Abstract

The past years have been substantial for progress in Reinforcement Learning. Discoveries of how to use deep learning in conjunction with more traditional methods have allowed reinforcement learning to perform in highly complex domains, achieving superhuman results in environments like Go or video games. However, these new methods suffer from extreme sample-inefficiency, high variance and randomness. Deep learning in a supervised setting, on the other hand, has proved itself as a stable approach that can very often outperform more classical models out of the box and, when compared to reinforcement learning case, is much more data-efficient. In this thesis, we explore the possibility of transforming a reinforcement learning setting into a supervised problem with the use of differentiable environment models. We further show that this approach was able to achieve 14% better sample efficiency than the popular DDPG algorithm while providing a more stable model with standard deviation reduced by 8% on the LunarLander environment.

Contents

1 Introduction

Reinforcement learning (RL) has risen in prominence in the past years with the emergence of Deep RL. Deep Q-Learning algorithm introduced by (Mnih et al., 2015) initiated a series of breakthroughs in devising artificial agents that have reached superhuman performance in a plethora of different very complex and high-dimensional environments. First, RL mastered playing Atari games directly from pixels, the same way humans do (Mnih et al., 2015). It was then followed by solving Go (Silver et al., 2016; Silver et al., 2017), the game that contains more possible states than computer created from all atoms in the universe could store, which further led to last year's breakthroughs in StarCraft(Arulkumaran et al., 2019) and Poker(Brown & Sandholm, 2019). Although StarCraft and Poker may not be as complicated as Go, they proved that RL could handle situations when the full state of the game is highly obstructed and involves multiple independent competitors.

Despite reinforcement learning's impressive track record, it still has key issues that stop it from more popular adoption, such as high variance, a large likelihood of divergence, and enormous sample inefficiency (Buesing et al., 2018).

Model-based RL algorithms promise to tackle these problems by using known dynamics of the environment to analyse probable scenarios. The ultimate goal is an agent which can accurately simulate the environment and forgo the execution of expensive exploration in the real world.

A breakthrough in learning the model of the environment in RL setting was proposed by (Oh et al., 2015), where they described novel architectures to learn the transition probabilities function T. This was later extended to support modelling of reward function R (Leibfried et al., 2016), meaning it could learn the full model of the environment.

The idea of using a modelled environment was then used to drastically increase sample efficiency in regard to the state-of-the-art (Kaiser et al., 2019; Kielak, 2019). However, it was later found that correctly tuning the hyperparameters for Rainbow DQN gives a similar performance at a much lower computational complexity (Kielak, 2020; van Hasselt et al., 2019).

The key difference between the real environment and the modelled one that previous studies have not investigated is that the latter is always differentiable. This leads to various possible opportunities, one of which is employing the gradient of the differentiable world model to do supervised learning.

This research aims to investigate and evaluate the use of supervised methods on the modelled environment of the world. Current deep RL is highly sample inefficient, unstable and random, leading to even well-tuned models to produce bad policy in some cases and tuning hyperparameters not well enough may leave a policy that is worse than random (Irpan, 2018). Supervised learning, on the other hand, is stable, small adjustments to the hyperparameter space will not affect performance drastically. To mitigate these issues, Deep RL uses techniques such as experience replay or target network (Mnih et al., 2015), adding unnecessary memory requirements and conceptual complexity when compared with the possibility of supervised learning.

As such the goals of this research will be to improve on the current RL algorithms in sample efficiency or variance, as these are very significant problems for reinforcement learning (Mankowitz et al., 2019).

2 Background

2.1 Reinforcement Learning

At its core, RL is concentrated on how to make decisions and sequential actions in environments so as to maximise some numerical reward which represents a long-term objective (Szepesvari, 2010).

RL agents traverse the environment by using a policy which says how they should perform at a given state. A policy is the agents action selection process, and it can be imagined as a map

from states to actions. A policy can be either deterministic, where for every state there is only one action, or stochastic where every state maps to a probability distribution over possible actions.

A long-term objective in the form of a numerical value could be represented in the form of maximising the Value function $\boldsymbol{V_\pi(s)}$. It is defined as the expected total cumulative reward starting with state s and following policy π. In other words, it estimates the expected value of a given state. A discount factor γ is used that can range between [0,1) and it controls the importance of future rewards versus the immediate ones. The lower γ s, the less crucial future rewards are, and the agent will prioritise immediate rewards. The reward is multiplied by the discount factor to the power of how many steps into the future we are looking at.

$$\boldsymbol{V_\pi(s) = E[R] = E\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right],}$$

Where R denotes the sum of the future discounted rewards.

There are three different types of Reinforcement Learning algorithms:

- Policy-based - directly learn policy mapping $\boldsymbol{\pi : S \rightarrow A}$ E.g Reinforce (Williams, 1992).
- Value-based - learn the state-action value function $\boldsymbol{Q^\pi : S \times A \rightarrow R}$, which gives a reward estimate for a state & action combination, assuming that after that action policy π will be followed . Then it is possible to use it to infer the optimal policy $\boldsymbol{\pi(s) = \arg\max_{a \in A} Q^\pi(s,a)}$. An example would be Q-learning (Watkins & Dayan, 1992).
- Actor-critic - Utilises both policy (actor) and action-state value function (critic), both of them learn from the environment interactions. The actor acts in the environment and is tweaked by the critic towards a higher reward. E.g natural actor-critic (Peters et al., 2005).

Such traditional approaches fare well in discrete environments where, for example, it is possible to store a complete table of Q(s, a) in memory. Nevertheless, for substantial or continuous environments where it is impossible to store all $\langle s, a \rangle$ combinations or even visit all states, these approaches as they are, are not sufficient.

2.2 Deep Reinforcement Learning

For RL to scale to state spaces which are too large to be completely known, function approximators in the form of neural networks can be used for Q. However, for a long time, the combinations of RL and neural networks had failed considerably due to unstable learning.

Firstly, it has problems with unstable targets. We defined $Q^\pi(s, a)$ as the reward performing an action in a given state and afterwards following policy π. When we want to update our function approximator Q^π we use the error between the predicted value and the target(actual) value.

$$\boldsymbol{target = R(s,a) + \gamma max_{a'} Q^\pi(s', a')}$$

$$\boldsymbol{predicted = Q^\pi(s,a)}$$

That means the target Q value depends on itself, meaning we are chasing a non-stationary target which can cause our learning to diverge.

