REINFORCEMENT LEARNING FOR CONTROL AND MULTIPLAYER GAMES

IIB Project Investigating the application of Google Deepmind's AlphaZero Algorithm to Classical Control Problems

UNIVERSITY OF CAMBRIDGE
DEPARTMENT OF ENGINEERING

Author Alex Darch
Supervisor Dr. Glenn Vinnicombe
Assessor Dr. Ioannis Lestas

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.
Signed ________________

St John's College
Cambridge
May 29, 2019

Technical Abstract

Author Alex Darch
Supervisor Dr. Glenn Vinnicombe
Assessor Dr. Ioannis Lestas
St John's College, Cambridge

AlphaZero is a Reinforcement Learning algorithm developed by Google Deepmind. It can be interpreted as a Robust Model Predictive Control (RMPC) algorithm that performs a directed tree search through the future states to return a control action rather than optimising over all future states. This allows it to dramatically reduce computational time, whilst searching more likely trajectories to a greater depth. Additionally, AlphaZero is a learning algorithm that trains a neural network representing a policy through self play. The adversary in self play can be interpreted as a sequence of optimally adverse disturbances.

Controlling dynamical systems is an active area of interest and this project looks at the application of AlphaZero to these as a novel method of control. This has a number of potential benefits above traditional optimal control such as the ability to robustly handle disturbances; readily model non-linear systems; and, due to the generality of the algorithm, the ability to be applied to many control problems. The inverted pendulum was chosen as a simple starting dynamical system which also met the requirements of being sufficiently complex such that a trivial solution was not possible.

In order to adapt AlphaZero into a control algorithm, methods from the field of Model Predictive Control (MPC) and Optimal Control have been modified. These include the design of a cost function and the introduction of a type of receding horizon for the state value, although this is retrospectively calculated. Adding these allowed for the algorithm to deal with continuous states and indefinite horizons with no easily discernable "winner", unlike the board games played by AlphaZero. In addition to this, two different methods of state representation are considered: the true state, $\boldsymbol{x}^T = [x,\ \dot{x},\ \theta,\ \dot{\theta}]^T$, and a 2-dimensional (2D) state, $\boldsymbol{x}^{(2D)}$, which histogrammed (binned) the evolution of positions, (x_t, θ_t), over time on a grid. The true state has the benefit of being continuous with a low number of dimensions, which made it more ideal for this specific problem. However, the 2D state does not rely on measurements of the system velocities. Relying purely on positional information allows for a more generalisable state space representation as, in real systems, states are often unobservable. Furthermore, the binning of the states was proven to be lossless in the limit of infinite time, assuming a deterministic model and that the neural network could emulate an optimal Kalman Filter.

One of the greatest difficulties in adapting AlphaZero was "power matching" - balancing the effect of each agents actions such that they have an equal effect on the system.

The adversary was chosen to act at the tip of the pendulum so that the adversary would be forced to learn a different policy to the player. However, due to the effect of gravity and a different point of action, the adversary and player had unequal effects on the system. It was found that simply choosing the force of the adversary would either lead to the pendulum falling very quickly, meaning that minimal training examples were recorded; or the adversary was too weak to push over the pendulum. In the case of the adversary being too weak, the adversary invariably learnt to push solely in one direction. This is postulated to be the most efficient way for the adversary to maximise the cost of the system. Due to the adversary learning this, robustness to general disturbances was severely hindered. It was found that the player's internal model of the adversary could not predict the actions of more general adversaries, and therefore the player could not control these.

It has been shown that MCTS can be used to improve the control of a system to great effect, and system non-linearities do not impede this. This method can model any system given that it is deterministic and the player has a perfect knowledge of the action space of the adversary. However, more work is needed for this method to be considered "robust control".

0 Contents

1 Introduction

1.1 Project Aims and Motivation

AlphaZero is a Reinforcement Learning algorithm developed by Google Deepmind. It can be thought of as a Robust Model Predictive Control algorithm that performs a directed tree search through the future states to return a control action rather than optimising over all future states. This allows it to dramatically reduce computational time, whilst searching more likely trajectories more deeply. Additionally, AlphaZero is a learning algorithm that trains a neural network representing a policy through self play. The adversary in self play can be considered to be a sequence of optimally adverse disturbances. This project looks at the application of AlphaZero to dynamical systems as a control method. This has a number of potential benefits above traditional optimal control or Reinforcement Learning methods. These include:

Robust Disturbance Handling. The Adversary in AlphaZero is effectively a "worst case scenario disturbance". By incorporating this into the training process, the controller should be able to cope with situations well outside normal operating conditions.

Non-Linear Systems. Neural networks are universal function approximators and, hence with the correct architecture, should be able to model any system.

General Framework. AlphaZero is a general algorithm that should be able to be applied to many different systems with minimal changes and still model the control well.

1.2 Controlling Dynamical Systems

Dynamical systems have long been of great interest to engineers and scientists due to their ability to readily describe real world phenomena. They describe how a system changes through geometric space with time and, in principle, their trajectories can be predicted solely from their initial state. In recent years advances in computing power have allowed numerical methods to solve non-linear systems with relative ease. However, often a precise solution is of less importance than a prediction of the long term qualitative behaviour of the system. Significant effort has been made to find methods to describe this long-term behaviour such as those made by Lyapunov in stability theory [1].

A corollary of the importance of dynamical systems is that influencing their behaviour is particularly useful. Highly non-linear models are difficult to control optimally and robustly, and the few mathematical techniques developed to deal with these can only handle very small subcategories of problems [1, 2]. The study of more general techniques to solve complex control problems has recently come to the forefront of the field with the advent of machine learning techniques such as Reinforcement and Deep Learning. Many of these are not aimed specifically at control problems and are often designed to play games. This project looks at the possibility of adapting one such algorithm - AlphaZero - from playing board games to solving general control problems such as the control of an aircraft in flight or the control of under-actuated robotics by starting with a simple inverted pendulum.

1.3 Control Theory

The first attempts at controlling dynamical systems came from classical control theory, which consisted of a series of “cut-and-try” techniques based largely on adjusting the gains of PID controllers and lead/lag compensators until satisfactory closed loop dynamics were achieved [4].

It wasn’t until the unification of the calculus of variations, classical control, random process theory, and linear/non-linear programming by Bellman in the 1950s [4] that truly optimal control was discovered. Optimal control consists of finding a control law for a specified optimality criterion and can be thought of as a non-linear feedback law, $u(t) = -K(t)x(t)$ based on the initial state, $x(0)$. In a discrete setting, an optimal control law can be found via *dynamic programming* (DP). DP is a systematic procedure for transforming an optimisation over a sequence of h inputs into h minimisations over 1 input (but for all states). The dynamic programming equations are:

$$V(x_k, k) = \min_{u_{k:h-1}} \left(\sum_{i=k}^{h-1} c(x_i, u_i) + J_h(x_h) \right) \tag{1.1}$$

$$= \min_{u_k} \Big(c(x_k, u_k) + V(x_{k+1}, k+1) \Big) \tag{1.2}$$

$$u_k^* = \operatorname*{argmin}_{u_k} \Big(c(x_k, u_k) + V(x_{k+1}, k+1) \Big) \tag{1.3}$$

$$\tag{1.4}$$

Where $c(x_i, u_i)$ is the cost as a function of the state and input at time i, $J_h(x_h)$ is the terminal cost, and $V(x_k, k)$ is the value function.

