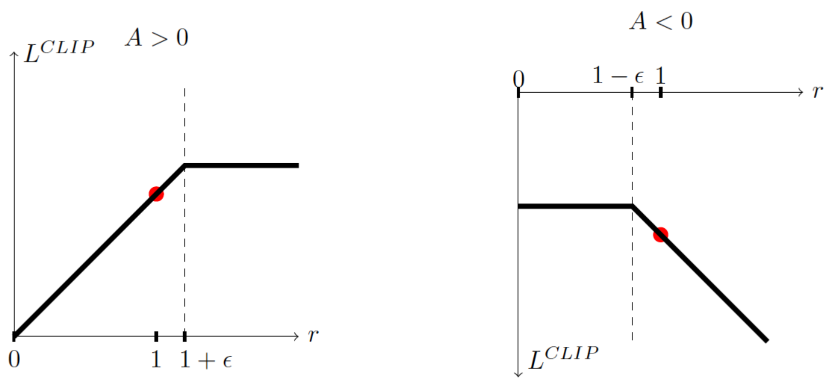

Figure 2.3: The two cases of the clipped objective. Taken from PPO paper

The second main contribution of the paper is the use of multiple epochs of stochastic gradient descent. As the updates are clipped, multiple epochs of updates allow for multiple smaller and smarter updates which won't wreak havoc on the policy network. This also allows for easy parallelism with distributed updates. This method became popular with A3C [15]. All distributed gradient updates are calculated and then applied until there is convergence, and the clipping is being hit, on the current sample. Then the algorithm collects new samples, and the gradient updates restart.

2.6 Deep Deterministic Policy Gradients (DDPG)

Deep Deterministic Policy Gradients [12](DDPG) is an Actor-Critic algorithm which learns a Q function and a policy on separate networks. It is an off-policy algorithm, and samples state transitions from a replay buffer for training. As the name suggests, the policies generated are deterministic and not stochastic as in other policy gradient approaches. Meaning for each state, the policy can only output one action. The algorithm performs only in continuous action spaces. It is often considered the continuous action space version of Deep Q learning as the algorithm learns an approximate optimal Q function $Q^*(s,a)$, as in DQN, and similarly uses a replay buffer. Conversely to DQN, we cannot simply output the action space and take the max of our table or function approximator.

The algorithm uses a critic network $Q(s,a|\theta^Q)$ and an actor-network $\mu(s|\theta^\mu)$ with parameter weights θ^Q and θ^μ. Each of these networks has a target network which is periodically updated. The critic network outputs a single, continuous value $Q(s,a)$ given both the state and actions as input. These actions come from the actor-network, which outputs a set of continuous actions given the states. The critic or value network is trained similar to the Q-network in DQN that is we want to minimise the squared loss over a sample batch of N transitions

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{2.18}$$

Where $y_i = r_i + \gamma Q'(s_{i+1}, \pi(s_{i+1}|\theta')$. Note that instead of using the action a in the samples, we are using the output of the actor network, the policy.

The actor is trained using the gradient of the critic and is based on the deterministic policy gradient theorem [24]. The policy networks objective is to maximise the expected return.

$$J(\theta) = \mathbb{E}\left[Q(s,\mu_t)\right] \tag{2.19}$$

Keeping in mind here, that the action μ is the networks output action, and not the action stored in the replay buffer. This action has the added noise, which is the algorithms method of exploration.

Using the gradient to find the largest policy increase, we can derive the following using the chain rule and then take the average over multiple samples.

$$\nabla_{\theta^\mu} J(\theta) = \frac{1}{N} \sum_i \left(\nabla_a Q(s_i, a_i) \nabla_{\theta^\mu} \mu(s|\theta^\mu) \right) \tag{2.20}$$

This gives us the objective function, which is negated and used as the policy loss for optimisation of the actor-network.

The target network updates use soft updates, with the parameter τ. These networks slowly track the learned networks and are used in calculating the critic or value function. The networks are updated with the following rule.

$$\theta^{'} \leftarrow \tau\theta + (1-\tau)\theta^{'} \tag{2.21}$$

Contrary to deterministic action spaces, exploration in DDPG is done via sampling and adding noise to the action itself. In the original paper [12] this is Ornstein Uhlenbeck noise. Ornstein Uhlenbeck noise is an annealing process, similar random walk noise where the deviation tends to revert to the mean over time. However, some results have shown Gaussian noise has been found to work well [9]. The original DDPG paper uses OU noise as in many robotic control situations. Parameter space noise has also shown promising results [20]. Here, noise is added directly to the parameters of the actor-network, rather than the output actions. As in DQN, recent iterations of DDPG include adaptations such as prioritised experience replay or noisy exploration networks.

2.6.1 Twin Delayed Deep Deterministic Policy Gradients (TD3)

Twin delayed deterministic policy gradients, or, TD3 [7] is often known as the successor to DDPG and is a very similar algorithm. It is also an off-policy algorithm and a deterministic policy gradient method, meaning each state has an exact action output, rather than a probability distribution. TD3 aims to solve a number of problems which are present in DDPG, as shown in [7]. DDPG often overestimates the Q-value or critics state value estimation of the expected discounted reward. The overestimations accumulate throughout training and force the agent into local optima due to overestimation bias of certain states or state sequences. The true discounted reward following the current policy was found to be much lower than the actual predicted Q-value by the critic. To solve this overestimation, the authors suggest using a pair of critic networks $Q_\theta 1$ and $Q_\theta 2$ and a policy network π_ϕ along with their respective target networks $\theta 1'$ and $\theta 2'$ and ϕ. Policy updates are also delayed. The twin critic networks, inspired by Double Q learning [28], are used to reduce overestimation by taking the minimum of their respective outputs.

The algorithm first selects an action with exploration noise, stores the transition and then trains its network over a batch of several iterations N. The critic is trained using a target action a' taken from the target actor-network on stored replay buffer states. The target action has noise added and then is clipped to ensure it's not too far away from the original action value. With this target action and the next state from the buffer, the target Q-values are computed using the target critic networks, from which the minimum is selected. This is used to train the critic network using gradient descent.

$$y \leftarrow r + \gamma \min\{Q_{\theta'_1}(s', a'), Q_{\theta'_2}(s', a')\} \tag{2.22}$$

$$\text{Critics updated: } \theta_i \leftarrow \text{argmin}_{\theta_i} \frac{1}{N} \sum_N (y - Q_{\phi_i}(s,a))^2 \tag{2.23}$$

Next, target and policy network updates are delayed, such that these networks are updated about half as much as the Q-network. This introduces stability such that the target networks policies don't fluctuate and results in more stable and efficient training. These are updated using the deterministic policy gradient update as in 2.20. These features have been shown to produce more stable training in a range of environments over DDPG.

2.7 Curriculum Learning

Curriculum learning [3] in machine learning introduces the idea of a curriculum for different optimisation problems. These problems, such as training a neural network, are given increasingly more difficult problems. The idea is that the problem may be too hard or, rewards too sparse from the initial state. It was found that introducing more difficult training examples gradually, can in some cases, speed-up training. Subtasks are to be less difficult than the final task and quantitatively easier.

In this paper, a naive curriculum learning approach is given in the environments. Two approaches to curriculum learning are implemented. Either the opposing agent becomes increasingly more difficult, or the size of the state space is increased directly.

Chapter 3

Proposed Environments

This paper proposes a set of reinforcement learning environments which scale in difficulty through growing their state or action sizes. Each environment contains a baseline which is found through a known planning approach or closed-form solution. Other, less tested environments are included in the repository. However, this report focuses on those with a known baseline. A detailed description of each environment's configurations and documentation can be found on the GitHub page[1]. Each environment contains multiple state encodings, reward signals and an array of parameters for which careful testing is necessary in order to identify how environment state encodings, rewards and algorithms interact.

3.1 N-Joint Reacher

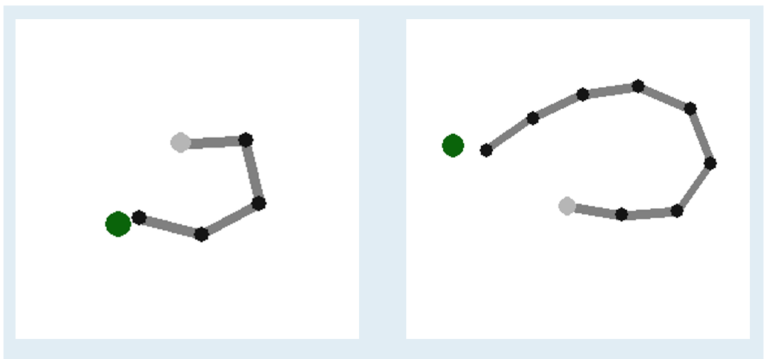

Figure 3.1: A 2 jointed Reacher and a 7 jointed reacher

The N-joint Manipulator is a classic control robotics problem. The goal of the environment is to configure a multi-jointed reacher to some point of interest in as few steps as possible. The environment's difficulty grows when scaling the number of joints due to the flow-on effects of moving multiple joints to the end effector. A small movement at the beginning of the arm can cause large changes at the end. With multiple additive joints, it is evident how this becomes difficult to control. Both the action and state spaces are continuous vectors of each joint's position. The vectors grow in size with the number of joints. When the reacher is longer with more joints, the possible positions of the goal grows with the maximum as the maximum radius of the sum of all connected arms. The action space is an angular step which is clipped by a maximum amount or minimum amount (both directions), which is the maximum allowed movement per environment step. The reward is the negative normalised distance from the goal, minus an action penalty.