Furthermore, doing network updates in the traditional RL sense - online, means there is a lack of independent and identically distributed variables. As such future rewards/states highly correlate with current $\langle s, a \rangle$, meaning that we may end up updating our network with a sequence of actions in the same trajectory, magnifying the effect beyond our intention.

These problems have been addressed with the DQN algorithm introduced by (Mnih et al., 2013). It allowed to successfully obtain strong performance in a wide, high-dimensional, set of ATARI games. It did so by employing two significant changes:

Replay memory - exploring the environment usually means that the experiences that the agent takes from interacting with the environment are highly correlated. To circumvent this the replay memory keeps all the information from the last n steps. The steps are held in the form of tuples $\langle s, a, r, s' \rangle$. When updating the model, batches of tuples are sampled randomly. This allows for updates to cover a wide range of the state-action space that consequently break the correlation between samples. Furthermore, it gives the possibility to make a greater update of the parameters, while also being an efficient means of parallelisation (François-Lavet et al., 2018).

Target network - The algorithm maintains two neural networks, the online network Q and a target network Q'. The target networks parameters $\theta - k$ are updated every $C \in N$ iterations, by assigning the parameters of the online network $\theta - k = \theta k$. The actions by the agent are chosen normally - by the Q network. However, Q is updated using a modified Bellman equation that employs the target network:

$$\boldsymbol{Q(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a' \mid s') \hat{Q}(s', a')}$$

This helps with stabilising the learning process and avoiding exploding gradients.

Nevertheless, DQN being based on Q-learning means that it chooses the optimal action in a state, with policy $\boldsymbol{\pi = max_a Q^{\pi}(s,a)}$. However, this becomes problematic when the action space is continuous as it becomes very easy to diverge. To solve the continuous action problem, approximators need to be used. A combination of learning an approximator for Q with learning an approximator to π, was proposed (Lillicrap et al., 2015) leading to the deep deterministic policy-gradient(DDPG) actor-critic algorithm. Recently, multiple incremental improvements to DDPG were consolidated to form Distributed Distributional Deep Deterministic Policy Gradient(D4PG) algorithm (Barth-Maron et al., 2018). The improvements included the introduction of a distributional critic, using distributed agent's running on multiple threads to collect experiences, prioritised experience replay, which made the reward of an experience correlate with its chance of being used for training and N-step returns where the critic instead of computing one step ahead incorporates more steps ahead (5 in the paper).

Google Deepmind, when evaluating their control suite, found D4PG to be the best performing agent, with close results to DDPG, which was second (Tassa et al., 2018). D4PG is the current state-of-the-art algorithm for continuous action and state space environments.

3 Related Work

3.1 Model-based Reinforcement Learning

RL can be split into Model-based and Model-Free approaches. The model being some representation of an environment. Essentially, if the RL algorithm can somehow infer the future state of the environment (e.g. after a series of actions), it is model-based. If it only cares about policy and its expected return, without any direct internal reasoning about the external world, it is model-free.

Contemporary approaches in the area of model-based RL focus on variational autoencoders (VAEs) (Kingma & Welling, 2013), recurrent neural networks(RNN) (Giles et al., 1994) and Bayesian

methods. More recently there have been significant approaches to improve RL sample efficiency by taking the traditional ideas of DQN and modifying it so the experiences in the memory contributed to creating a model of the environment either by GANs (Azizzadenesheli et al., 2018; Kielak, 2019), supervised (Kaiser et al., 2019) or unsupervised (Ha & Schmidhuber, 2018) methods. This modelled environment was then used in place of the real environment either interchangeably (Azizzadenesheli et al., 2018; Kielak, 2019) or fully (Kaiser et al., 2019) for the actors training to reduce the number of samples it required from the real environment to converge.

Lastly would be the RL concept of latent imagination. This is the agent using the modelled environment to generate/imagine trajectories in the latent state space (Buesing et al., 2018; Doerr et al., 2018; Schrittwieser et al., 2019). Then using these trajectories for planning in the environment for the optimal reward. Most recently (Hafner et al., 2019) presented Dreamer, an agent that learns long-horizon behaviours from images purely by latent imagination. The papers key contributions are learning long-horizon behaviours by latent imagination and setting the new state-of-art for visual control in data-efficiency, computation time and final performance.

3.2 Reinforcement Learning with supervised learning

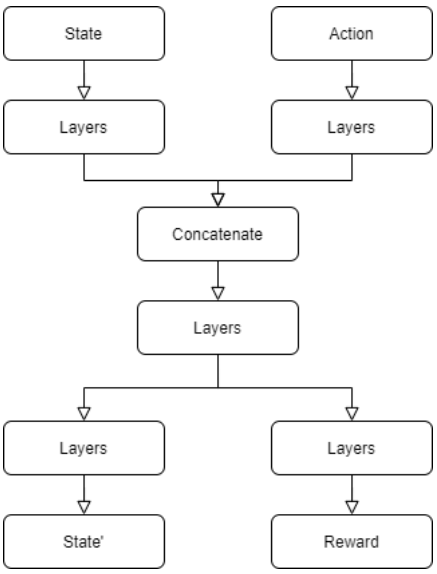
Searching for synergies between RL and SL was for a long time overlooked. This can be explained with ease as both research communities thought they were working on different or even incompatible approaches.

However, there do exist combinations of RL and SL in a variety of fields. The best example would be imitation learning (Kumar, 2020). It is a technique for learning from human demonstration, hence using a supervised approach for a reinforcement learning task. In (Henderson et al., 2008) Reinforcement Learning is used to optimise a measure for dialogue reward, while the Supervised Learning is used to restrict the learned policy to the portions of state spaces for which they have data for learning dialogue management policies from a fixed dataset. (Fathinezhad et al., 2016) uses a combination of Supervised Learning and Reinforcement Learning for driving E-puck robot in an environment with obstacles, with the approach decreasing the learning time and the number of failures.

Nevertheless, none of the previous studies or studies to our knowledge have used SL to train actions to act in a RL environment or tuned a RL agent's weights directly with SL via a modelled environment, which are the two main novel ideas of this thesis.