This enables a backwards recursion to find a sequence of value functions. This can be solved analytically for the linear case with quadratic costs (known as the linear quadratic regulator, LQR) or, if the system is non-linear, via gradient descent. However, over a finite horizon this is essentially open-loop control. Optimal control of this form has been used to control complex dynamical systems such as spaceflight and aileron folding on aircraft [5, 6]. A closed loop extension to this is Model Predictive Control (MPC), which employs a receding horizon rather than a finite or infinite horizon. MPC can therefore easily deal with plant disturbances and uncertainties, constraints, indefinite horizons and can also be extended to get a control law for non-linear systems. MPC has recently been shown to work in trajectory control for interplanetary rovers [7]. Optimal control is limited in that it requires sophisticated models of the environment/plant, and it generally struggles with highly non-linear models (state of the art is currently linearisation about the predicted trajectory). Furthermore, it is only feasible to "grid" up to 5 or 6 dimensions in discrete cases [2].

Robust MPC (RMPC) is a further extension to MPC that attempts to optimise the "worst case" performance under uncertainty by modelling the system dynamics or control input as stochastic processes. RMPC sequentially minimises and maximises the value function and control policy respectively over all state-action pairs, which becomes computationally intractable as this number increases. Furthermore, the optimisation is non-convex and, if the process is continuous, the state and actions must be discretised. Therefore, it is unclear whether RMPC brings reliable benefits [8].

The inverted pendulum is a seminal problem in control theory. It is inherently unstable, under-actuated, dynamically simple, yet highly non-linear making it an ideal teaching aid. For the inverted pendulum system, a standard method of controlling the pole is to use swing-up control followed by LQR. Swing-up control aims to find a homoclinic trajectory through energy shaping to drive the pendulum to the unstable equilibrium. The region near the equilibrium can be approximated as linear and therefore LQR can optimally stabilise the pendulum near it. This is a particularly good method for the inverted pendulum, but it is difficult to generalise to more difficult systems [9]. Note, that due to this simplicity of this problem, it could be solved with RMPC.

1.4 Reinforcement Learning

Two further generalisations to dynamical systems and optimal control as defined in eq. (1.1) are stochastic dynamics and indefinite horizons (i.e. episodic tasks). This discrete time stochastic control process is known as a Markov Decision Process (MPD). In

MDPs the cost function is often written as a reward function and, due to the indefinite nature of the process, the value function for the next step is discounted (where $\lambda \approx 0.9$ typically). Equation (1.5) gives the value function for a MDP, where i is set to zero for notational simplicity, but represents a sum from $i = k \to \infty$.

$$V(x_k) = \max_{u_k}\left(\sum_{i=0}^{\infty} \lambda^i r(x_i, u_i)\right) \tag{1.5}$$

$$= \max_{u_k} \mathbb{E}\Big[r(x_k, u_k) + \lambda V(x_{k+1})\Big] \tag{1.6}$$

$$u_k^* = \underset{u_k}{\operatorname{argmax}}\, \mathbb{E}\Big[r(x_k, u_k) + \lambda V(x_{k+1})\Big] \tag{1.7}$$

Reinforcement Learning (RL) aims to learn the optimal policy, $\pi^*(x_k)$ $(= u^*(x_k)$ in control) of an MDP. This differs from optimal control in its inherent stochastic nature and therefore can lead to intractable search spaces. A solution to this is to learn form sample trajectories. Algorithms such as Q-Learning, SARSA and DYNA have recently had great success in control applications such as their use in controlling mobile robots [11, 12]. Furthermore, the advent of neural networks has led to the extension of these to functional approximations from tabula-rasa methods, making the control of highly non-linear dynamical systems possible. Notably, Deepmind's recent success with training a robot to gently manipulate objects [13], would not be possible to reproduce using classical or modern control techniques due to dimensional problems.

1.5 AlphaZero

AlphaGo Zero is a revolutionary Reinforcement Learning algorithm that achieved superhuman performance in the game of Go, winning 100–0 against the previously published, champion-defeating AlphaGo. It's successor, AlphaZero, is a generalised version that can achieve superhuman performance in many games. There are two key sub-algorithms that form the basis of their success: Monte-Carlo tree search (MCTS) for policy improvement, and a deep CNN for the neural policy and value network. Policy iteration is then implemented through self-play. AlphaGo Zero and AlphaZero differ only in the latter's use of a single neural network that is updated iteratively, rather than evaluated against the previous one, and by not taking advantage of the symmetries of the games.

A key feature of AlphaZero is that it only requires the ability to simulate the environment. It does not need to be told how to win, nor does it need an exact model of the system dynamics, $p(s_{t+1}|s_t, a_t)$, as this can be learnt through self-play. Furthermore, the algorithm often "discovers" novel solutions to problems, as shown by *move 37* in a game of Go against the reigning world champion, Lee Sedol. This makes it particularly suitable for learning to control complex dynamical systems where approximate simulations can be made.

Note that AlphaZero can been seen as a RMPC algorithm that uses a MCTS rather than optimising over all state-action pairs, thereby making the computation tractable.

1.5.1 Self Play and The Neural Network

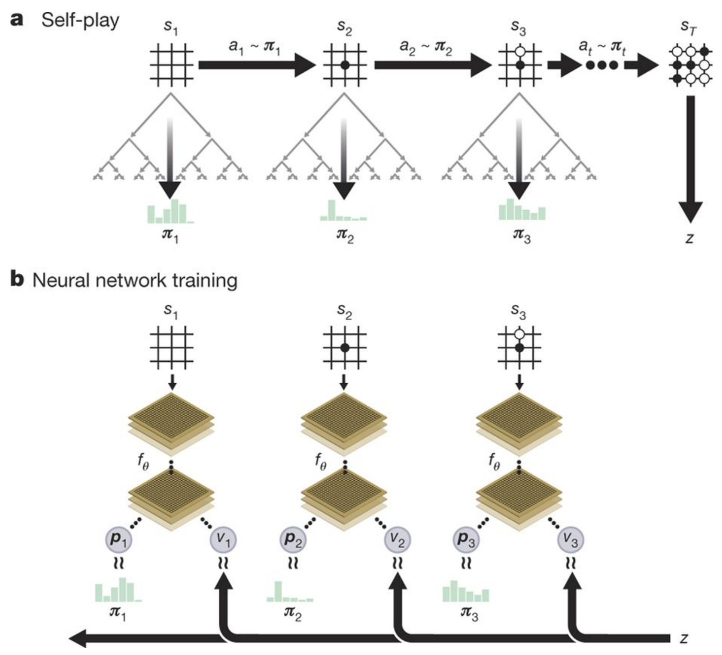


Figure 1.1: A schematic showing how self-play and policy training are performed. Taken from [14].

Figure 1.1a shows how self-play is performed in AlphaZero. A game, known from now on as an episode, $\{s_1, ..., s_T\}$ is played and for each state, s_t, a Monte-Carlo Tree Search is performed, guided by the current policy f_θ. The MCTS outputs an improved action probability mass function ($p.m.f$), $\boldsymbol{\pi}_t = [p(a_1|s_t), p(a_2|s_t), ..., p(a_N|s_t)]$. The next move is then selected by sampling from $\boldsymbol{\pi}_t$. The agent that plays the final move then gets scored (e.g in chess $z \in \{-1, 0, 1\}$ for a loss, draw or win respectively). The game score, z, is then appended to each state-action pair depending on whose turn it was in that position, to give training examples of the form $(s_t, \boldsymbol{\pi}_t, (-1)^{\mathbb{I}(winner)}z)$.

The neural network, $f_\theta(s)$ takes a board state, s, and outputs a $p.m.f$ over all actions and the expected outcome, $(\boldsymbol{p}_\theta, v)$ (fig. 1.1). The networks are initialised with $\theta \sim \mathcal{N}(0, \epsilon)$, where ϵ is small. The neural network is trained to more closely match the MCTS-informed action $p.m.f.$s, $\boldsymbol{\pi}_t$, and the expected outcome (state-value), v_t to z.