[1]The source code, all environments and documentation can be found at: github.com/kinseyreeves/Deep-RL-Baselines

$$R = -d - \beta \sum_{i=1}^{n} a_i \tag{3.1}$$

Where n is the number of joints, d the normalised distance from goal and β a tunable parameter. The theory behind this reward function is that convex functions are preferable in optimisation. This holds true here as 3.1 was found to be more effective than a binary reward function. In each episode, the effector begins from a random position and must reach the goal. The arms position is used from prior episodes to achieve this randomisation. This is significantly harder than fixed and equal starting angles. The environment is finished when the end effector has positioned itself inside of the objective for a fixed amount of steps.

The environment was chosen as it has an analytic, optimal baseline to measure against. Many similar physics environments exist in the OpenAI library, although without similar analytic solutions. Therefore, this baseline serves as a guideline for finding effective reinforcement learning models in similarly continuous action and state spaces where no analytic solution can be used.

Multi-jointed systems are a common problem within the reinforcement learning literature. Wherever a continuous RL algorithm is proposed, it is usually presented with a humanoid or animal control problem. Suites of environments exist, such as MuJuco [27] for similar control problems. In these settings, two issues questions arise which this environment attempts to answer. Firstly, Optimal solutions are only ever assumed when a solution looks good, or it converges to some optima. How can one confidently assess how close we are to the true optima? And secondly, how does the complexity of the environment affect the convergence towards this optima?

3.1.1 Efficient State Encoding

Two state encodings for the N-jointed reacher are provided: First, the positions of each moving joint and goal. Secondly, the positions and angles of each joint and the goal. The motivation for including the second was to represent the problem similar to what is required when calculating the inverse kinematics baseline. It was found that the positional only representation resulted in the most efficient training.

3.2 Grid World

The next set of environments exist within a common grid-world. These environments all share an action space of North, South, East, West and have different rewards based on the individual problem. If an agent selects a move that is not possible, it will remain in place. These environments scale in difficulty by increasing the size of the grid, therefore increasing the state space. The baselines for each are based around optimal search and planning algorithms. A number of configurations are available for each grid environment to vary the complexity of the environment. The starting positions of the entity and goals or opposing entity can be set fixed or random. Environments are easily created by the user in a text file, which is then loaded by the environment[2]. The maze generation is based on Prim's algorithm, with some probability of deleting walls. Generation of all entity positions: goals, enemies, or the agent, is either fixed or based on a uniform distribution over all maze positions. For testing, all entity positions were randomised after each episode. Non-randomised positions result in a minimal explored state space, as the algorithm only has a short path to learn. Therefore, this configuration is less appealing, and the environment is only considered in the randomised start and goal setting.

Reward functions are based around rewarding on goal where possible, rather than punishing as is used in the world in [26]. This is due to the intermediate reward allows an agent to more easily learn intermediate steps to the overall completion goal. As the algorithms acting in continuous action spaces are policy gradient methods, this seems to improve learning, as later shown. Rewarding only on the overall completion, or punishing until termination, forces the agent to infer the correct actions from an equal set of all the episodes state-action pairs. In this case, it was found intermediate rewards for each goal aided in learning, as shown later in the results.

[2]A script for generating mazes of any size is provided, and a set of mazes which are used in this paper are included in the repository

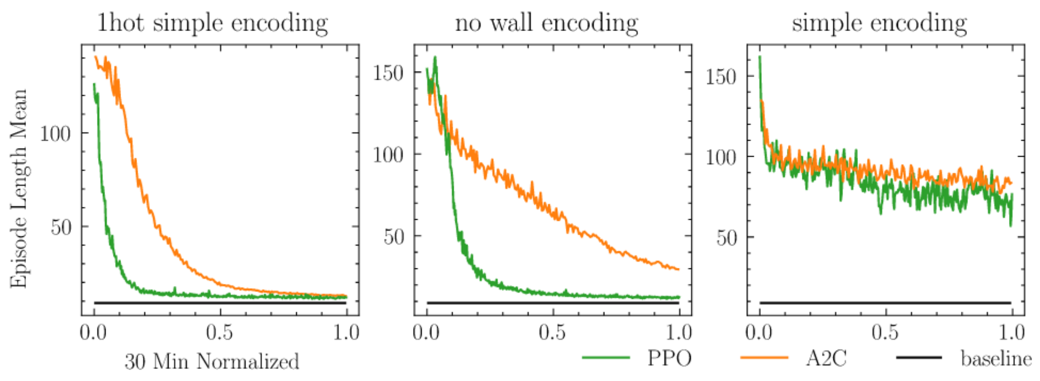

Figure 3.2: Different state encoding trained on a 5x5 maze with 3 goals

Efficient State Encoding

The design of the Grid environment's state encoding required some thought, with problems in each encoding. In RL literature, grid state spaces are generally positional and may include some map data. Typical Grid-World implementations across RL literature often only include the agent's position, as in Suttons Grid-world [26]. These representations are mainly used in discussing tabular reinforcement learning algorithms. With multiple entities (goals, the agent and the enemy), we need a smarter solution. Effective state encodings should ensure that unique states have only one representation and do not include unnecessary information. A trade-off must accordingly be made, between efficient and concise representation. As in typical Machine Learning, we want to avoid encodings which represent different binary features in a continuous encoding. That is, we do not want categorical differences to be represented ordinally. We do this by converting the encoding to a 1-hot sparse vector. For each encoding, there is a sparse representation. Ideally, 1-hot vectors are used when possible; however, converting a multidimensional grid with possible values ranging from 0 to the number of entities plus goals creates a very sparse 3-dimensional array. The state is initially loaded from a file using characters to represent different grid positions and entities. From here it is converted to each separate encoding.

Three possible state encodings were considered for training.

- Simple encoding: The positions of the RL controlled entity and that of the points of interest as mentioned (other entities or goals). Goal positions are set to 0 if non-existent and sorted to remove the symmetric encoding. 1 is appended to the array if a goal was captured.
- No wall encoding: An array of walkable positions only, encoded to integers.
- Full wall encoding: a full encoded array of the grid map all encoded to numeric values, including all walls, directly from the internal representation. Each cell assigned an integer value.
- 1-hot vector encoding. Encodes the simple encoding to a 1 hot vector of shape entities by map size

Each state encoding offers a differing level of simplicity from only highly relevant information, i.e. a vector of the position of the entity and its goals, a full matrix of walkable positions or 1-hot positional encoding including information about goal capture. Including the walls in encodings led to a very large state encoding (16x16) for an (8x8) maze (as walls are counted as a block). However, it shows the full complexity of the problem and is represented with everything required for a human to play.

Figure 3.2 shows the different encodings trained on two of the top-performing algorithms A2C and PPO. Contradicting what was initially thought to be the most efficient encoding, the wall removed encoding of only walkable positions showed better results than the entity positions. The simple positional encoding caused slower training with much higher variance. This is likely due to representing coordinate positional data as ordinal. The removal of walls removes some information; however, the reduction in state size is reflected in the faster training convergence. We also see the benefit of the use of 1-hot encoding over the full grid or no wall encoding. Each possible grid position is encoded by an integer 1 through 6. (unoccupied, RL agent, goal etc.). This causes the problem that two separate entities are represented as numerically close to each other in the same dimension. It is for this reason we see the performance boost of 1-hot encoding. The 1-hot encoding encodes the simple positional vector to a 1-hot matrix. Each different combination is represented as a binary value in its own dimension. It is for this reason that the 1-hot simple encoding was chosen for testing, as it efficiently and succinctly represents the state observation.

3.2.1 Simple Maze

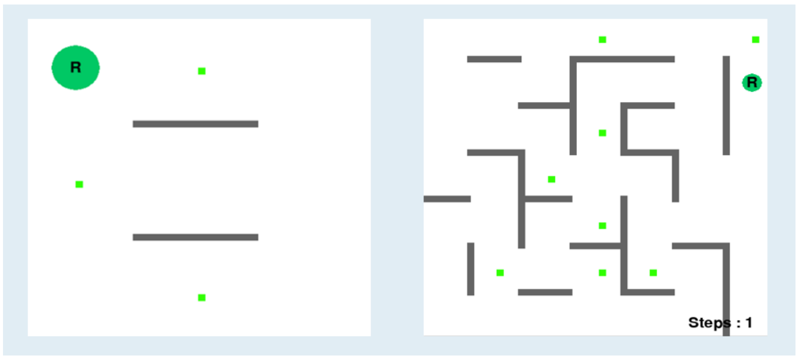

Figure 3.3: A 3x3 maze with 3 goals, and a 7x7 maze with 8 goals

The maze environment's goal is to move the agent towards one or multiple points of interest as efficiently as possible. The optimal solution to the problem is the shortest path or trip from the start to all goals. This problem is known as the travelling salesman problem (TSP). TSP is considered an NP-hard problem in combinatorial optimisation. The goal is to move an agent, or salesman throughout a set of goals in the shortest possible distance. The one difference here is that the agent person must not return to the start state. As the number of goals N grows, the brute force approach included, guarantees a global optima in exponential time $O(n2^n)$. This is obviously inefficient and we see why we might want an approximate approach. Other exact solutions can perform better, but all remain exponential in complexity. Multiple configurations of the maze environment are given including single goal and multiple goals and can be scaled as with the other grid environments. The maze environment provides complexity scaling in both size and number of goals. The number of possible unique states in the environment grows as the map size grows.

$$S_{unique} = \sum_{i=1}^{g} \binom{n}{i} (n-i) \tag{3.2}$$

Map size	Positions	Unique States g=1	Unique States g=3	Avg. Baseline Steps g=1	Avg. Baseline Steps, g=3
3	9	72	855	2.14	4.97
4	16	240	9304	2.90	6.91
5	25	600	58375	3.77	8.99
6	36	1260	258894	4.76	13.6
7	49	2352	906255	6.09	14.07
8	64	4032	2672480	7.51	19.91
9	81	6480	6920559	9.33	23.42

Table 3.1: Table of the growth of unique state size in the grid environment with each levels respective TSP baseline. 3 goals and 1 goal both shown

Equation 3.2 provides the number of unique states in a maze of size n with g goals. We can also use this to calculate the chaser and evader environments unique states by setting $g = 1$. In the maze environment, the agent is rewarded when it collects each goal (in some cases of testing, it was only punished, which is a later topic of discussion). If we keep the number of goals constant and grow the state size, the possible rewards become more and more sparse. Sparse rewards are a hard problem in reinforcement learning, and credit assignment over long delays leads to slow or divergent learning. Generally, RL fails when applied to problems where rewards are too time delayed. In DeepMinds Atari DQN evaluation, the games where the model failed were those with delayed rewards. We see exactly why this is a problem in table 3.1. With only 3 goals, there is a large number of possible states in each episode. RL problems are often classified into sparse and non-sparse rewards. This environment allows linear scaling towards increasingly sparse rewards, allowing one to study the point at which this becomes a problem. However, in reducing the number of rewards, the number of unique states also decreases. It is evident when calculated, that as we scale this problem, why we might need neural networks over tabular approaches.