4 Main

4.1 The model of the environment(env')



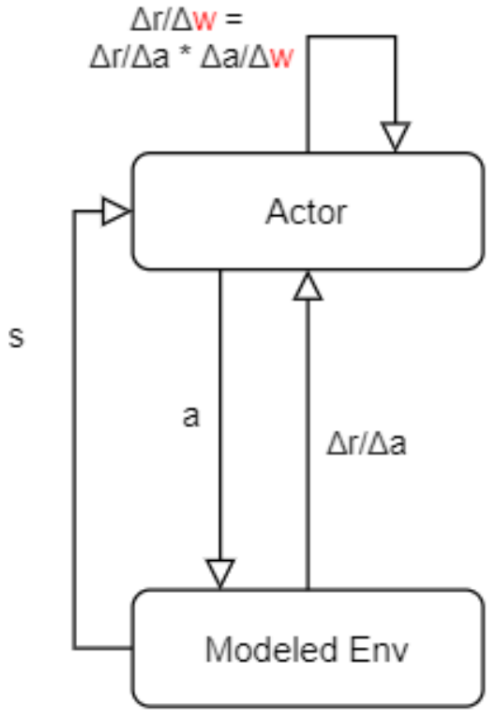
Our environments of choice are part of the OpenAI Gym. They are deterministic environments, meaning that given the same starting state and the same set of actions, the result will be the same. The environment is always in a specific state and given an action, acts in the state, and produces the next state and the immediate reward. Our goal is to build a model that can encapsulate the same functionality.

OpenAI Gym: $\boldsymbol{env.step(a) \rightarrow s', r}$
Our model env': $\boldsymbol{env'(s, a) \rightarrow s', r}$

As such, to maintain the same behaviour, the logical choice is to go for a Multi-Input Multi-Output(MIMO) model.

Having our modelled environment env' we will describe three potential approaches to utilise its differentiability. We will then follow by evaluating the two of the most promising ones to see if they improve sample efficiency and reduce variance.

4.1.1 Aproach 1 - Optimal action for the state - greedy approach



A general RL agent is one that takes in a state and produces an action that it thinks would produce the optimal reward. As such, when optimising the agent's weights, we would like to optimise them in regards to the reward. To do that, we need the gradient of the reward with respect to the agent's weights. Unfortunately, in standard RL setting, since the real environment that produces rewards is not differentiable, it is impossible. However, in our case, this issue can be easily circumvented thanks to the modelled environment env' that was introduced earlier. I.e., since env' is a neural network model, it is fully differentiable.

Optimising as such leads us to a greedy model, that has learned to take actions that yield the highest immediate rewards.

The approach can be defined as:
Objective **:** At each step t, given s_t, $argmax_\theta r_{t+1}$
Definitions **:**

- $r_{t+1} = R(s_t, a_t)$ - reward after t'th step (with state s_t and action a_t)
- $s_{t+1} = R(s_t, a_t)$ - state after t'th step (with state s_t and action a_t)
- $a_t = A(s_t)$ - action at t'th step (with state s_t)
- $S(s, a)$ - function defining reward given state and action (1st output of env')
- $R(s, a)$ - function defining reward given state and action (2nd output of env')
- $\Pi^\theta(s)$ - function defining action given state parametrised by θ
- $\theta = \{\theta_0, .., \theta_m\}$ - set of learnable parameters of $\Pi^\theta(s)$ (weights of the neural network of our agent)
- m - number of parameters defining $\Pi^\theta(s)$

$S(s, a), R(s, a)$ have their own parameters but when training the agent we assume they are trained and fixed.

Method **:**
Because, when optimising our agent, we use gradient descent, to achieve $argmax_\theta r_{t+1}$ we need to calculate $\frac{\partial r_{t+1}}{\partial \theta_i}$ for each $i \in \{0, ..., m\}$, in this case it is very simple:

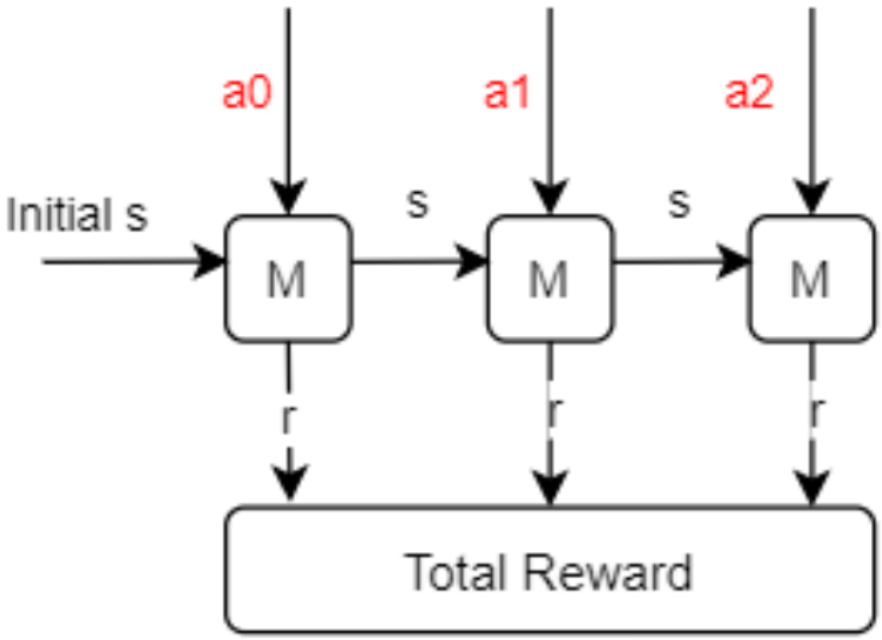
$$\frac{\partial r_{t+1}}{\partial \theta_i} = \frac{\partial r_{t+1}}{\partial a_t} \cdot \frac{\partial a_t}{\partial \theta_i}$$

for every i $\in \{0, ..., m\}$

4.1.2 Approach 2 - Optimal chain of actions without agent

Let us think of what RL algorithms are trying to solve. In general, we want an agent that when put in env acts in such a way that maximises the cumulative reward. The cumulative part is important because, in most cases, it may be needed to take locally sub-optimal actions for the current state, to eventually end up in a position which leads to the highest long-term cumulative reward. Thus, although relatively simple, the previous approach does not achieve what we need.