The loss function for the neural network is given by:

$$\mathcal{L} = (z - v)^2 - \boldsymbol{\pi} \cdot log(\boldsymbol{p}) + c||\theta||^2 \tag{1.8}$$

For chess, the input state consists of an image stack of 119 8x8 planes representing the board state at times $\{t, t-1, ..., t-8\}$, and planes representing repetitions, colours and castling etc. The output action $p.m.f$ is a 8x8x73=4672 vector representing every possible move from each piece, illegal moves are then masked. The output value is predicted such that $v \in (-1, 1)$. The network itself consists of an input convolutional block and two separate “heads”. The policy head has *softmax* activation function, preceded by series of *ReLU* linear layers and batch normalisation layers. The value head also has this but with a *tanh* output activation. The input convolutional block consists a single convolutional layer followed by 19 to 39 residual blocks (depending on the game).

1.5.2 Monte Carlo Tree Search

Figure 1.2 depicts the steps involved in a monte-carlo tree search (MCTS) iteration, and are described below.

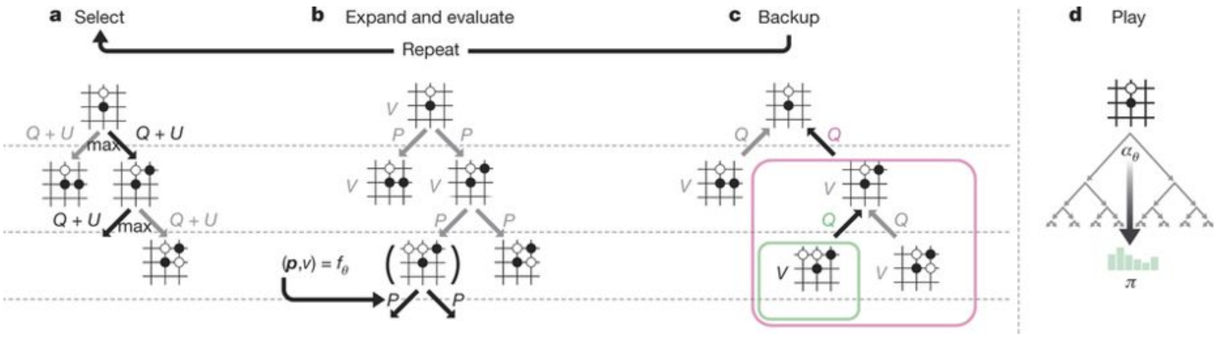


Figure 1.2: A schematic outlining the steps involved in a monte-carlo tree search. Taken from [14].

Figure 1.2a: A MCTS is performed at each step of an episode. The state at which the tree search starts then becomes the root state, s_{root}. From the root state, the tree search can move to an edge (s, a) by selecting an action, a. To ensure exploration, before selection a prior, $P(s,a) = (1-\epsilon)\boldsymbol{p}_{\theta,root} + \epsilon\boldsymbol{\eta}$, where $\boldsymbol{\eta} \sim Dir(0.3)$ and $\epsilon = 0.25$ is added. Each edge stores a prior action- *p.m.f.* and an initial value $[\boldsymbol{p}_{\theta}(s_{t-1},a_{t-1}), v(s_{t-1},a_{t-1})] = f_{\theta}(s_t)$ [1]; a visit count, N(s, a); and an action-value, Q(s, a). Actions are selected by maximising an the action-value plus an upper confidence bound, which encourages exploration. The constant c (~ 1) can be increased to encourage exploration.

$$a_{selected} = \underset{\forall a}{\operatorname{argmax}} \left\{ Q(s,a) + c \cdot \boldsymbol{p}_{\theta}(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \right\} \tag{1.9}$$

Figure 1.2b: Once a leaf node (N(s, a) = 0) is reached, the neural network is evaluated at that state: $f_{\theta}(s) = (p(s,\cdot), v(s))$. The action *p.m.f.* and state-value are stored for the leaf state.

Figure 1.2c: The action-values, Q, are calculated as the mean of state-values in the subtree below that action. The state-value are then calculated as the mean of all the action-values branching from that state, and so on.

[1]note that action a_{t-1} from state s_{t-1} yields state s_t, if the system is deterministic. Therefore, $(s_{t-1}, a_{t-1}) = (s_t)$ in this setting.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s_{t+1}|s_t,a_t} v(s_{t+1}) \tag{1.10}$$

Figure 1.2d: After a set number of simulations (1600), the MCTS-improved action $p.m.f$s, $\boldsymbol{\pi} = p(\boldsymbol{a}|s_{root}) \propto N^{1/\tau}$, are returned where N is the visit count of each move from the root state and τ controls the sparseness of the probability mass function ($\{\tau = 0\} \rightarrow$ argmax$\{N(s,a)\}$). For self-play, $\tau = 1$ for the first 30 moves and $\tau \rightarrow 0$ thereafter. For evaluation $\tau \rightarrow 0\ \forall t$. If $\tau \rightarrow 0$ then $\boldsymbol{\pi}$ becomes one-hot and the loss function of the neural network makes sense as a prediction of a categorical distribution.

1.5.3 Policy Iteration

AlphaZero uses a single neural network that continually updates, irrespective of whether it is better or worse than the previous network. Whereas AlphaGo Zero games are generated using the best player from all previous iterations and then only replaced if the new player wins $> 50\%$ of evaluation games played.

Evaluation is done by playing 400 or more greedy ($\tau \rightarrow 0$) games of the current best neural network against the challenging neural network. The networks are then ranked based on an elo scoring system (the standard ranking system used in professional chess).

2 Theory and Methods

2.1 The Inverted Pendulum (IP)

2.1.1 Dynamics

The Inverted Pendulum is an inherently unstable system with highly nonlinear dynamics, and is under-actuated.

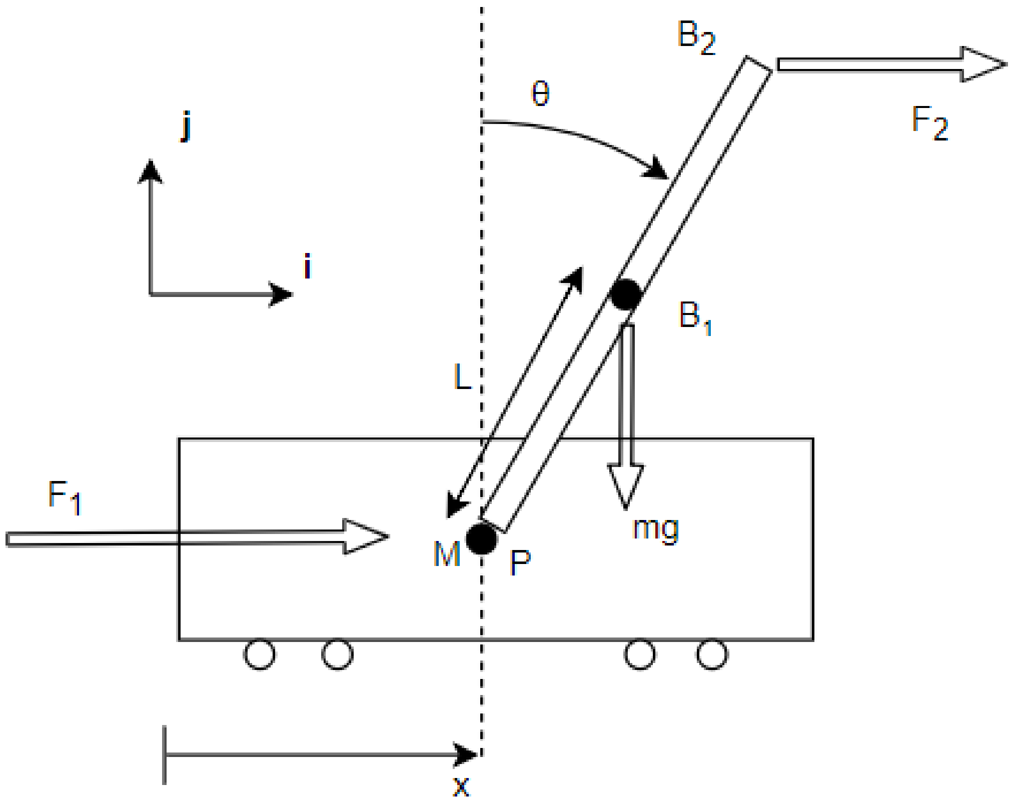


Figure 2.1: A free-body diagram of the inverted pendulum system. For the OpenAI IP the system is in discrete time and constrained by $L = 0.5$m, $m = 0.1$kg, $M = 1$kg, $F = \pm 10$N, $x_{max} = \pm 2.4$m, $\theta_{max} = \pm 12^o$.