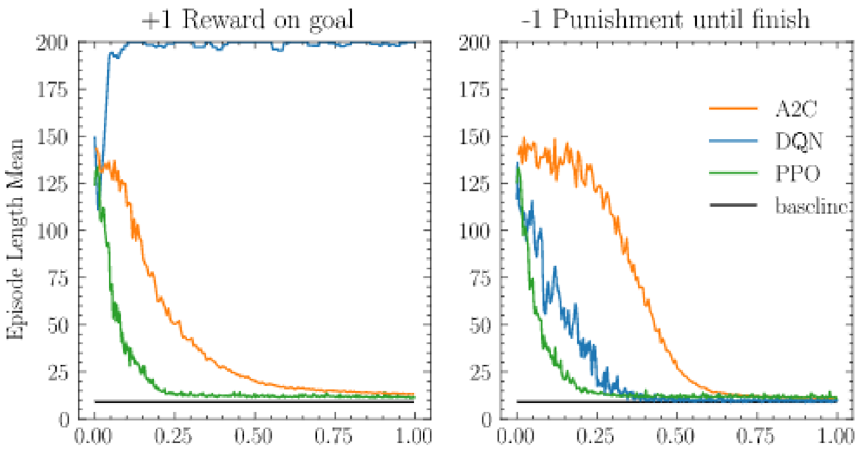

Figure 3.4: Rewarding on goal vs rewarding on maze completion in a 5x5 maze with 3 goals over 30 minutes training time

As mentioned, reward functions play a large part in the training. Here, the maze environment gives us a prime example of how a reward function affects training efficiency and convergence. Sutton and Barto [26] describe that an agent should be rewarded by the overall goal one wants it to achieve, rather than how to achieve it. Each reward function here can fit both categories, however intermediate rewards are shaping how we want the task to be achieved. However, this leads to very sparse rewards in the maze environment when increasing complexity. In just a 6x6 maze, with 3 goals, the agent must learn to act on hundreds of thousands of possible state combinations, with a broad set of possible goal states. We see in 3.4 that giving the agent intermediate rewards actually results in slightly more efficient learning for the policy gradient methods, whereas DQN completely diverges. This is due to an issue outlined later in the discussion, which relates to the non-stationary targets of each goal.

Maze Curriculum Learning

The maze curriculum learning implementation occurs at a fixed n time steps. Each n steps, the amount of possible positions the goals can take on increases by $|G_{positions}| + |G_{neighbours}|$. At $t = 0$, $|G_{positions}$ is only the current, fixed goal positions, and at each curriculum update, this value increases to include $G_{positions}$ neighbours. This process continues until all positions are possible goal positions. The idea behind this is that the agent learns to complete a simpler task of capturing relatively fixed goals. Once this is achieved, the goals spread out slightly more, and so on. This allows for the transition from the fixed problem, with a much smaller state space, to the full problem with a larger state space.

3.2.2 Maze Chaser and Evader

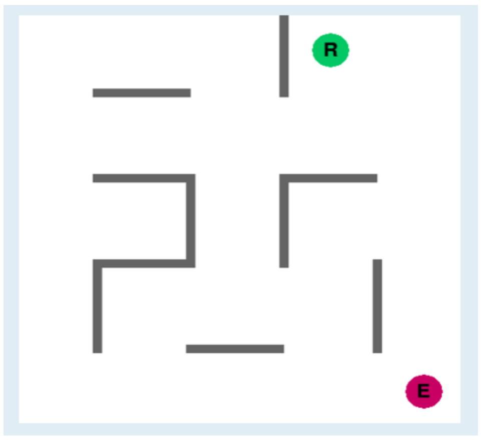

Figure 3.5: Chaser and evader environment in a 5x5 maze

Both the maze chaser and maze evader are similar environments where an AI controls the opposing, enemy agent. Each environment shares the same implementation only differing in settings.

Maze Chaser

The Chaser environments goal is to capture a moving goal, the evader. The evading search agent is a distance maximising evader which regularly selects the point which maximises its distance to the chaser when the chaser enters a certain radius. The agent is rewarded when it receives the goal. The evader has some probability ϵ of taking a random action. Curriculum learning is implemented such that the evader uses an ϵ-greedy strategy. It was decided to have the evader and chaser AI as imperfect, otherwise, the agents would be attempting an impossible task. A perfect evader in a looped maze could simply loop around and never get caught, causing the step limit to reach, meaning the RL controlled Chaser agent would never learn. The optimal policy implemented in the maze Chaser is A*. Although this may not always be the optimal policy in certain environments, it is assumed close enough here, and with the sub-optimal performance of the evader, this makes sense. A* against an A* evader can get stuck in certain situations. An obvious example might be in a 2x2 grid; each agent moving North-South repeatedly after observing the opponent.

Maze Evader

The Evader environments goal is to evade a chaser for as long as possible. The chasing agent is an A* chaser. The agent is rewarded +1 at each step. Each maze is designed such that any position in the maze is not a dead end. From this, an evader can ”loop” in an environment and continue along a path and never be caught. An optimal evasion policy in any maze lacking dead ends is to continue along any loop until the maximum episode step limit.

Chaser Evader Curriculum Learning

Curriculum learning is implemented such that the AI uses a changing ϵ parameter. As in RL methods, ϵ tells the agent whether to stay on its respective A* policy or to take a random action. The idea behind this method is such that the agent is easier to catch, or evade when it is taking solely random actions. This value decreases until it is of fixed value, as previously mentioned.

3.2.3 Other environments

Other environments are included in the API; however, they were not used in results. These environments include a multi-agent chaser evader: competitive maze and a competitive goal environment. Competitive environments required different strategies of multi-agent training and lacked strong baselines.

3.3 Baseline Algorithms

3.3.1 Reacher Baseline - Jacobian Inverse Kinematics

The baseline of the reacher is based on the Jacobian Inverse Kinematics method [19], which solves the inverse kinematics problem of finding the joint configurations given the start and end positions of the effector. This method requires the positions and angles of each arm. The RL state is given only the positions. The goal position is computed using the Jacobian Inverse Kinematics method. Forward kinematics calculates the effector's position given the positions of each joint and the respective arm lengths. Inverse kinematics calculates the positions of each joint, given the desired effectors position. The Jacobian method computes the partial derivatives of each joints positions relative to the previous arm, and therefore the fixed position, with respect to the goal coordinate.

The Inverse Kinematics baseline is, however, not the perfect optimal, but a very close approximation, due to the method calculating the angles of the optimal solution. The implemented method then uses the difference between current and goal angles and steps with the maximum possible step. The difference is divided. As some angles have much larger distances to move, the prior angles in the arms move in tiny increments. However, it was only on single jointed examples that the Inverse Kinematics baseline outperformed - and only slightly.

Algorithm 1: Algorithm 1: Jacobian Inverse Kinematics baseline

```
Result: Move effector to a goal using Jacobian IV
while effector not at goal do
    $targetVec = (effector_{pos} - target_{pos}) * max_step$;
    calculate jacobian $J$ given joint positions;
    calculate inverse $J^{-1}$;
    $\theta_J = J^{-1} \cdot targetVec$;
    $action = \theta_J$ clipped to step size;
    clip action by step size and apply to angles;
end
```

3.3.2 Maze Baseline - TSP

The travelling salesman problem is especially interesting in the field of reinforcement learning as humans and animals approximate a solution to TSP limited goals with often near-optimal performance when less than 20 nodes are considered [13]. As RL is often postured as an equivalent to the process by which humans and animals learn, additionally so with approximating neural networks, a learnt approximate solution becomes especially exciting. The analytic solution provided is the brute force approach and guaranteed global optima. A discrete genetic optimisation algorithm approximate solution approach is also included as a local optima baseline, however ideally this is not used.

The brute force TSP solver calculates the shortest path between the start node through all goals by computing all paths. It differs from typical TSP slightly such that there is no requirement of a round-trip to the start. That is, the episode ends when the last goal is captured. Therefore, the solver calculates the TSP path with the direction it must follow. The distance of each node is calculated using A*.

Algorithm 2: Algorithm 2: TSP Maze solver

```
Result: Calculate one way TSP starting at agent position
given agent position and goal positions and all A* distances;
if maze size < optim thresh then
    for Result:
    each permutation of agent and goals do
        calculate trip length starting at agent through all goals;
    end
    take minimum trip length, without return to agent;
else
    Use discrete genetic optimisation to solve TSP;
end
```

3.3.3 Evader A* Baseline

The evader baseline is a distance maximising A* evader. The agent takes all locations around it at a fixed diameter, then from these chooses the position which maximises its A* distance to the chaser. However, as each maze provided as part of the environment has no dead ends, one can safely assume a global optima will always have the same return as the max episode length * reward. While this is often optimal, the agent does get stuck in corners when the chaser approaches from certain directions. Therefore, while a baseline agent was implemented, the max episode steps of 200 were used as the baseline value throughout testing. The baseline agent was used as the AI agent in the chaser environment.