Before coming up with a final solution to the general problem, let us try to tackle slightly simplified variation. I.e. let us assume we always start in the same initial state in the deterministic environment. Thus, an optimal sequence of actions is always the same. Now instead of learning a general agent's mapping from state to actions that provide an optimal action for every possible state, we need to learn an optimal sequence of actions (independent of states). Thus we can remove the agent from the equation and try to learn this sequence directly. Furthermore, now we do not want to maximise merely a single step reward but the full sum of all rewards in the episode.



The approach can be described as:
Objective **:** At each episode e, given s_0 and N, $argmax_A \sum_{t=1}^{N} r_t$
Definitions **:**

- $r_{t+1} = R^\theta(s_t, a_t)$ - reward after t'th step (with state s_t and action a_t)
- $s_{t+1} = R(s_t, a_t)$ - state after t'th step (with state s_t and action a_t)

- $A = \{a_0, ...a_{N-1}\}$ - set of **learnable** parameters defining actions at each step
- $S(s, a)$ - function defining reward given state and action (1st output of env')
- $R(s, a)$ - function defining reward given state and action (2nd output of env')
- N - number of parameters

$S(s, a), R(s, a)$ have their own parameters but when training the agent we assume they are trained and fixed.

***Method*:**

Because we use gradient descent, to achieve $argmax_A \sum_{t=1}^{N} r_t$ we need to calculate $\frac{\partial \sum_{t=1}^{N} r_t}{\partial a_i}$ for each $i \in \{0, ..., N-1\}$

$$\frac{\partial \sum_{t=1}^{N} r_t}{\partial a_{N-1}} = \frac{\partial r_N}{\partial a_{N-1}}$$

$$\frac{\partial \sum_{t=1}^{N} r_t}{\partial a_{N-2}} = \frac{\partial r_{N-1}}{\partial a_{N-2}} + \frac{\partial r_N}{\partial a_{N-2}} = \frac{\partial r_{N-1}}{\partial a_{N-2}} + \frac{\partial r_N}{\partial s_{N-1}} \cdot \frac{\partial s_{N-1}}{\partial a_{N-2}}$$

$$\frac{\partial \sum_{t=1}^{N} r_t}{\partial a_{N-3}} = \frac{\partial r_{N-2}}{\partial a_{N-3}} + \frac{\partial r_{N-1}}{\partial a_{N-3}} + \frac{\partial r_N}{\partial a_{N-3}} = \frac{\partial r_{N-2}}{\partial a_{N-3}} + \frac{\partial r_{N-1}}{\partial s_{N-2}} \cdot \frac{\partial s_{N-2}}{\partial a_{N-3}} + \frac{\partial r_N}{\partial s_{N-1}} \cdot \frac{\partial s_{N-1}}{\partial s_{N-2}} \cdot \frac{\partial s_{N-2}}{\partial a_{N-3}}$$

.
.
.

$$\frac{\partial \sum_{t=1}^{N} r_t}{\partial a_0} = \frac{\partial r_1}{\partial a_0} + \frac{\partial r_2}{\partial s_1} \cdot \frac{\partial s_1}{\partial a_0} + \frac{\partial r_3}{\partial s_2} \cdot \frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_1}{\partial a_0} + ... + \frac{\partial r_N}{\partial s_{N-1}} \cdot \frac{\partial s_{N-1}}{\partial s_{N-2}} \cdot ... \cdot \frac{\partial s_2}{\partial s_1} \cdot \frac{\partial s_1}{\partial a_0}$$

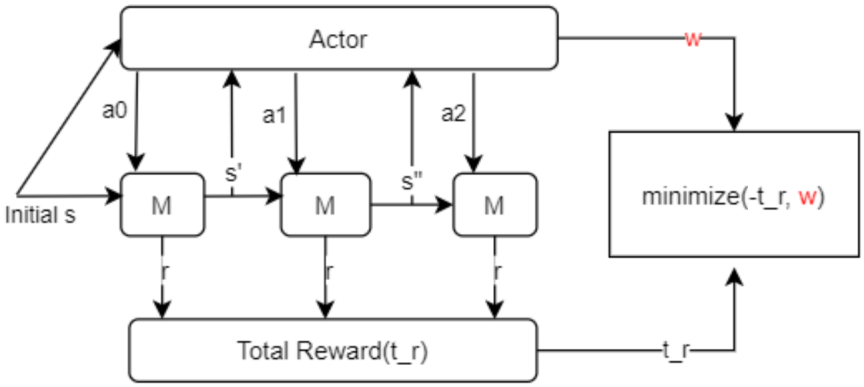
Following this we are able to know the gradient for all actions.

4.1.3 Approach 3 - Augmenting agent with env'

Approach 2 is efficient at finding the optimal actions for the highest cumulative reward but suffers from the fact that it can only work if the starting state is always the same, which is not something we can easily assume.

For our last approach, we propose an architecture that addresses that. Instead of creating an algorithm to compute the optimal actions for each scenario, what if we generalise our novel idea and use it to augment the current state-of-art model.

Instead of randomly exploring the environment we instead build our env' from the experience replay(1) from our agent. We then do learning in the real environment and in env' interchangeably, similarly to (Kaiser et al., 2019; Kielak, 2019). But instead of using env' to generate next states, we like in approach 2 generate a sequence of tensors from env' . However, now the actions are not variables but are the output of our agent, which lets us link the total reward to the agent's weights. After generating the whole chain of tensors, we maximise the accumulated reward in respect to the actor's weights, as such, pushing the actor to perform better in the environment.



The approach can be described as:

Objective **:** At each episode e, given s_0 and N_e, $argmax_\theta \sum_{t=1}^{N_e} r_t$

Definitions **:**

- $r_{t+1} = R(s_t, a_t)$ - reward after t'th step (with state s_t and action a_t)
- $s_{t+1} = R(s_t, a_t)$ - state after t'th step (with state s_t and action a_t)
- $a_t = \Pi(s_t)$ - action at t'th step (with state s_t)
- $S(s, a)$ - function defining reward given state and action (1st output of env')
- $R(s, a)$ - function defining reward given state and action (2nd output of env')
- $\Pi^\theta(s)$ - function defining action given state parametrized by θ
- $\theta = \{\theta_0, .., \theta_m\}$ - set of learnable parameters of $\Pi^\theta(s)$ (weights of the neural network of our agent)
- m - number of parameters defining $\Pi^\theta(s)$
- N_e - number of steps in episode e

$S(s, a), R(s, a)$ have their own parameters but when training the agent we assume they are trained and fixed.