The full state space equations for the inverted pendulum as defined in fig. 2.1 are given by:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \frac{\left(\frac{2M+m}{m}F_2 - F_1\right)cos\theta + g(M+m)sin\theta - mL\dot{\theta}^2 sin\theta cos\theta}{(M+msin^2\theta)} \\ \dot{\theta} \\ \frac{F_1 - F_2cos(2\theta) + msin\theta(L\dot{\theta}^2 - gcos\theta)}{L(M+msin^2\theta)} \end{bmatrix} \tag{2.1}$$

Using Lyapunov's indirect method, we can write the linearised equations about the equilibrium, $\boldsymbol{x}_e = [x_e, \dot{x}_e, \theta_e, \dot{\theta}_e]^T = [0, 0, 0, 0]^T$, as:

$$\begin{bmatrix} \delta\dot{x} \\ \delta\ddot{x} \\ \delta\dot{\theta} \\ \delta\ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(m+M)}{ML}g & 0 \end{bmatrix} \begin{bmatrix} \delta x \\ \delta\dot{x} \\ \delta\theta \\ \delta\dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{M} & -\frac{1}{M} \\ 0 & 0 \\ -\frac{1}{ML} & \frac{2M+m}{mML} \end{bmatrix} \begin{bmatrix} \delta F_1 \\ \delta F_2 \end{bmatrix} \tag{2.2}$$

The eigenvalues are given by $det(\lambda I - A) = \lambda^2(\lambda^2 - \frac{(m+M)}{ML}g) = 0$. Therefore, the system is unstable about $\boldsymbol{x}_e$ due to the right half plane pole, $\lambda = \sqrt{\frac{(m+M)}{ML}g}$. Additionally, the time constant of this unstable system is $\tau = \sqrt{\frac{ML}{g(m+M)}} = 0.22s$. Note, if $M >> m, \tau \rightarrow \sqrt{\frac{L}{g}}$, which is the time constant for a simple pendulum.

OpenAI's gym is a python package that supplies an inverted pendulum environment built-in (called CartPole). This environment was wrapped to use the dynamics above, whilst providing a rendering function shown in fig. 2.2.

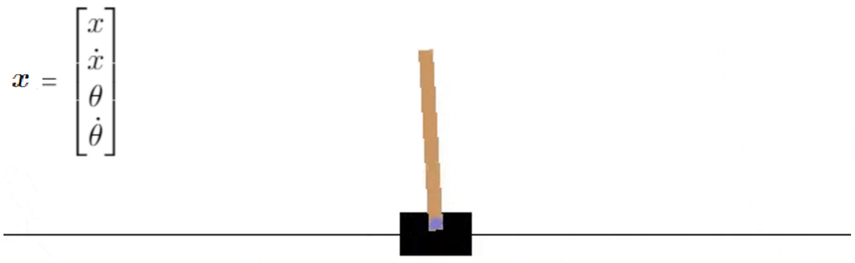

$\boldsymbol{x} = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$

Figure 2.2: The OpenAI gym CartPole environment. The classical state representation is shown in the top left. Actions by the player and the adversary are taken as an impulse to the left or right as defined in fig. 2.1.

2.1.2 Cost and Value Function

For each step, a cost is calculated as in eq. (2.3), where $\boldsymbol{w}^T = [w_1, w_2, w_3, w_4] = [0.4, 0.1, 0.7, 1]$ and $0 \geq c(x_t, u_t) \geq -1$. The weights, $\boldsymbol{w}$, were chosen through empirical measurement of the the importance of each state.

$$c(x_t, u_t) = -\frac{1}{\sum_i w_i} \cdot \hat{\boldsymbol{x}}^T \begin{bmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & 0 & w_4 \end{bmatrix} \hat{\boldsymbol{x}} \tag{2.3}$$

$$\text{where, } \hat{\boldsymbol{x}}^T = \left[\frac{x_t}{x_{max}}, \frac{\dot{x}_t}{\dot{x}_{max}}, \frac{\theta_t}{\theta_{max}}, \frac{\dot{\theta}_t}{\dot{\theta}_{max}}\right]^T \tag{2.4}$$

Weighting on the inputs was set to zero as there are only two inputs for this problem, thus the cost can be written as $c(x_t)$. The max values can be approximated experimentally (note, $x_{max} = 2.4$ and $\theta_{max} = 12^o$ are given constraints):

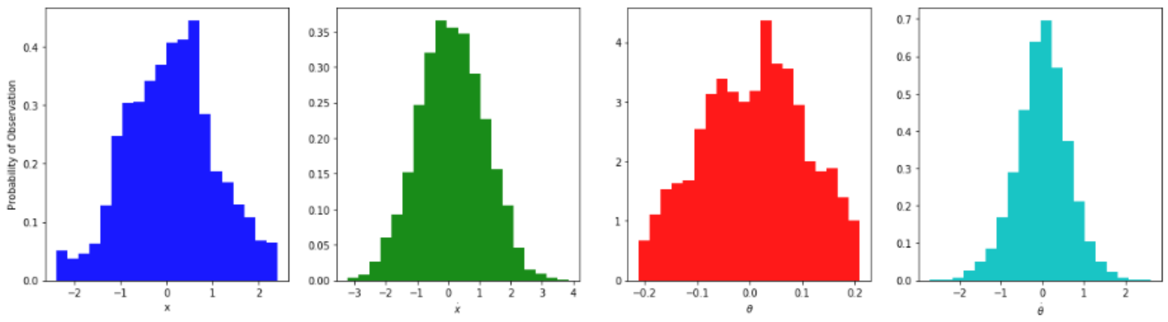


Figure 2.3: Histograms of typical state values. The frequencies greatly depend on the quality of the controller, with better controllers giving much narrower distributions. However, these are typical for a controller of medium efficacy over many episodes where the starting state is randomised (possibly to an uncontrollable state).

Suitable estimates for the the values of $\dot{x}_{max}$ and $\dot{\theta}_{max}$ are thus approximately 3 and 2 respectively. After the episode has completed, the "true value" of each state is computed as the discounted future costs over a finite horizon (eq. (2.5)). I.e. the value of a state, $\boldsymbol{x}_t$, is given by $v(\boldsymbol{x}_t) \propto c(\boldsymbol{x}_t) + \gamma c(\boldsymbol{x}_{t+1}) + ... + \gamma^k c(\boldsymbol{x}_{t+k})$, where k is the length of the horizon (chosen to be 14). A horizon constraint of $\gamma^k < \frac{1}{20}$ was chosen as it is a standard factor for insignificance. Therefore γ is calculated as $\gamma < \frac{1}{20}^{\frac{1}{k}} \implies$ for $k = 14$, $\gamma = 0.8$.

$$v_0 = \frac{\sum_{\tau=0}^{k} \gamma^\tau c(x_\tau)}{\sum_{\tau=0}^{k} \gamma_\tau}, \qquad \text{where } \gamma^k < \frac{1}{20} \tag{2.5}$$

Where for simplicity of notation, $v_0 = v(t)$, the state value at step t. Once a terminal state (either out of bounds, or the maximum number of steps) has been reached, the simulation continues for k steps in order for a value to be calculated for every step of the episode.