Algorithm 3: Algorithm 3: A* based evader

```
Result: Evade opposing agent as long as possible
initialization;
while not caught do
    for each position within radius r do
        calculate A* distance to chaser;
    end
    move towards position with maximum distance to chaser;
end
```

3.3.4 Chaser A* Baseline

The baseline for the chaser environment is simply A*. Due to the simplicity of the evader, this can safely be assumed to be optimal. However, this is not to say that this algorithm is optimal in any predator-prey type environment.

Chapter 4

Experiments

A large number of experiments were run on the environments. Each environment has a number of parameters and can be scaled in complexity. To list all experimentation would be outside the scope of this report. The parameters used and more results can be found in the Appendix.

This report will deal with the initial environment testing, including baselines for each environment and their sizes dealt with in this report. For each environment, it will be displayed how changing the state size, and therefore the environment complexity changes algorithmic convergence and variance in learning towards the baseline. A number of parameters are selected that show interesting results and are presented for each environment. Parameter selection will then be performed to attempt to diminish the gap between algorithm convergence and the baseline.

Testing was performed using the Ray, RL-Lib framework RLLib [10] for benchmarking each environment. RLLib allows for the easy implementation and tuning of all of the outlined algorithms along with many popular adaptations. Tune [11] was used to find suitable hyperparameters at each setting. In some cases, algorithms were implemented directly from the papers using PyTorch to investigate parameter options or non-working features of RLLib. Due to the large range of hyperparameters, only a select number of environment setting combinations and algorithms tested and some parameters were left as their default. Finally, a naive version of curriculum learning was implemented for each environment.

4.1 Methodology and Hypothesis

The hypothesis of this Thesis is that most deep reinforcement learning algorithms approximate the solution and converge to some local optima. As the S, the total number of unique states increases, the gap between the converged local optima and the global optima baseline will increase. Each algorithm will converge to some local optima due to the relative simplicity of the MDP. The loss in performance is analogous to the increased computation time in solving more difficult planned baseline problems. All algorithms mentioned in the Background will be tested in each environment relevant to their action space. Popular adaptations are often used when offering a performance boost, which can be found in the RL-Lib documentation. The results presented against their baselines use the parameters after tuning. Finally, curriculum learning will be applied in an attempt to increase the convergence towards the global optima or baseline.

Each algorithm was trained over a fixed-length time of 30 minutes. Time was chosen as the independent variable, rather than training steps as to compare on-policy and off-policy algorithms fairly. Off-policy algorithms such as DQN and DDPG use an experience replay and train in batches, whereas on-policy algorithms such as PPO or A2C will use many more sampled steps to train. Consequently, comparing the training steps draws a false equivalence between the two. While training time was the chosen and best possible metric to measure against, different libraries and implementations were in some cases used, having different levels of training efficiency. In some graphs, certain algorithms are excluded; this is either due to them being similar and presenting poor results compared with their counterpart. This was a resource tradeoff made, i.e. including more PPO results over A2C, due to the similar methods, but with PPO showing much better performance. Each trial was run on a dedicated quad-core i5 6500 with 8GB RAM. Where possible, distributed training was performed in RLLib.

Before results were collected, the RL algorithms had to have some convergent properties when applied to the environments and were tested extensively throughout development. Performance increases were evident in both the graphs and when watching loaded models. The following experiments test the algorithms previously mentioned in each environment. Tabular approaches were also tested in the grid environments to ensure they worked as expected. Q-Learning [31] was used with a modified state space in the grid environments. With a working environment, the Deep Reinforcement Learning algorithms were then applied. Note that the reacher is omitted from this as it is a continuous problem and cannot be feasibly be represented in a table, and a Deep RL approach was applied directly.

The baselines for each algorithm will be presented, showing the convergence of each algorithm towards its global optima. For each algorithm and environment, points of interest will be discussed, such as interesting parameter options, divergence and any reasoning behind them. Curriculum learning will be applied in the environments to measure its effect on optimality and convergence. Finally, hyperparameter analysis and tuning will be performed using Tune [11]. For each experiment, hyperparameters can be found in the Appendix. If no parameters are listed, then their RLLib defaults are used, which are a known strong performing set of parameters.

4.2 Parameter Exploration

Certain algorithms are considered for parameter exploration, where interesting results arose. The Tune [11] framework was used for grid based parameter exploration. Due to a lack of resources, not all parameters were tested. Furthermore, in an ideal situation, all parameters would be tested with all different environment combinations over a large timeframe. This means running a full grid search at each parameter level. However, as environmental and algorithmic parameters are added to the grid search, the training time grows exponentially in $O(2^n)$ with the number of parameter options. Therefore, for these results, the parameters were run on the 5x5 grid, or, on the 2-jointed reacher. From here, these parameters were used for baseline experimentation. A full list of parameters chosen for each algorithm is in the index.

4.3 Baseline Comparisons

Each algorithm is tested against the baseline in a scaling environment. The baseline was calculated with the same environmental settings, averaged over 1000 episodes. Each baseline is implemented as described in the Background.

4.3.1 N-Joint Reacher baseline

Figure 4.1 shows the performance of the N-joint reacher with 1,2 4 and 8 joints using PPO, A2C, TD3 and DDPG using Rllib [10] implementations. Both DDPG and TD3 use OU (Ornstein Uhlenbeck) noise for exploration, as mentioned in the Background, whereas PPO is stochastic and doesn't require an explicit exploration technique. These algorithms are selected as they are known to perform well in continuous action space environments. When the number of joints is 1, we see the convergence to the optimal baseline with PPO and TD3, and its decay increasing the reacher's number of joints. DDPG fails to converge. It is known to be less sample efficient; however, it looks as though it would converge given more time.

With all 3 algorithms, the rate at which they converge to some minima is slowed. Additionally, the learning stability is also affected, the more complex state space causes more variance in the average episode length. This is especially prominent in PPO. The variance visibly increases, resulting in less steady learning. Note that DDPG and TD3 have far less variance due to being off policy and training over large batches of sampled data. This means that less actual time steps are recorded as for each time step executed in the environment, a number of batches are trained on.

With more than 1 free joint, PPO performs the best when compared to the baseline, while other algorithms show poor results, and don't necessarily converge at all. We see the baseline is only

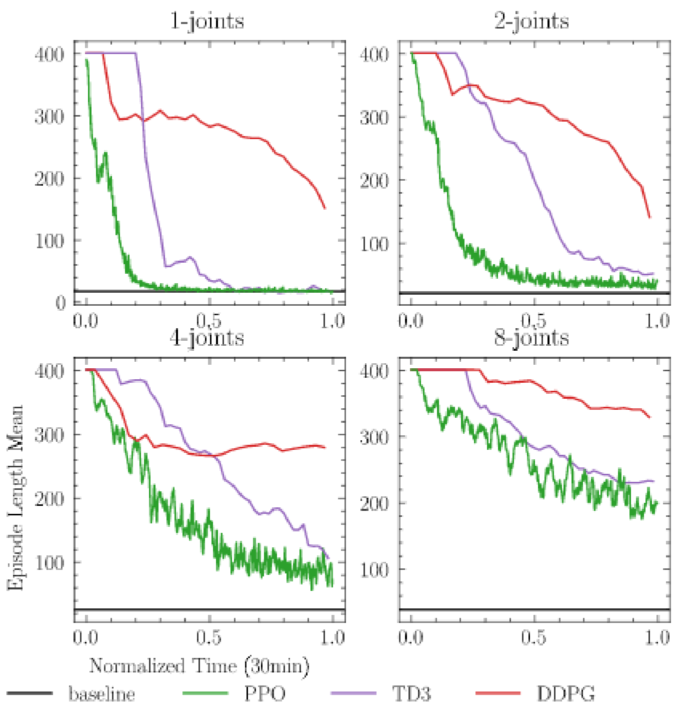

Figure 4.1: All algorithms against the baseline with increasing number of joints

met once, with the smallest state size of a single joint by PPO. The convergence seen in 2 and more joints is therefore to a local Minima and not the global optimal policy. TD3 performs well, and we see that the improvements mentioned in the Background result in better convergence than DDPG.

In this specific MDP, the state size increase has cumulative negative effects on the complexity of the problem. Each additional joint must adjust its movements for the joints prior to it. As the state space is continuous, it cannot be precisely measured. However, with each joint added, it adds a degree of freedom. This not only creates a larger action space, but it adds a joint for which all following joints must account for with their movements. Therefore, adding joints adds significant complexity to the environment. This is reflected in these results. The explosion in state complexity can be demonstrated to discretise step sizes into 1-degree angles. For no free joints, we would have 360 possible states (excluding the infinite goal positions). For a single joint added, we get $360 * 360$ states until 360^n states for n free joints.

In 4.1 DDPG, a top-performing continuous action space algorithm diverges quickly in the first reacher however shows strong results in larger cases. Despite parameter tuning in the Rllib library, DDPG failed to perform near optimally. PPO and TD3 have converging results; however, they are still well off the baseline within the training time limit, which is a surprising result.

It was found that DDPG performs well with a non-standard exploration approach. As mentioned, the standard approach uses OU noise as in the initial paper, however, it was found and has been shown [29] that mean-zero Gaussian noise also works well. RLLib didn't include this as working functionality. Therefore, a separate implementation using Gaussian noise is provided that shows promising results in converging towards the baseline. It was found that a separate implementation, with lower tau values, causes DDPG to converge better to the baseline and is shown in 4.6. Keeping hyperparameters equal, the direct PyTorch DDPG implementation showed better results than RLLib's. It is unknown whether it is implementation errors or training time (of which Rllib requires more) which caused the significant differences in results between implementations. This requires further investigation, however debugging the RLLib implementation is out of the scope of this report. The improved results of DDPG can be seen in Figure 4.6, which also used a 30-minute training window.