Method **:**

Because we use gradient descent, to achieve $argmax_\theta \sum_{t=1}^{N} r_t$ we need to calculate $\frac{\partial \sum_{t=1}^{N_e} r_t}{\partial \theta_i}$ for each $i \in \{0, ..., m\}$: Because derivative of sum is sum of derivatives, we calculate $\frac{\partial r_t}{\partial \theta_i}$ for each $t \in \{1, ..., N_e\}$ separately by using recursive equations.

$$\frac{\partial r_t}{\partial \theta_i} = \frac{\partial r_t}{\partial s_{t-1}} \cdot \frac{\partial s_{t-1}}{\partial \theta_i} + (\frac{\partial r_t}{\partial a_{t-1}} \cdot \frac{\partial a_{t-1}}{\partial s_{t-1}} \cdot \frac{\partial s_{t-1}}{\partial \theta_i} + \frac{\partial r_t}{\partial a_{t-1}} \cdot \frac{\partial a_{t-1}}{\partial \theta_i})$$

for every $t \in \{1, ..., N_e\}$ Where:

$$\frac{\partial s_t}{\partial \theta_i} = \frac{\partial s_t}{\partial s_{t-1}} \cdot \frac{\partial s_{t-1}}{\partial \theta_i} + (\frac{\partial s_t}{\partial a_{t-1}} \cdot \frac{\partial a_{t-1}}{\partial s_{t-1}} \cdot \frac{\partial s_{t-1}}{\partial \theta_i} + \frac{\partial r_t}{\partial a_{t-1}} \cdot \frac{\partial a_{t-1}}{\partial \theta_i})$$

for every $t \in \{1, ..., N_e\}$ and

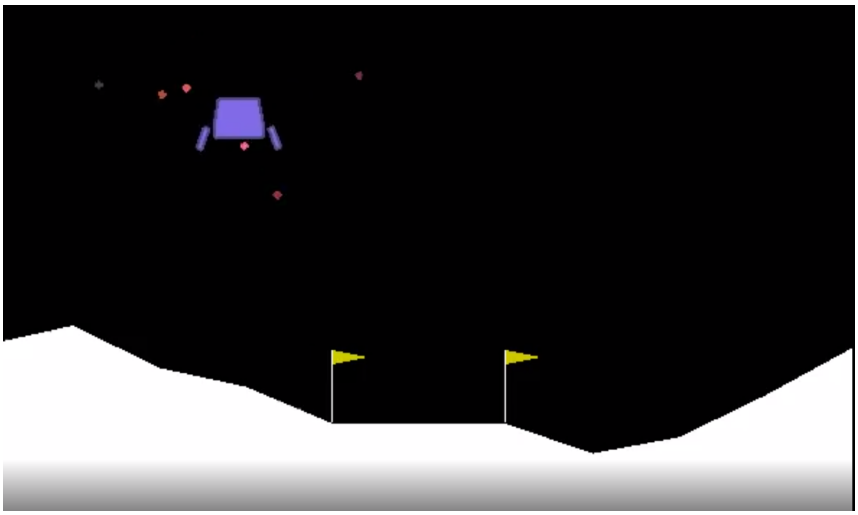
$$\frac{\partial s_0}{\partial \theta_i} = 0$$

The rest of the derivatives ($\frac{\partial s_t}{\partial s_{t-1}}$, $\frac{\partial s_t}{\partial a_{t-1}}$, $\frac{\partial a_{t-1}}{\partial \theta_i}$) are known for every $t \in \{1, ..., N_e\}$ if$R(s, a)$, $S(s, a)$, and $A^\theta(s)$ are differentiable.

5 Experiments

To test the approaches, OpenAI Gym (Brockman et al., 2016) was used, due to it being one of the most popular RL evaluation environments in the research community. In recent years the most widely used RL benchmarks from it were games from the Atari Learning Environment (Bellemare et al., 2013). However, due to the very intense hardware required to converge on these games in a timely manner and given our time constraints, we will be employing a moderately less complicated benchmark from the aforementioned OpenAI Gym, with the plan to continue the work to analyse the performance of approaches on adopted test environments.

5.1 LunarLanderContinuous



LunarLanderContinuous is the continuous action version of the LunarLander environment. The environments goal is to teach the agent's "lunar lander" how to correctly land on the specified spot(0, 0) regardless of the agent's starting position or velocity and avoiding the potential random terrain.

State space: (x, y, v_x, v_y, θ, v_θ, left_leg, right_leg)

(x, y) the agent's position.
(v_x, v_y) horizontal and vertical velocity.
θ agents orientation in space.
v_θ angular velocity.
(left_leg, right_leg) two flags which say whether the agent is touching the ground with left or right foot.

Action space: (fire_main_eninge, fire_side_engines)
Fire_main_engine: controls the main middle engine that propels the agent up in regards to its orientation. Value between -1 and $1, -1..0$ is off, $0..1$ is engine power from 50% to 100%.

Fire_side_engines: controls either the left or right side engine which push in the directions respective to the orientation of the agent. Value between -1 and $1, -1.. - 0.5$ fire left engine, $-0.5..0.5$ off, $0.5..1$ fire right engine.

Reward: Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from the landing pad, it loses reward back. The episode finishes if the lander crashes or comes to rest, receiving additional -100 or $+100$ points. Each leg ground contact is $+10$.

We will be using this environment to test the novel Approaches 2 and 3.

5.2 Env' training

To be able to construct a good env' we first need to get examples from the real one. As such, we employ the concept of memory M. Similar to the replay buffer of the DQN style algorithms. It is an array storing real-environment experiences.