2.1.3 State Representations

The state can be represented in a number of ways. The simplest method would be $\boldsymbol{x} = [x, \dot{x}, \theta, \dot{\theta}]$. This has a number of advantages such as lower computational cost, greater numerical accuracy (if the process is fully observable) and simpler implementation. Conversely, a 2-dimensional (2D) representation may be used. There are several possibilities for this, all of which first require binning $\boldsymbol{x}$:

(1) A matrix stack of x vs $\dot{x}$ and θ vs $\dot{\theta}$, both of which would only have one non-zero entry. This scales as b^n where b = number of bins and n = number of states.

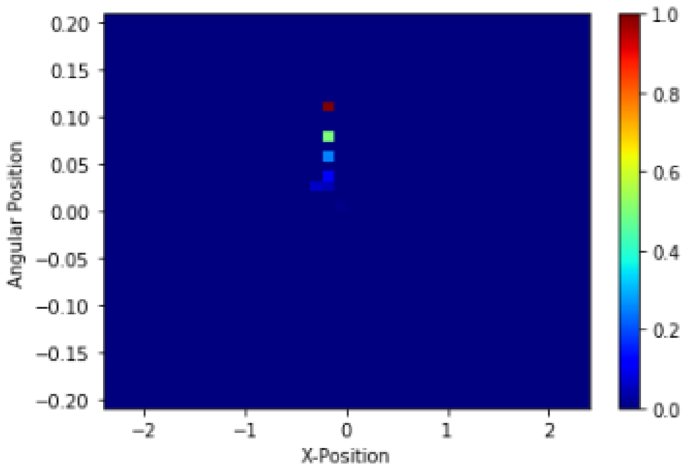


Figure 2.4: An example of a 2D state representation with 40 bins and 8 random actions have been taken. The discount factor, λ, is 0.5.

(2) A matrix stack of x_t vs x_{t-1} for all states. Similarly this scales as b^n, however the derivative states do not need to be plotted as these can be inferred. This has the advantage that, if the derivatives are not observable, they are built into the 2D representation, however, if they are observable then this is less accurate than (1).

(3) A matrix of discounted previous states, forming a motion history image. An example of this is shown in fig. 2.4, and algorithm 1 shows the implementation details. This was the chosen representation, where the discount factor, λ was chosen to be 0.5 as this ensures that successive summing of states gives distinct values (e.g. 0.5+0.125 is the only way to get 0.625, therefore the IP must have been in that position both 1 and 3 time steps ago).

A 2D representation such as this allows us to use a convolutional neural network (CNN), which has the benefit of various transformation invariances - these are particularly useful for the inverted pendulum since it is symmetric.

The linearised inverted pendulum (valid for small time steps) can be modelled as a state space model:

$$\boldsymbol{x}_t^{(2D)} = C\boldsymbol{x}_t + \boldsymbol{V}_t \qquad \boldsymbol{V}_t \sim \mathcal{U}\left(\begin{bmatrix} \frac{1}{\delta x} \\ \frac{1}{\delta \theta} \end{bmatrix}\right) \tag{2.6}$$

$$\boldsymbol{x}_t = A\boldsymbol{x}_{t-1} + B\boldsymbol{u}_t \tag{2.7}$$

Where A and B are the linearised system dynamics, and C is the linear transformation to a 2D state space, with quantisation noise $\mathbf{V}$ modelled as a uniform random disturbance, $\mathcal{U}$, centred on (0, 0).

The initial mean squared error (MSE), and the propagation of the error (when estimated optimally) are given by eqs. (2.8) and (2.9) where $\delta x = \delta\theta$ (derivation details can be found in appendix A.2).

$$\Sigma_0 = \sigma_v^2 I = \begin{bmatrix} \frac{\delta x^2}{12} & 0 \\ 0 & \frac{\delta\theta^2}{12} \end{bmatrix} \tag{2.8}$$

$$\Sigma_{n+1} = (I - \bar{\Sigma}_{n+1} C^T (\sigma_v^2 I + C\bar{\Sigma}_{n+1} C^T)^{-1} C)\bar{\Sigma}_{n+1} \tag{2.9}$$

Where $\bar{\Sigma}_{n+1} = A\Sigma_n A^T$. It can be shown (see appendix A.2) that as $t \to \infty$, the covariance of quantisation decays to zero since $\underset{\delta x \to 0}{lim} \sigma_v^2 = 0$ and therefore this form for a 2D state becomes lossless (assuming that the neural network acts as an optimal Kalman filter). For a given bin size, the MSE decreases linearly with time, however, the MSE decreases quadratically with bin size. Therefore, a smaller bin size will quadratically improve the rate of decay but quadratically increase the computational load of when computing the state and through the neural network. Therefore, there is a tradeoff in choosing the number of bins between computational complexity and state error.

With a non-zero bin size the overall error can be reduced by binning more densely in regions in which the IP is expected to spend more time. Figure 2.3 shows that the state visit frequencies roughly correspond to normal distribution, therefore by transforming the bins with an inverse Gaussian c.f.d. a flattened distribution can be obtained with a greater density of bins in the centre (fig. 2.5). This has the additional benefit of allowing finer control where it is needed. For example, if the pole is far from the equilibrium the optimal action more easily determined, and subject to less change with small state variations, therefore coarser binning can be used.

Algorithm 1 Create 2D State

```
1: function GETSTATE2D($\hat{\boldsymbol{x}}_{t-1}^{(2D)}, binEdges, nBins$)
2:     $\hat{\boldsymbol{x}} \leftarrow$ getNormedState()
3:     for all $x_i \in \hat{\boldsymbol{x}}_t$ do
4:         $x_i \leftarrow$ argmin$|binEdges - x_i|$        ▷ Get the index of the nearest bin edge.
5:     end for
6:     $HistEdges \leftarrow$linspace$(-0.5, nBins - 0.5, nBins + 1)$        ▷ Centre by shifting -0.5
7:     $\hat{\boldsymbol{x}}_t^{(2D)} \leftarrow$ histogram2d$(x, \theta,$ bins $= (histEdges, histEdges))$        ▷ Inbuilt function
8:     return $\hat{\boldsymbol{x}}_t^{(2D)} + \lambda\hat{\boldsymbol{x}}_{t-1}^{(2D)}$
9: end function
```

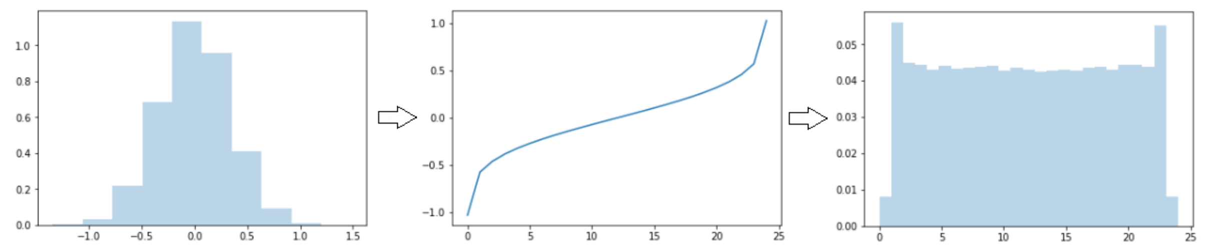


Figure 2.5: Binning of a Gaussian distribution with bin edges scaled with an inverse Gaussian c.f.d. For this example there are 25 bins.