A recurring theme in new frameworks based on new technologies is that hyper-parameters are not always working or applied as expected. Certain algorithms were found to perform better when implemented directly. Because these algorithms are complex, it was assumed the libraries contained small implementation differences or bugs. This was evident in Rllibs faulty Gaussian noise for DDPG and TD3. Due to these algorithms being modern, and the scope of these libraries, these bugs occurred frequently. In these cases, a second framework was used, or the algorithm was implemented directly from the paper using PyTorch.

4.3.2 Maze Baseline

The maze environment can scale in two ways to increase state space: the size of the maze can increase, or the number of goals in the maze can increase. Both of these increase the size of the state space as in formula 3.2. Here, both scaling approaches will be used to test the algorithms against the TSP baseline. The baseline value was averaged over 1000 trials of the respective maze size using the TSP solver.

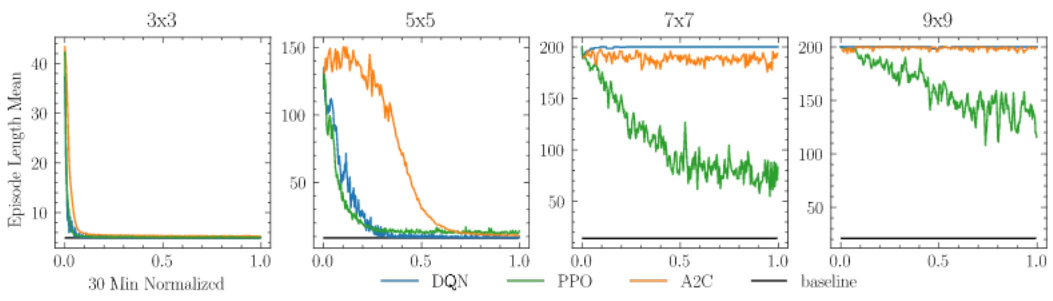

Figure 4.2: Baseline Maze (TSP), 4 different sized environments with 3 goals. Stacked encoding. Separate reward for DQN and PPO

We see in figure 4.2 that increasing the maze sizes causes divergence from the baseline. Increasing the maze size also increases the variance of learning. DQN and A2C both failed to learn in the larger maze settings, completely diverging from the baseline. This issue is likely to the TD error target used in training the DQN and the critic network in A2C. Interestingly, PPO still learns as if it will converge to some local minima at larger maze sizes. However, it shows that the variance of its learning significantly increases.

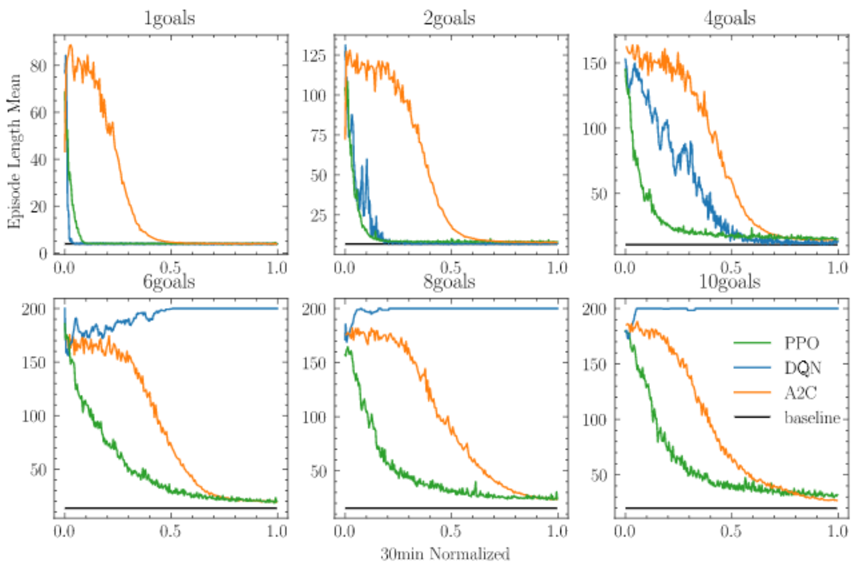

Figure 4.3: Baseline Maze evaluated with changing number of goals for PPO and DQN. Stacked encoding. 5x5 maze. Separate reward for DQN and PPO

Interestingly, A2C outperforms PPO in the final experiment, surpassing the converged policy. However, A2C is much less sample efficient and takes much longer to train. Increasing from 8 to 10 goals actually improves the rate of convergence for A2C when compared to PPO. This is likely due to the fact that the rewards become less sparse as the maze size decreases, despite the number of unique states increasing. This may suggest A2C generalises better in some settings when the number of unique states is considerable. This can be contrasted with the finding that A2C takes longer to converge in environments with fewer goals than DQN and PPO.

It is also worth noting the decay in the convergence of DQN, such that at smaller states, it learns both faster and towards a better optima than both A2C and PPO. Yet when the state space increases, it becomes the first to diverge.

4.3.3 Chaser and Evader Baselines

In the chaser and evader environments, only the chaser reaches the baseline. It is known that these problems are significantly easier than the maze problem due to their limited state space. We can calculate this state-space size by setting the goal value to 1 with formula 3.2. The chaser environment also converges best with DQN and PPO. DQN works well on the chaser environment due to the use of a single reward on completion, meaning no issues with overlapping states and TD-error occur. A2C, however, struggles to converge, which is interesting and an area for further research. The chaser is able to approximate the A* baseline chaser well.

The evader environment struggles to meet the optimal baseline policy of the max episode length. What's interesting is that A2C struggles to converge once again while increasing state space size, whereas DQN and PPO seem relatively unaffected by the increase. DQN converges quicker to a local optima in the 6x6 maze whereas A2C converges quicker in the smaller 3x3 maze. This is likely due to the sample efficiency of DQN given the training time.

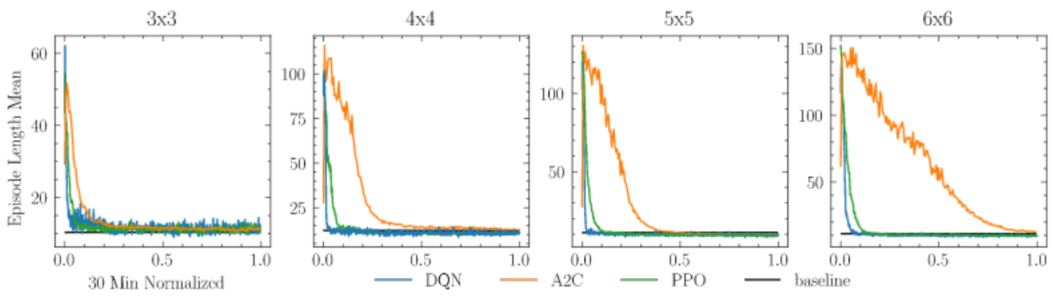

Figure 4.4: Chaser against baseline, different maze sizes

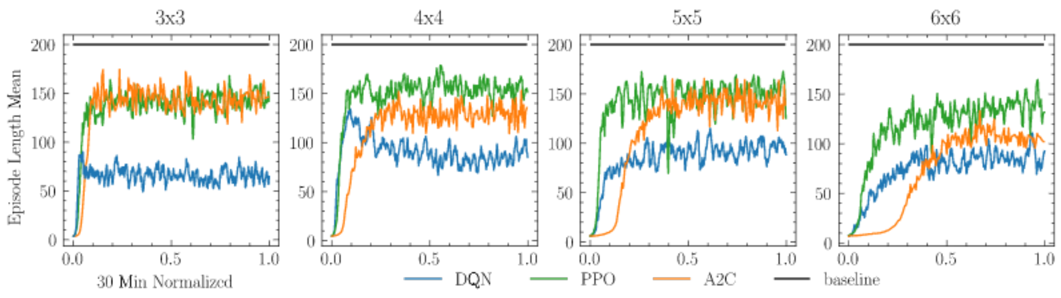

Figure 4.5: Evader against baseline, different maze sizes

4.4 Highlighted Results

Certain results were found to be particularly interesting and reasons as to why this may be the case are given. These results are the areas which would require further research and highlight why such environments are useful.