For approach 2, we will be training the env' by randomly sampling the environment.

```
Env' training:
Initialise model env' ;
Initialise empty list M ;
COLLECT_SAMPLES(env, M) #collects samples from the real env with following the π
  of the agent and add them to list M;
env'.TRAIN_SUPERVISED(M) #Train env' on the collected samples to better represent
  the true env;
```

For approach 3, we will train the modelled env from the memory buffer of our agent.

```
Env' training:
Initialise model env';
Initialise model agent;
Initialise empty list agent.M;
while not done do
    agent.TRAIN_RL(env) # agent trains in the real environment for some time and
      collects experience in the memory buffer ;
    env'.TRAIN_SUPERVISED(agent.M) # Train env' on the collected samples to better
      represent the true env ;
    agent.TRAIN_FROM_MODEL (env) # Trains the agent with env' ;
end
```

6 Implementation

6.1 State-of-the-art - D4PG

Since the thesis is focused on exploring model-based RL techniques in a time-constrained manner and comparing it to the state-of-the-art, we used a D4PG implementation from the OpenAI gym leaderboards that has been already hyperparameter-tuned to perform well. As mentioned in previously, D4PG is an updated version of the Actor-Critic DDPG which uses approximators both for the policy(actor) and Q (critic).

D4PG Appendix:

Hidden layers (Same for actor and critic)	[400, 300]	
Hidden layer type	Dense	
Optimiser	Adam	
Critic learning rate	0.0001	
Actor learning rate	0.0001	
Batch size	256	
Replay memory size	1000000	
Tau	0.001	For transferring weights from target network to online: $W = tau * w + (1 - tau) * w\_target$
Gamma	0.99	Used for discounting future reward in bellman equation
Priority α	0.	Controls the randomness vs prioritisation of prioritised memory replay (0.0 = Uniform sampling, 1.0 = Greedy prioritisation)
Priority β start	0.4	Starting value of beta - controls to what degree weights influence the gradient updates to correct for the bias introduced by priority sampling (0 - no correction, 1 - full correction)
Priority β end	1	Beta will be linearly annealed from its start value to this value throughout training
Gaussian noise scale	0.3	The harshness of Gaussian noise applied to the model's action, used for controlling exploration
Noise decay	0.999	Decay rate for Gaussian noise each episode to decrease exploration as time goes

6.2 Modelled environment action planning - AP

For our modelled environment, we employed a dual input and output model, so as to replicate the real environment as closely as possible. The model has hidden layers for the inputs separately, "middle" hidden layers after the input network outputs have been combined, and finally two outputs with their hidden layers branch off from the middle. We have gotten these parameters from optimal hyperparameter searching. However sklearn's GridSearchCV, a widely popular hyperparameter searching algorithm, does not support multi-output models, so we had to devise our own hyperparameter search algorithms.

Environment model appendix

Input Hidden layers (For Each)	[32, 64]
Middle Hidden layers	[256,256]
Output Hidden layers (For Each)	[256, 64]
Hidden Layer Type	Dense
Hidden layers activation	ReLU
Final layer activation	Linear
Optimiser	Adam
Learning rate	0.001

Boiled down pseudocode for action planning (Approach 2):

```
session = tf.get_session ;
s = env.starting_state ;
total_r =tf.constant(0) ;
for a in actions do
    (s', r') = env'([s, a]) ;
    s = s' ;
    total_r + = r ;
end
opt = GradientDescentOptimizer(lr);
gradients_variables = opt.compute_gradients(-total_r, actions);
clipped_gradients = [(clip(grad, -1, 1), var) for grad, val in gradients_variables];
train_op = opt.apply_gradients(clipped_gradiens);
for i in range(training_length) do
    session.run(train_op);
end
return actions;
```

We get our session, define our starting state and initialise reward. Actions is a list of randomly sampled actions in our env turned into trainable tensor variables. The important thing to note here in the actions loop is that env'([s,a]) does not evaluate, but returns us tensors s', r' whose values depend on s and a. These same tensors then get passed as well either back into env' or accumulated to total_r, constructing a chain of tensor dependencies. This leaves us with actions, which if one were tweaked, would impact all further states and rewards.

After constructing the full chain, we initialise our optimiser and compute the gradients and values between the total reward and actions. Lastly, we set our training operation and run it. In the end, we return the planned actions.

A critical detail here is the clipping of gradients, from our derivations of approach 2, we know that the earlier the action, the more effect it has on the reward, as such early actions suffered from exploding gradients.

Furthermore, our env' internally clips inputs, as we can provide inputs in the valid range, but when optimising them, they may go out of bounds, cheating our model for higher reward. Clipping them means there is no change for going out of bounds.

6.3 DDPG with action plan learning - DDPG-AP

As mentioned in Approach 3, we want to try and modify the current state-of-the-art(SOTA) agent by adding to it intermediate steps where AP instead of training the actions trains the weights of an agent which produced the actions. Current SOTA for continuous action and state space is D4PG, an improved version of DDPG. One of the improvements from D4PG is having multiple distributed agents running in parallel. Attempts at making our novel algorithm integrate with these distributed agents in a manner which is not atrociously computationally inefficient have been unsuccessful due to time constraints and the difficult nature of the algorithm. However, since the project's main aim is not a distributed version of this algorithm, but exploring how it does on its own or as an effective modification, we instead made our Approach 3 on the popular DDPG, which is second in performance only to the aforementioned SOTA(Tassa et al., 2018).

Hidden layers (Same for actor and critic)	[256, 128]
Hidden Layer Type	Dense
Optimiser	Adam
Critic learning rate	0.001
Actor learning rate	0.0001
Batch size	128
Replay memory size	200000
Tau	0.001
Gamma	0.99