2.1.4 Discretisation and Continuous Time

For a small enough discrete time step, the simulation will approximate the continuous process very well. Under a continuous action space, the sampling rate must be faster than the system dynamics to ensure controllability. When the action space is restricted to $u \in \{-1, 1\}$, there is a trade off between the sampling rate and the size of the action space. If the sampling rate is fast enough pseudo-continuous actions can be achieved via pulse-width modulation. For the IP system described, the time constant, τ, is approximately 0.022s. Moreover, the agents move alternately, therefore the sampling time step δt, must be less than 0.011s, however due to computational limitations, a time step of 0.01s was chosen.

The positional change per time step can be computed by considering the average maximum velocities: $3m/s$ and $2rad/s$. Thus the maximum positional change between an agent's actions is $\dot{x}_{i,max} \times \frac{\delta t}{x_{i,max}} = 2.5\%$ and 30% respectively. In practice a 30% change in θ would only occur around θ_{max} and would be uncontrollable. Therefore, the choice of $\delta t = 0.01$ is not detrimental to the control, but will not give pseudo-continuous actions.

This maximal positional change per time step also informs the bin size needed. Ideally, each time step lands within a new bin, therefore for the x-position, $\frac{1}{2.5\%} = 40$ is the minimum number of bins needed. Similarly, $\frac{1}{30\%} = 3$ is the minimum number of bins needed for the angular position, however, empirically, θ is a more "important" state variable due to its relative ease of exceeding θ_{max}. Therefore, given that the bins are more finely space near (0, 0) and for simplicity, 40 bins for both x and θ was chosen.

2.2 Self Play and Adversaries

2.2.1 Cost and Value Functions

In AlphaZero, the adversary is working to minimise the value function whereas the player is working to maximise it. For board games, where agents are on equal footing, a value function $(-1 < v < 1)$ representing the expected outcome can be used. This has the advantage of symmetry - when the board is viewed from the other agent's perspective the value can be multiplied by -1, giving the expected outcome for that agent.

The adversary for the inverted pendulum has the additional advantage of gravity, making the play asymmetric. Given that both the state-value and cost are $-1 < v, c < 0$, multiplying by -1 would mean the adversary is maximising a value function $0 < v < 1$. State-values outside these ranges have no meaning. If a single neural network is used, values close to equilibrium may be predicted into the wrong range. Consequently, both the adversary and the player must predict true state-values. This also has the advantage of maintaining an intuitive meaning of the state-value.

2.2.2 Choice of Action

The inverted pendulum system is symmetric in both x and θ. By taking advantage of this the number of training examples could be doubled by reflecting episodes along the axis of symmetry. However, as shown with AlphaZero [15], this provides minimal benefit and also hinders generalisation. The adversarial point of action was chosen to be at the top of the pole, acting horizontally (fig. 2.1), thus ensuring that two distinct policies must be learnt, rather than one being just inverse probabilities of the other. For simplicity $u_{adv} \in h \cdot \{-1, 1\}$ was chosen, where h is a handicap applied to the adversary.

The handicap can be approximated using the linearised dynamics eq. (2.2) by investigating what force would be needed by each agent to achieve the same change in the state (eqs. (2.10a) and (2.10b)) (setting $\delta \boldsymbol{x} = \boldsymbol{0}$).

$$\delta\ddot{x} = \frac{1}{M}\delta F_1 - \frac{1}{M}\delta F_2 \quad \Longrightarrow \quad \delta F_2 \approx \delta F_1 \tag{2.10a}$$

$$\delta\ddot{\theta} = -\frac{1}{ML}\delta F_1 + \frac{2M+m}{mML}\delta F_2 \quad \Longrightarrow \quad \delta F_2 \approx \frac{m}{2M+m}\delta F_1 \tag{2.10b}$$

Empirically, the pendulum is more likely to fall over (exceed θ_{max}) rather than to exceed x_{max}. Therefore, a good estimate of h would be $\frac{m}{2M+m} = \frac{0.1}{2+0.1} \approx 0.0476$. When testing this it was found that if F_1 was set near $0.04 - 0.05$ initially, the adversary gets the upper-hand quickly and the pendulum falls very quickly, not allowing the player to gain any training experience. Therefore, for the first N iterations, the adversary force was set to 0. This was increased as the number of iterations increased according to $h = (1 - 0.7^{i-N+1})$, where i is the iteration, and 0.7 was chosen such that the adversary could be implemented on the first iteration without overpowering the still-training player.

2.2.3 Agent Representation

There are two good ways of representing whether it is the adversary's turn for the neural network:

Multiply the board representation by -1 such that opponent pieces are negative. This has the disadvantages that it can only take two agents and, a network that outputs both state-values and action *p.m.f.*'s should predict the same state values for both player and adversary, but predict vastly different actions. Therefore, the values could be decoupled from the actions, which was one of the major benefits of using a single neural network. However, a single network is simpler, and negating the board more closely follows AlphaZero's methodology.

Record the agent with each example and use agent-labeled examples to train different neural networks. Using a neural network for each agent causes half of the examples to be lost as only the relevant agent's examples are used to train each network. However, this does not suffer from the problems above and can cope with agents with a different number of possible actions more easily.

Recording the agent with each example was therefore chosen. Note, in the case of the inverted pendulum, the optimal action is the inverse of the worst action. However, this is not a general result. For example, in a system with non-linear and asymmetric dynamics it is possible to have the target perpendicular to the closest edge of the invariant set, thus for the adversary it is better to push the system into instability rather than away from equilibrium.

2.2.4 Episode Execution and Policy Iteration

Pseudocode for episode execution following the sections above is shown in Algorithm 2.

The action, u, is sampled from $\boldsymbol{\pi}$. However, after S steps, the temperature, $\tau \to 0$, which is equivalent to $u = \text{argmax}\ \boldsymbol{\pi}$.

The overall policy iteration algorithm is given by section 2.2.4.

2.3 Neural Networks

2.3.1 Loss Functions and Pareto

The following loss function is minimised when training the neural networks:

$$\mathcal{L} = \underbrace{\sum_t (v_\theta(\boldsymbol{x}_t) - v_t)^2}_{\mathcal{L}_{value}} + c_{pareto} \cdot \underbrace{\boldsymbol{\pi}_t \cdot log(\boldsymbol{p}_\theta(\boldsymbol{x}_t))}_{\mathcal{L}_{action}} + \underbrace{c||w_{nnet}||^2}_{regularisation} \tag{2.11}$$

Algorithm 2 Execute Episode

```
 1: function EXECUTEEPISODE
 2:     $example \leftarrow []$
 3:     $\boldsymbol{x}, \boldsymbol{x}^{(2D)}, c \leftarrow$ resetEpisode()    ▷ Set initial $\boldsymbol{x}$ randomly and initialise the cost
 4:     $agent \leftarrow 0$
 5:     repeat
 6:         $\boldsymbol{\pi} \leftarrow$getActionProb($\boldsymbol{x}, \boldsymbol{x}^{(2D)}, agent$)    ▷ Perform MCTS Simulations
 7:         $example$.append(($\boldsymbol{x}^{(2D)}, \boldsymbol{\pi}, c, agent$))
 8:         $u \sim \boldsymbol{\pi}$    ▷ Sample action. Note after S steps, $\tau \rightarrow 0$
 9:         $\boldsymbol{x}, c \leftarrow$step($\boldsymbol{x}, u$)    ▷ Take next true episode step
10:         $\boldsymbol{x}^{(2D)} \leftarrow$getState2D($\boldsymbol{x}, \boldsymbol{x}^{(2D)}$)
11:         $agent \leftarrow$ nextAgent($agent$)
12:     until episodeEnded($\boldsymbol{x}$)
13:     $example \leftarrow$ convertCostsToValues($example$)
14:     return $example$
15: end function
```