4.4.1 DDPG Exploration and Critic Issues

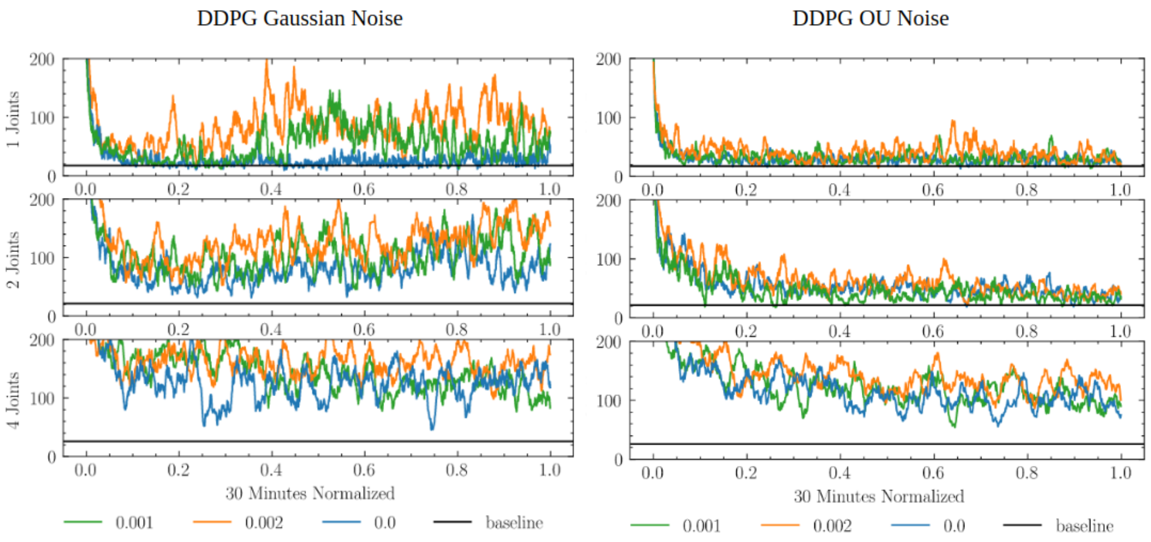

Figure 4.6: DDPG with OU and Gaussian noise, changing tau parameter, trained for 30 minutes

Figure 4.1 does not capture the full utility of the DDPG. Out of the box solutions provided by RL frameworks, while useful, were sometimes unstable and difficult to debug. Exploration in DDPG is generally done using Ornstein-Uhlenbeck noise (OU). However, it's apparent in 4.1 that DDPG fails to converge to a baseline. Here, a version of DDPG was implemented from the paper, and a range of parameters were tested. The exploration was tested with both OU and Gaussian noise. Interestingly, Gaussian noise performed better than OU noise with 1 free joint, which goes against the norm of DDPG papers and implementations. The Gaussian variance was set to anneal from 1 to a fixed 0.1 throughout training, as is common in many exploration techniques. The exact range of variance values did not seem to have a large effect on convergence. Interestingly, the tau parameter was found to have an increasingly negative effect on the convergence of the algorithm. The τ parameter is used to update the target networks as in 2.21

$$\theta^{'} \leftarrow \tau\theta + (1-\tau)\theta^{'} \tag{4.1}$$

Both the target actor and critic networks are updated in this way. As in the pseudocode, setting τ to 0 results in no meaningful change to the target critic, or value network. The target networks are used only in the calculation of the DDPG equivalent of TD error, with the target

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{4.2}$$

where $\mu'(s_{i+1}|\theta^{\mu'})$ is the actor target network and Q' is the critic target network. The important idea here that both target networks are not updated with tau set to 0. This is a significant result, suggesting that τ is an unnecessary parameter in some problem domains. Giving each state a value acts as a heuristic to weight the learning of the policy function. If the heuristic function is bad, or arbitrary, the algorithm may learn, or it may act as an extra form of exploration when calculating actions. We can think of this as a poor heuristic still allowing a search algorithm to work, just not well. With this knowledge, the purpose of critic networks is questioned in deterministic policy gradient methods and specifically DDPG. This is an area for further research and a significant question that this paper highlights.

4.4.2 An issue with TD error

The way TD error is calculated, the state space representation and reward functions result in an issue with the maze problem when using DQN, and other value-based approaches. We can, therefore, assume it also affects the critic state estimation in policy gradient methods. To illustrate this, we can consider the simple example of a 2 goal maze, with g_1 and g_2 using a tabular Q-learning algorithm as in [31]. The maze uses a +1 reward on the goals with a -0.1 penalty per time step. The positions of the goals are randomised between episodes. Our state space encoding can be any discussed in the Background. Consider the Episode Ep_2 where the agent moves from S_1 and captures the first goal and sits at the state it has just captured S_2, at this state, it is rewarded for obtaining the goal. Now, a second episode Ep_1 where the agent has already captured g_1, which was elsewhere in the maze and is on its way to g_2. In this situation, it passes the exact state S_1 it was in in the first situation but receives no reward at it. Herein lies an issue: the same state can be both rewarded and not. This reward is then propagated to prior states, in the non-reward episodes, due to the recursive nature of Bellman's Equation [1], in both the case where there is a goal at S_1 and isn't. This is not necessarily a problem, as in the tabular case, the agent will still learn paths which lead it to a goal, just taking possible detours to prior episodes goals due to a very overestimated value function. If the start and goal positions are uniformly distributed, then we can assume this happens about equally among all states, with some middle paths likely more common than others. For a simple problem, like solving a fixed goal maze, this would not be an issue, however, due to the randomisation and very sparse rewards when the maze gets large, this becomes problematic, and we can see this in the results. This is especially evident in DQN, which is only value-based and it diverges on larger mazes. This was addressed in the stacked and simple state encoding by adding a "just captured" value onto the vector. However, this likely doesn't fully address the problem as the neural network only approximates the state space and still sees the next state with the just captured value as very similar.

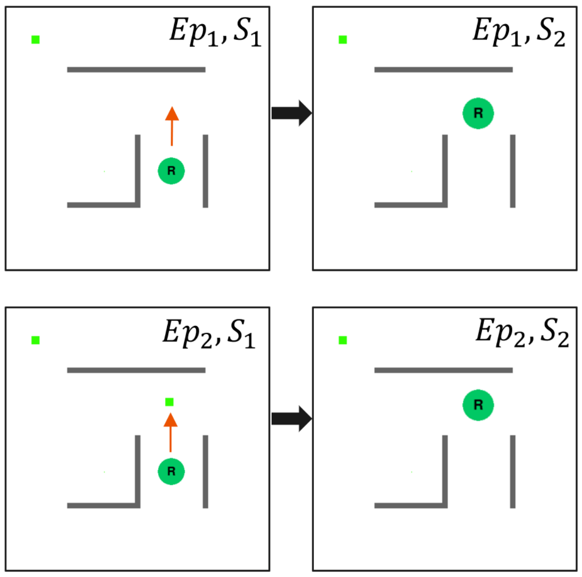

Figure 4.7: Illustration of equivalent states, leading to an issue with the TD error calculation

With policy gradient approaches, this is not an issue, and we actually see a speed up in the efficiency of learning when using +1 reward on goals as is shown in 3.4. Intermediate reward functions are often noted as poor practice. However, this is only the case with value-based approaches which use the TD error calculation. Such functions speed up learning efficiency and make the rewards less sparse. Policy-based approaches do not completely rely on the recursive relationship of TD-error; instead, optimising the best actions directly. However, this issue exists within the critic network or advantage function in the case of PPO. This is a significant shortcoming of highly randomised grid-based environments when using value-based approaches. We either need to engineer the state encoding so that it differentiates between rewarded and non rewarded states or simply use policy-based approaches. In more complex environments, this may not be so simply recognised. To draw a parallel to other literature, using only the pixel output with a CNN as in [17], would still cause the same issue as the two states would be represented as the exact same pixels. Therefore, algorithms such as DQN fail and fail strongly with intermediate rewards for this reason.

4.4.3 Curriculum Learning

Curriculum learning, as implemented in the environments, did not show any speedup in training and did not create better convergence. In both the chaser and evader environment, the algorithms converged quicker and closer to optimal without any random enemy annealing (the enemy agent gradually improved in performance throughout training at fixed intervals). Several rates of change were tested, however, none performed even close to the control without curriculum learning. This was a surprising result as it was predicted that scaling the environment towards the more difficult problem would increase the convergence or rate of training. It is also interesting to note how much better the learnt method performs than A* as previously mentioned.

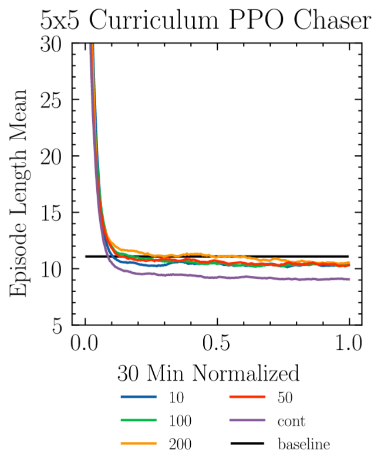

Figure 4.8: PPO curriculum learning in the chaser environment, different annealing levels against the control of no curriculum learning. Annealing levels are the number of episodes before the enemy ϵ parameter was multiplied by 0.99

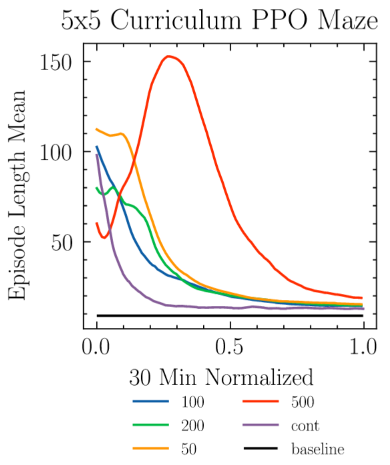

Figure 4.9: PPO curriculum learning in the maze environment. Different annealing levels against the control of no curriculum learning

In the maze, 4.9, gradually increasing the possible goal positions caused the rate of learning to jump at each increase. Because the states are discrete, the number of states jumps largely with each set of new possible states added. This is like giving the algorithm a new problem mid training. It is interesting that the algorithms not only perform better without assistance but are actually hindered by scaling the difficulty of the problem. At 30 minutes, PPO often completed over 1.5m environment steps in the 5x5 maze. We can easily conclude that, with a maximum episode length of 200, that in each annealing case that all positions were possible goal positions (This is likely where we see the spike at the 500 anneal). From this, we can conclude that this sort of curriculum learning did not work.

4.5 Discussion of results

The results have produced a number of interesting insights into reinforcement learning and into the effect state design has on individual algorithms. The results have shown how policies diverge from their optimal baseline as we increase the state complexity. Secondly, these environments have proven to be hard problems for certain Deep RL algorithms. Throughout the literature, we see reinforcement learning excelling in complex domains. Some examples of these complex domains are: humanoid robots learning to run, or agents learning to play Atari games. These domains, however, may have a simple underlying structure for the agent to solve.