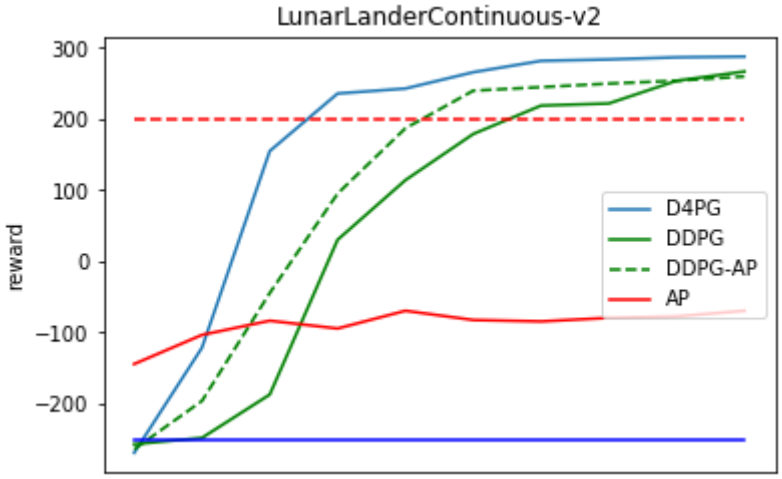
Boiled down pseudocode for action plan learning(Approach 3):

```
for eps in range(training_episodes) do
    s = env.starting_state ;
    if eps % planning_frequency == 0 then
        trainWithPlanning(s, agent, env', lookahead);
    end
    DDPG standard training bellow...
end
Function trainWithPlanning(s, agent, env', lookahead):
    total_r= tf.constant(0) ;
    for _ in range(lookahead) do
        a = agent(s) ;
        s', r' = env'([s, a]) ;
        s = s';
        total_r + = r' ;
    end
    opt= GradientDescentOptimizer(lr);
    gradients_variables = opt.compute_gradients(-total_r,agent.trainable_weights) ;
    clipped_gradients = [(clip(grad, -1, 1), var) for grad, val in gvs] ;
    train_op = opt.apply_gradients(clipped_gradiens) ;
    for i in range(training_length) do
        session.run(train_op);
    end
```

The implementation shares a lot with Approach 2, the primary key difference being that instead of having a list of trainable tensor variables we instead use the tensor output of our agent, meaning our trainable variables, in this case, are the agent's weights. Furthermore, now that instead of finding optimal actions, we are training weights, we want to decrease our learning rate and our training period so that we only nudge them in the right direction.

7 Results and Discussion

7.1 Reward



We compiled the rewards for D4PG, DDPG, DDPG-AP and AP. The rewards are shown with respect to the number of episodes (sample efficiency). The blue bottom line represents the reward for a random policy, while the red dotted line signalise the environment being solved.

For the actor-critic variants, the models were run three times with different seeds. AP was run ten times for each training milestone, training actions for each starting state.

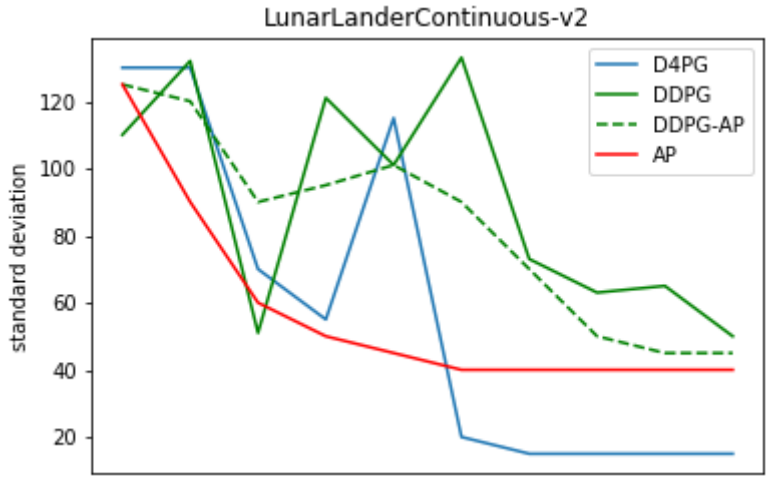
We can observe that our AP module has improved the performance of the popular DDPG by a significant margin. Nonetheless, the most notable observations which are contradicting are that AP only performs better than random policy, but improves DDPG results.

Upon visual inspection of AP in the environment, it seems like the algorithm is performing the correct actions but undershoots or overshoots the target frequently by a small margin. Since the agent starts in a location with random velocity and simulates the trajectory with respect to hundreds of actions, even the tiniest mistake can be costly in this reward scarce environment.

This could be imagined with a human analogy. Let us say a person is in a field and 50 meters in front of him is a tree. He plans on how he will walk to the tree, closes his eyes and walks in the way he planned. Chances at arriving exactly at it are slim, not because of a big mistake, but due to the fact that small inaccuracies over the journey accumulated.

Even though our AP solution does not solve the environment, it can help guide DDPG towards it, and decrease the number of episodes required to be considered solved by approximately 14%.

7.2 Reward variance



We measured the standard deviation of rewards with respect to training episodes.

We also calculate the average standard deviation, by taking the square root of the average of the variances:
D4PG - 75
DDPG - 95.2
DDPG-AP - 87.6
AP - 63.1

We can see that out of our actor-critic methods D4PG has the lowest standard deviation. This can be attributed to the distributed multi-agent approach of D4PG, which curbs its overall variance by having multiple agents training at the same time. Unsurprisingly the most stable approach was a SL based one - AP. This also led to the decrease in the average standard deviation of our augmented agent approach DDPG-AP, when compared to the original a decrease of 8%.

7.3 Conclusion

AP underperformance can be attributed to trying to model what is essentially a simple physics engine, as such small inaccuracies can accumulate, especially in long action sequences. A possible way to go around this would be to split the entire action sequence into subsequences. Train a subsequence of actions, then act them in the real environment and repeat. This would limit the accumulation of inaccuracies and would let the model correct its course after every subsequence.

However, this suffers from the fact that in reward scarce environments where we might only get a reward on the last step, our model would not know how to prioritise as the end of the subsequence of actions may not reach a reward. A way to circumvent this would be by using our modified actor-critic agent, and instead of using the reward accumulated from our env' we would pass the states generated into the critic and accumulate critic evaluation. Since the critic is a Q function which tells us how good it is to be in a position, this would allow us to plan without the full amount of actions. Another improvement that this would let us achieve would be scrapping the reward output from our env' as it would be unnecessary, reducing the complexity of our environment model.

The most interesting remaining question is how our AP modification would improve state-of-the-art D4PG performance. Due to prioritisation under the given time constraints, it was not possible to implement and test this, but we hope that our future work will explore this and if concrete improvement is made on a range of environments it could be published and contest for the coveted state-of-the-art title.