Algorithm 3 Policy Iteration - Training the NeuralNetwork

```
 1: function POLICYITERATION
 2:     $nnet \leftarrow$ NeuralNetwork    ▷ Initialise NeuralNetwork
 3:     $examples \leftarrow$ [ ]
 4:     for $iter$ in policyIterations do
 5:         incrementHandicap($iter$)    ▷ $F_2 = 0$ for first n iters
 6:         for $ep$ in trainingEpisodes($iter$) do    ▷ More episodes for first n iters
 7:             resetMCTS($nnet$)    ▷ Use the most current nnet
 8:             $example \leftarrow$ executeEpisode()
 9:             $examples$.append($example$)
10:         end for
11:         $recents \leftarrow$ getMostRecentExamples($examples$)    ▷ Last N policys' examples
12:         $nnet \leftarrow$ trainNeuralNetwork($recents$, $nnet$)
13:         saveNeuralNetwork($nnet$)
14:     end for
15: end function
```

Assuming that policy action predictions and value predictions are of similar importance, similar magnitudes of action and value losses are desireable. The constant, c_{pareto}, was chosen such that ($\mathcal{L}_{action} \approx \mathcal{L}_{value}$). The minimum entropy for two possible actions is 1bit, and since the action loss is the negative log likelihood of a categorical distribution, the minimum loss is -1. The value losses are calculated using the MSE, which is positive, and has an average value of approximately 0.1 (figs. A.1 to A.4), therefore c_{pareto} was set to 0.1.

During training, the agents were chosen to move probabilistically throughout for the first 14 moves, and then move greedily. The number 14 was chosen as it is the average episode length when the player moves randomly without an adversary.

2.3.2 Architectures

Two neural network architectures are used: one that takes the raw state, $\boldsymbol{x} = [x\ \dot{x}\ \theta\ \dot{\theta}]^T$ as input (the "*baseline*" network) and another that takes the 2D state, $\boldsymbol{x}^{(2D)}$ (the convolutional neural network, "*CNN*"). The adversary and player have separate networks, both predicting the state-value and action *p.m.f*s.

For the network with $\boldsymbol{x}^{(2D)}$ as input, there are 2 fully connected convolutional layers with max-pooling, followed by a feedforward layer with *ReLU* activations. The two "heads" split here and each has fully-connected layer, with the value head outputting v_θ and the action head outputting $\boldsymbol{p}_\theta$. The network with $\boldsymbol{x}$ as input is the same, except the convolutional layers are replaced with 2 feedforward layers.

Both architectures perform training with the *Adam Optimiser* using a learning rate of 0.0005, a mini-batch size of 64, a dropout of 0.3, batch normalisation and 1 epoch. The networks are implemented in pytorch[1].

The neural network is evaluated once every step in the MCTS, therefore in an episode of length L and S MCTS simulations/step, the neural network is evaluated more than $L \times S$ times. The GPU used to run experiments is a single NVIDIA GeForce GTX 1050. The convolutional layers of a neural network are the most computationally expensive part and, increasing with the image size, are more than 50% of the computational time in the architectures defined above. Therefore, there is a trade off between the number of convolutional layers used and computation time. The number of layers was chosen such that running the program takes less than 24hrs. Additionally, the memory of the GPU is 4GB, which limits the number of weights that can be used in the neural networks. It was found that adding an additional layer to the CNN exceeds this memory limit. The architecture of the baseline network was chosen to match that of the CNN.

[1]www.pytorch.org

2.4 MCTS

The general MCTS algorithm, outlined in section 1.5.2, was implemented for the inverted pendulum as in algorithms 4 and 5 .

2.4.1 Tree Nodes and Simulations

The state for the inverted pendulum is continuous ($\boldsymbol{x} \in \mathbb{R}^4$), as opposed to the discrete nature of board games. In order to be able to compare nodes, the states must be stored as unique integers. Each MCTS simulation ends with a leaf node being expanded, therefore after S simulations from a node, S states will be visited. Over the whole episode, $S \times L$ distinct states will be visited in total. Thus, the probability of revisiting a state from a different trajectory is $(S \times L)b^4$, where b is the number of bins for each dimension. Given that $L = 400$ and, if $S > 30$, the maximum recursion depth of the MCTS will be reached - limiting the maximum value of S. Multiplying the states by 10^{12} and converting to integers ensures a vanishingly small probability of encountering a state twice.

2.4.2 Action Selection

Action selection moving down the tree was modified to reflect the asymmetry of the state-value:

$$u^*_{player} = \underset{u \in \mathcal{U}_{player}}{\text{argmax}} \left\{ Q(\boldsymbol{x}, u) + c_{explore} \cdot \boldsymbol{p}_\theta(\boldsymbol{x}, u) \frac{\sqrt{\sum_b N(\boldsymbol{x}, b)}}{1 + N(\boldsymbol{x}, b)} \right\} = \underset{u \in \mathcal{U}_{player}}{\text{argmax}}\ U_{player}(\boldsymbol{x}, u) \tag{2.12}$$

$$u^*_{adv} = \underset{u \in \mathcal{U}_{adv}}{\text{argmax}} \left\{ -Q(\boldsymbol{x}, u) + c_{explore} \cdot \boldsymbol{p}_\theta(\boldsymbol{x}, u) \frac{\sqrt{\sum_b N(\boldsymbol{x}, b)}}{1 + N(\boldsymbol{x}, b)} \right\} = \underset{u \in \mathcal{U}_{adv}}{\text{argmax}}\ U_{adv}(\boldsymbol{x}, u) \tag{2.13}$$

Where c = 1 was used. Negating Q in U_{adv} causes the adversary to minimise the state-value whilst still maximising the upper confidence bound, thus ensuring exploration.

When the inverted pendulum exceeds a constraint the tree search should return -1. Note, the simulation terminates after 400 steps (the arbitrarily set "end" of the episode), but this should not return -1. Therefore, when 400 steps is reached, the neural network is evaluated and v is returned.

2.5 Player and Adversary Evaluation

For problems in which one agent has a distinct advantage, determining the utility of the player's policy compared to other policies becomes difficult. For example, if the pendulum

Algorithm 4 Get Action Probabilities (based on S.Nair's implementation [16])

```
1: function GETACTIONPROB($\boldsymbol{x}_{root}, \boldsymbol{x}^{(2D)}, agent$)
2:     for i in nMCTSSims do                          ▷ Simulate nMCTSSims episodes from $\boldsymbol{x}_{root}$
3:         resetSimulation($\boldsymbol{x}_{root}$)
4:         MCTSSearch($\boldsymbol{x}^{(2D)}, agent$)
5:     end for
6:     $N(\boldsymbol{x}_{root}, u) \leftarrow$getCounts()     ▷ Count the times each edge, $(\boldsymbol{x}_{root}, u)$, was visited.
7:     $N(\boldsymbol{x}_{root}, u) \leftarrow N(\boldsymbol{x}_{root}, u)^{\tau}$          ▷ Control Sparsity with the temperature, $\tau$
8:     return $\pi \leftarrow$ norm($N(\boldsymbol{x}_{root}, u)$)
9: end function
```

stays up for longer, is is difficult to determine whether the player has improved because it is controlling the system better, the adversary is less effective, or both have improved but one improved more. The score used is the cost plus one, $score = c(\boldsymbol{x}) + 1$, such that the score is between 0 and 1, with 1 having the highest utility. If the pendulum has fallen over, the score is 0 for all steps until the episode ends, therefore making it possible to compare between episodes of differing lengths.