The failure of naive curriculum learning is indicative of the fact that an algorithm performs best when given the whole state space for the majority of training. The naive approach of curriculum learning here goes somewhat against intuition. The harder task, with the larger state space, is best left as-is from the beginning of training.

4.5.1 Limitations of DQN

Regarding Deepmind DQN [17], it is seen in this Thesis how DQN fails with a large and randomised state space in the case of the maze environment. We can draw parallels of the pixel-only approach of DQN and the full matrix no-wall encoding tested (a 2D matrix of all grid occupancies). Specific problems, when passed through CNN's or given precise state encoding, may actually have a small unique state space, with few goal states. Alternatively, when many states are very similar and internally represented the same in the deep convoluted network. Although the maze problem is simple to a human, DQN diverges at a map size of 7, with 3 goals, or 900,000 unique states. It is likely that at this point, the accumulated overestimation of state values means the agent struggles to achieve the goal under the step count. The further use of approximation, either through the use of CNN's or using a quadrant based encoding, may aid this. Due to the highly randomised setting, and very large state space, we see here why DQN fails. This, coupled with the error of TD learning representing individual states equally, strongly suggests that DQN is not suitable for specific reinforcement learning tasks.

4.5.2 Policy gradients to the rescue

Throughout Deep RL literature, the general focus of study recently has been on policy gradient and actor-critic methods. A conclusion of this Thesis shows why this is the case. Policy gradient methods allow for a robust solution which addresses directly the task of finding the best action given the current state without the direct need of a value function. Even in situations where the critic networks overestimate [7], or fail to be useful at all as shown in DQN, policy gradient methods can rely on the actor-network, with the critic network acting like poor heuristic. In the obtained results, PPO is the top-performing algorithm. In both continuous and discrete action spaces, PPO showed the most promising results at all scales, almost always converging to an optimum. While this is unsurprising, as it is known as a strong performer, it is also one of the simplest methods. It is simple both to understand and to implement in modern deep learning libraries.

4.5.3 On the difficulty of the reacher and maze environments

When observing these results, one may question how seemingly simple environments lead to diverging results. While in contrast, the same algorithms work on complex humanoid robotic tasks [16] or in games such as Go [30]. Often, environments are chosen where the results are impressive. Hard environments in which reinforcement learning struggles to converge are seldom seen. Such results suffer from negative selection pressure during the process of research and publication, random flukes and the like. First and foremost, present-day scientific practice prescribes that in order to study an unknown process Y, one has to design an experiment such that an X, which is known to be related to Y and directly controllable is perturbed. If changes in X resonate with Y in a salient manner in comparison with the Background, then we have learnt something about the relation of X with Y and, by extension of Y itself from the range of behaviours exhibited, without needing to formulate first a theory. To obtain theory that enables us to predict the performance of learning algorithms; we need to look as well into those experiments where the "Y" failed to resonate and find an explanation for it. Often, more emphasis is on the algorithmic approaches and far less on the variable that is the environment, why it is hard or not so. However, that's not to say there aren't difficult tasks where RL excels. It certainly does, yet the methods are often not as simple as taking the algorithm as-is from the paper and applying it to the task.

Both the randomised maze and reacher tasks are difficult tasks for a few reasons. Firstly, their state spaces are large. Having a large state space means the neural network must learn a complicated function mapping. Secondly, by randomising start and end positions, two episodes are rarely the same. Episodes with two similar start and goal states, and hence intermediate states, rarely appear in consecutively. Looking at OpenAI examples such as the LunarLander (landing a 2D rocket with thrusters), CartPole Acrobat-v1 or even control problems such as the HalfCheetah (a 2 legged walking simulation), we notice they all have identical start states, and the goal is similar among episodes. The set of possible goal states would be small in comparison, and an optimal policy may converge to a minimal set of states. For example, the starting state in HalfCheetah is always the same, and the set of goal states (of moving upright and forward) would be relatively small, and continuously reinforced when found. If we were to map out this as a tree, we would see all states coming from a common root to a common goal or small set of goals. In the reacher and maze environments, the approximation must be highly general. Furthermore, we could define our environments using augmented features such that the goals and start states are common, using a higher-level feature representation. An agent could easily train given the A* distance and direction towards the goal as a state. However, we lose the initial goal of reinforcement learning and doing so might reduce the problem to the extent that RL is not necessary.

The very large state space, with random start and goals, is likely the reason behind the critic failing to aid in the DDPG environment. The critic fails to assess the value of a state as it can't create a precise prediction of it. In a problem such as the reacher environment, an approximate state value might cause the end effector to diverge massively from the goal. The top-performing algorithm throughout all experimentation was PPO. While PPO also uses a critic network, it uses a clip function to reduce bias. The strong performance of PPO is indicative of the need to clip our critic or value estimates for the objective function. PPO clips updates whose value estimates are higher than average such that they don't wreak havoc on the current policy. Or if the action was worse than average, remove its likelihood by a small amount. As our actions are unlikely to affect a large proportion of states due to the complexity of both the reacher and maze environments, this clipping is essential in producing stable learning. This is reflected in the performance of PPO in both environments. Conversely, the unrestricted updating of the value network in both DQN and A2C leads to worse performance.

Scaling such hard tasks as shown, indicates which reinforcement learning algorithms perform well with vast state spaces. Modern algorithms such as PPO and TD3 manage to converge to an optima without significant parameter tuning. However older, more naive algorithms such as DQN, despite the recent augmentations fail to train and diverge from an optima. We can classify certain RL tasks converging to common goal states or diverging from common start states as a factor in their difficulty. The rate at which algorithms diverge from a baseline would be quantifiable in the state space. Variables which affect this would be the reward function, and state observation encoding. To be able to apply reinforcement learning algorithms effectively, an understanding of the dynamics of the problem is useful.

Chapter 5

Conclusion

5.1 Further research

This paper has highlighted a number of areas which could be pursued further and with closer examination. First, the issue with the τ parameter and the critic network in DDPG for robotic control is an interesting observation. Ideally, one would test this in a multitude of environments which use similar continuous action spaces. Secondly, the working implementation of Gaussian noise in a standard environment such as Rllib would be useful to control for any differences across implementations.

Secondly, the relationship between TD error and certain state space intermediate rewards is not thoroughly analysed throughout literature. Reward functions which contain bonuses, for collecting, destroying capturing etc throughout an episode are common throughout literature. These types of rewards potentially cause an issue with the TD error calculation. This is not generally obvious when designing environments or figuring out why certain algorithms fail in certain state spaces. While this is true, these also can speed up learning with some policy gradient methods. An interesting line of research would be to find reward functions which suit both cases, or design a protocol for designing policy gradient and value based reward functions.

Other interesting environments could also be implemented, one being an optimal reacher for multiple targets, approximated by both the jacobian for controlling the arm, and a TSP solver for controlling which points to go to first. More emphasis would be put into measuring or approximating the possible number of start states, intermediate states and goal states. With this information, a methodology could be designed such that we know, or can make educated guesses about which problems will have strong performing RL agents.

The parallel between human and reinforcement learning performance in a setting is also interesting. With the parallels drawn between human performance in TSP and RL, the rate of convergance decay would provide interesting insights into the parallels between animal and computer learnt behaviour. Further, measuring the rate of decay and classifying reinforcement learning problems into domains would be a highly useful topic. Factors such as sparsity of rewards, size of the goal state and start state spaces, total number of unique states (approximate) and the similarity between all start and goal states could be measured and the environments classified. By classifying the problems, we would begin to see exactly which algorithms work and why, with less speculation.

Finally, hyperparemeters could be extensively tested on a distributed computer. As this Thesis used limited resources, it's unknown whether certain hyperparameters could work better at different environment scales. Ideally, workloads would be significantly distributed such that running a full set of grid based parameter tuning did would not take multiple days.

5.2 Problems Encountered

A number of problems were encountered throughout this research which meant that the final results were different from what was first planned. Initially the goal was to implement each algorithm from scratch, however due to the number of variations and adaptations of algorithms it was simply too time consuming to do both the environment implementation and algorithms. A number of algorithms were however implemented and are included within the repository but with only their basic paper functionality. Extensions such as prioritized replay were not implemented. The implemented algorithms include : PPO, DQN, DDPG, Vanilla Policy Gradients (VPG/REINFORCE) and Cross entropy. A large amount of time of this thesis was spent learning and understanding these algorithms.

An issue with using external libraries was that sometimes features were buggy or non existant. This delayed development often. It was not always obvious was was the cause of an issue when a parameter was being changed, while not affecting the algorithm in any way. While these libraries are very useful, certain features should be used with caution and often the actual implemented code had to be read to be certain it would work as intended.

A major time draw in implementation and testing was the amount of parameters that could be possibly be tested, and that throughout experimentation, countless sets of results were discarded when better methods were found. Over 200GB of results were generated in total. Each time a new encoding, or reward function was tested, a full set of results was generated. If the reward function or encoding performed better it was then considered the standard. Therefore, the results shown are the work of many iterations of 'failed' results which did not make the cut. The bulk of this thesis was spent testing out reward functions, new environments and their state encoding and how the algorithms performed on these.

5.3 Conclusion

This paper has proposed a set of environments and their baselines against which we can benchmark modern RL algorithms. With the results and findings, this Thesis shows the decay of convergance in increasingly complex RL environments. Deep RL algorithms diverge from their optimal policies as we increase the environments state complexity. However, certain algorithms are shown to maintain their optimal policy approximation while others diverge completely. Policy based approaches with less emphasis on value functions are shown to perform well in complex environments.