Finally, our research aimed to explore the use of supervised learning techniques on modelled envir-

onments, with a goal in mind to either increase sample efficiency or reduce variance. Our modified DDPG-AP achieved both of these, improving sample efficiency by 14% and decreasing variance by 8%. These are very significant results as this is an improvement on some of the crucial pain points in deep RL. The novelty of our approach explored a completely new way of acting in a RL environment which can possibly open new directions for further research.

References

Arulkumaran, K., Cully, A. & Togelius, J. (2019). Alphastar: An evolutionary computation perspective, In *Proceedings of the genetic and evolutionary computation conference companion*.

Azizzadenesheli, K., Yang, B., Liu, W., Brunskill, E., Lipton, Z. C. & Anandkumar, A. (2018). Sample-efficient deep rl with generative adversarial tree search. *ArXiv*, *abs/1806.05780*.

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N. & Lillicrap, T. P. (2018). Distributed distributional deterministic policy gradients. *CoRR*, *abs/1804.08617*arXiv 1804.08617. http://arxiv.org/abs/1804.08617

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

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Brown, N. & Sandholm, T. (2019). Superhuman ai for multiplayer poker. *Science*, *365*(6456), "url–https://science.sciencemag.org/content/365/6456/885.full.pdf", 885–890. https://doi.org/10.1126/science.aay2400

Buesing, L., Weber, T., Racaniere, S., Eslami, S. M. A., Rezende, D., Reichert, D. P., Viola, F., Besse, F., Gregor, K., Hassabis, D. & Wierstra, D. (2018). Learning and Querying Fast Generative Models for Reinforcement Learning. *arXiv e-prints*, arXiv 1802.03006, arXiv:1802.03006.

Doerr, A., Daniel, C., Schiegg, M., Nguyen-Tuong, D., Schaal, S., Toussaint, M. & Trimpe, S. (2018). Probabilistic recurrent state-space models. *arXiv preprint arXiv:1801.10395*.

Fathinezhad, F., Derhami, V. & Rezaeian, M. (2016). Supervised fuzzy reinforcement learning for robot navigation. *Applied Soft Computing*, *40*, 33–41.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G. & Pineau, J. (2018). An introduction to deep reinforcement learning. *CoRR*, *abs/1811.12560*arXiv 1811.12560. http://arxiv.org/abs/1811.12560

Giles, C. L., Kuhn, G. M. & Williams, R. J. (1994). Dynamic recurrent neural networks: Theory and applications. *IEEE Transactions on Neural Networks*, *5*(2), 153–156.

Ha, D. & Schmidhuber, J. (2018). World models. *CoRR*, *abs/1803.10122*arXiv 1803.10122. http://arxiv.org/abs/1803.10122

Hafner, D., Lillicrap, T., Ba, J. & Norouzi, M. (2019). Dream to control: Learning behaviors by latent imagination.

Henderson, J., Lemon, O. & Georgila, K. (2008). Hybrid reinforcement/supervised learning of dialogue policies from fixed data sets. *Computational Linguistics*, *34*(4), https://doi.org/10.1162/coli.2008.07-028-R2-05-82, 487–511. https://doi.org/10.1162/coli.2008.07-028-R2-05-82

Irpan, A. (2018). Deep reinforcement learning doesn't work yet.

Kaiser, L., Babaeizadeh, M., Milos, P., Osinski, B., Campbell, R. H., Czechowski, K., Erhan, D., Finn, C., Kozakowski, P., Levine, S., Sepassi, R., Tucker, G. & Michalewski, H. (2019). Model-based reinforcement learning for atari. *CoRR*, *abs/1903.00374*arXiv 1903.00374. http://arxiv.org/abs/1903.00374

Kielak, K. (2019). Generative Adversarial Imagination for Sample Efficient Deep Reinforcement Learning. *arXiv e-prints*, arXiv 1904.13255, arXiv:1904.13255.

Kielak, K. (2020). Importance of using appropriate baselines for evaluation of data-efficiency in deep reinforcement learning for atari.

Kingma, D. P. & Welling, M. (2013). Auto-encoding variational bayes.

Kumar, N. (2020). The past and present of imitation learning: A citation chain study.

Leibfried, F., Kushman, N. & Hofmann, K. (2016). A Deep Learning Approach for Joint Video Frame and Reward Prediction in Atari Games. *arXiv e-prints*, arXiv 1611.07078, arXiv:1611.07078.

Lillicrap, T. P., Hunt, J. J., Pritzel, A. e., Heess, N., Erez, T., Tassa, Y., Silver, D. & Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv e-prints*, arXiv 1509.02971, arXiv:1509.02971.

Mankowitz, D. J., Dulac-Arnold, G. & Hester, T. (2019). Challenges of real-world reinforcement learning.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*arXiv 1312.5602. http://arxiv.org/abs/1312.5602

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*, 529 EP -. %5Curl%7Bhttps://doi.org/10.1038/nature14236%7D

Oh, J., Guo, X., Lee, H., Lewis, R. L. & Singh, S. (2015). Action-conditional video prediction using deep networks in atari games, In *Advances in neural information processing systems*.

Peters, J., Vijayakumar, S. & Schaal, S. (2005). Natural actor-critic (J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge & L. Torgo, Eds.). In J. Gama, R. Camacho, P. B. Brazdil, A. M. Jorge & L. Torgo (Eds.), *Machine learning: Ecml 2005*, Berlin, Heidelberg, Springer Berlin Heidelberg.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T. Et al. (2019). Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search [Article]. *Nature*, *529*, 484 EP -. %5Curl%7Bhttps://doi.org/10.1038/nature16961%7D

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T. & Hassabis, D. (2017). Mastering the game of go without human knowledge [Article]. *Nature*, *550*, 354 EP -. %5Curl%7Bhttps://doi.org/10.1038/nature24270%7D

Szepesvari, C. (2010). *Algorithms for reinforcement learning*. Morgan; Claypool Publishers.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A. Et al. (2018). Deepmind control suite. *arXiv preprint arXiv:1801.00690*.

van Hasselt, H. P., Hessel, M. & Aslanides, J. (2019). When to use parametric models in reinforcement learning?, In *Advances in neural information processing systems*.

Watkins, C. J. & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, *8*(3–4), 229–256. https://doi.org/10.1007/BF00992696