Through benchmarking modern Deep Reinforcement Learning algorithms, a number of interesting results were found. The implementation of the environments highlights the necessity of good MDP design, notably in reward shaping and state observation encoding. The implementation of optimal known policies has allowed for the parallel to be drawn between RL and planning or analytic solutions. Through this report, we can see the advantages of Deep Reinforcement Learning and also how it falls short in complex domains. In the situations where it falls short, we gain insights into our problem domains and why these may be difficult problems for Deep RL. Problems that not only have large state spaces, but highly randomised start and goal states. In learning this, a designer of an MDP can make informed design decisions about what will work when training an agent.

Reinforcement learning is an incredibly exciting field in machine learning and is often compared to human or animal learnt behaviour. While this is so, the underlying mechanisms fall short when simply applied to difficult tasks. Complex environments with large unique state spaces often require more consideration than simply applying a model and tuning the hyperparameters. Hyperparameters were shown to be essential and often caused conflicting results with the known literature.

5.3.1 Acknowledgements

I'd like to firstly thank my supervisor Miquel Ramirez for all his help and guidance throughout this Thesis. Secondly, this Thesis has allowed me to draw on knowledge I've gained throughout multiple subjects at the University of Melbourne in both my undergraduate and masters degrees.

This Thesis would not have been possible without the teachings of many lecturers at the university.

Appendix A

Appendix

A.1 Hyperparameters

Below is a list of all hyperparameters used in the reacher and grid environments. The parameters were used at all scales and configurations of the environments. This isn't necessarily the best assumption: that these parameters work at all levels, however it was made due to limited resources and the computational cost of testing everything. Where the parameters are not listed, they are the default parameters as in the RLLib documentation[1]. The parameters were tested in the 2 joint reacher environment and a 5x5 maze environment.

A.1.1 Reacher algorithm Hyperparameters

DDPG

Parameter	**Value**
Batch Size	512
Buffer Size	5000
Tau	0.001
Model Hidden layers	256, 256
Prioritized Replay	True

Table A.1: Tuned DDPG Reacher Hyperparameters.

TD3

Parameter	**Value**
Batch Size	512
Buffer Size	10000
Tau	0.001
Model Hidden layers	256, 256
Prioritized Replay	False

Table A.2: Tuned TD3 Reacher Hyperparameters.

PPO

Parameter	**Value**
Use critic	False
kl target	0.005
clip param	0.3
Model Hidden layers	512, 512

Table A.3: Tuned PPO Reacher Hyperparameters.

[1]These can be found at https://docs.ray.io/en/master/rllib-algorithms.html

A.1.2 Grid Hyperparameters

DQN

Parameter	**Value**
Dueling	True
Prioritized Replay	True
Buffer Size	50000
Model Hidden Size	256

Table A.4: Tuned DQN Grid Hyperparameters.

PPO

Parameter	**Value**
Use critic	True
kl target	0.01
clip param	0.5
Model Hidden layers	256, 256

Table A.5: Tuned PPO Grid Hyperparameters.

A2C

Parameter	**Value**
Use critic	False
Use GAE	False
Model Hidden layers	256, 256

Table A.6: Tuned A2C Grid Hyperparameters.

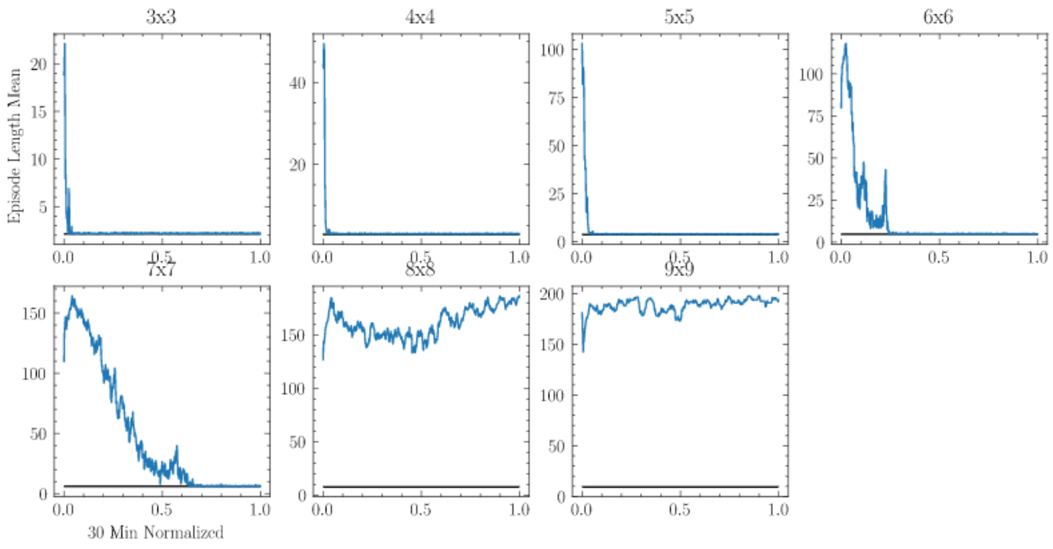

Figure A.1: DQN on maze with only a single goal

Bibliography

[1] Richard Bellman. “Dynamic Programming and Stochastic Control Processes”. In: *Information and Control* 1.3 (1958), pp. 228–239. DOI: 10.1016/S0019-9958(58)80003-0. URL: https://doi.org/10.1016/S0019-9958(58)80003-0.

[2] “Bellman’s Principle of Optimality and its Generalizations”. In: *General Systems Theory: A Mathematical Approach*. Boston, MA: Springer US, 2002, pp. 135–161. ISBN: 978-0-306-46962-6. DOI: 10.1007/0-306-46962-6_7. URL: https://doi.org/10.1007/0-306-46962-6_7.

[3] Yoshua Bengio et al. “Curriculum learning”. In: *Proceedings of the 26th annual international conference on machine learning*. 2009, pp. 41–48.

[4] Dimitri Bertsekas. “Dynamic Programming and Optimal Control”. In: vol. 1. Jan. 1995.

[5] Justin A Boyan and Andrew W Moore. “Generalization in reinforcement learning: Safely approximating the value function”. In: *Advances in neural information processing systems*. 1995, pp. 369–376.

[6] Greg Brockman et al. *OpenAI Gym*. cite arxiv:1606.01540. 2016. URL: http://arxiv.org/abs/1606.01540.

[7] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: *CoRR* abs/1802.09477 (2018). arXiv: 1802.09477. URL: http://arxiv.org/abs/1802.09477.

[8] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. “Reinforcement Learning: A Survey”. In: *CoRR* cs.AI/9605103 (1996). URL: https://arxiv.org/abs/cs/9605103.

[9] Norio Kosaka. “Has it explored enough?” PhD thesis. Sept. 2019. DOI: 10.13140/RG.2.2.11584.89604.

[10] Eric Liang et al. “Ray RLLib: A Composable and Scalable Reinforcement Learning Library”. In: *CoRR* abs/1712.09381 (2017). arXiv: 1712.09381. URL: http://arxiv.org/abs/1712.09381.

[11] Richard Liaw et al. “Tune: A Research Platform for Distributed Model Selection and Training”. In: *arXiv preprint arXiv:1807.05118* (2018).

[12] Timothy P. Lillicrap et al. “Continuous control with deep reinforcement learning.” In: (2016). Ed. by Yoshua Bengio and Yann LeCun. URL: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15.

[13] James N MacGregor and Tom Ormerod. “Human performance on the traveling salesman problem”. In: *Perception & psychophysics* 58.4 (1996), pp. 527–539.

[14] Laëtitia Matignon, Guillaume J. Laurent, and Nadine Le Fort-Piat. “Reward Function and Initial Values: Better Choices for Accelerated Goal-Directed Reinforcement Learning”. In: *ICANN*. 2006.

[15] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[16] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[17] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. URL: `http://arxiv.org/abs/1312.5602`.

[18] George E Monahan. "State of the art—a survey of partially observable Markov decision processes: theory, models, and algorithms". In: *Management science* 28.1 (1982), pp. 1–16.

[19] Adam Morecki and Józef Knapczyk. "The Inverse Kinematics of Manipulators". In: *Basics of Robotics: Theory and Components of Manipulators and Robots*. Ed. by Adam Morecki and Józef Knapczyk. Vienna: Springer Vienna, 1999, pp. 55–90. ISBN: 978-3-7091-2532-8. DOI: `10.1007/978-3-7091-2532-8_4`. URL: `https://doi.org/10.1007/978-3-7091-2532-8_4`.

[20] Matthias Plappert et al. "Parameter Space Noise for Exploration". In: *CoRR* abs/1706.01905 (2017). arXiv: `1706.01905`. URL: `http://arxiv.org/abs/1706.01905`.

[21] Tom Schaul et al. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[22] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[23] John Schulman et al. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015). arXiv: `1502.05477`. URL: `http://arxiv.org/abs/1502.05477`.

[24] Richard S. Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *In Advances in Neural Information Processing Systems 12*. MIT Press, 2000, pp. 1057–1063.

[25] Richard Stuart Sutton. "Temporal Credit Assignment in Reinforcement Learning". AAI8410337. PhD thesis. 1984.

[26] Sutton and 1998 Barto. *Reinforcement Learning, An Introduction*. Cambridge, Massachusetts, London, England: The MIT Press, 1998.

[27] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033.

[28] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Thirtieth AAAI conference on artificial intelligence*. 2016.

[29] Hado Van Hasselt and Marco A Wiering. "Reinforcement learning in continuous action spaces". In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE. 2007, pp. 272–279.

[30] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* 575.7782 (2019), pp. 350–354.

[31] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: `10.1007/BF00992698`. URL: `https://doi.org/10.1007/BF00992698`.

[32] Yuhuai Wu et al. "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation". In: *CoRR* abs/1708.05144 (2017). arXiv: `1708.05144`. URL: `http://arxiv.org/abs/1708.05144`.