Testing is performed by loading a saved policy's weights into the neural networks, and then simulating a number of episodes of this against various agents. Adversarial agents to be tested are the adversaries themselves and a random adversary with differing forces, F_2. Additionally, the effect of the number of MCTS simulations per step is tested for a random adversary.

The mean iteration score is calculated by summing all of the costs over all episodes and taking the mean value for an iteration, rather than taking the mean value for an episode and then taking the mean of the episodes in that iteration.

Algorithm 5 MCTS (based on S.Nair's implementation [16])

```
1: function MCTSSearch($\boldsymbol{x}^{(2D)}, agent$)
2:     if $\boldsymbol{x}$ is terminal then
3:         if fallen then
4:             return -1
5:         else
6:             $\boldsymbol{\pi}, v \leftarrow f_\theta(\boldsymbol{x}^{(2D)})$
7:             return $v$                         ▷ If the episode end is reached, return $v_\theta$
8:         end if
9:     end if
10:
11:    if $\boldsymbol{x} \notin Tree$ then                          ▷ Expand Leaf Node
12:        $\boldsymbol{\pi}, v \leftarrow f_\theta(\boldsymbol{x}^{(2D)})$
13:        $N(\boldsymbol{x}, \cdot) \leftarrow 0$
14:        $P(\boldsymbol{x}, \cdot) \leftarrow \boldsymbol{\pi}$
15:        return $v$
16:    end if
17:
18:    if $agent = player$ then                          ▷ Get best action using UCB
19:        $u^* \leftarrow \underset{u \in \mathcal{U}_{player}}{\operatorname{argmax}} U_{player}(\boldsymbol{x}, u)$
20:    else
21:        $u^* \leftarrow \underset{u \in \mathcal{U}_{adv}}{\operatorname{argmax}} U_{adv}(\boldsymbol{x}, u)$
22:    end if
23:    $\boldsymbol{x}, c \leftarrow \text{step}(\boldsymbol{x}, u)$                 ▷ Take next MCTS simulated step
24:    $\boldsymbol{x}^{(2D)} \leftarrow \text{getState2D}(\boldsymbol{x}, \boldsymbol{x}^{(2D)})$
25:    $agent \leftarrow \text{nextAgent}(agent)$
26:    $v \leftarrow \text{MCTSSearch}(\boldsymbol{x}^{(2D)}, agent)$              ▷ Recursion to next node
27:
28:    $Q(\boldsymbol{x}, u^*) \leftarrow \frac{N(\boldsymbol{x},u^*)Q(\boldsymbol{x},u^*)+v}{N(\boldsymbol{x},u^*)+1}$         ▷ Backup Q-values up the tree
29:    $N(\boldsymbol{x}, u^*) \leftarrow N(\boldsymbol{x}, u^*) + 1$
30:    return $v$
31: end function
```

3 Results and Discussion

3.1 Training

In this section, the training of the player and adversary will be investigated. The first sections will discuss the results of a baseline neural network, $f_\theta(\boldsymbol{x})$, which takes the true state as input. This is followed by a comparison with those of the network that took the 2-dimensional state as input, $f_\theta(\boldsymbol{x}^{(2D)})$. The results are interpreted with a discussion of the merits and shortcomings of the techniques. For the training runs shown, the parameters used are given in Table 3.1 and Section 2.3.2.

Parameter	Value
Unopposed Episodes for initial iteration	100
Opposed Episodes per iteration	40
Number of MCTS Simulations per step	20
$F_2^{(max)}$	$0.05F_1$

Table 3.1: Training parameters used

Iterations start at zero and are defined by two main steps: (1) A batch of training examples are generated using the current neural networks. (2) The neural networks are retrained using these examples. The full algorithm can be found in section 2.2.4.

3.1.1 MCTS

Figure 3.1 shows the structure of a typical tree search without neural network training. This is significantly better than with no tree search, which typically achieves an episode length of 14.

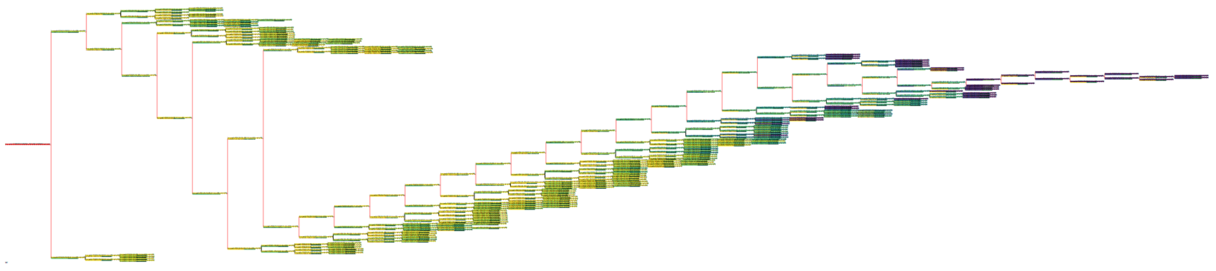


Figure 3.1: An example MCTS tree structure lasting 33 steps with 15 MCTS simulations per step and both agents are untrained. The adversary has an average power of 5% of the player ($F_2 = 0.05F_1$). Lighter (yellow) colours represent "better" states.

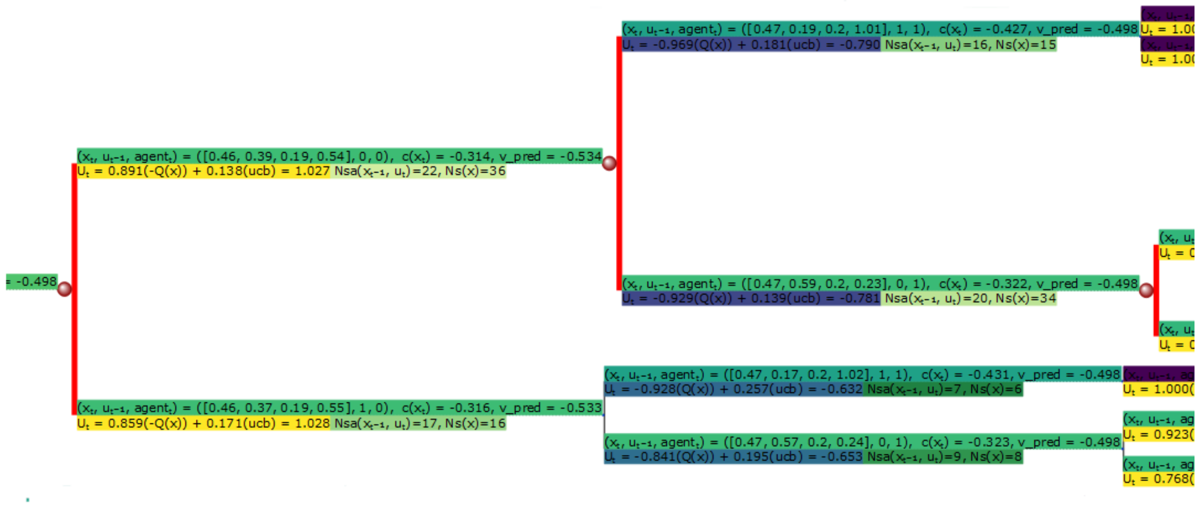

= -0.498
(x_t, u_t-1, agent_t) = ([0.46, 0.39, 0.19, 0.54], 0, 0), c(x_t) = -0.314, v_pred = -0.534
U_t = 0.891(-Q(x)) + 0.138(ucb) = 1.027 Nsa(x_t-1, u_t)=22, Ns(x)=36
(x_t, u_t-1, agent_t) = ([0.46, 0.37, 0.19, 0.55], 1, 0), c(x_t) = -0.316, v_pred = -0.533
U_t = 0.859(-Q(x)) + 0.171(ucb) = 1.028 Nsa(x_t-1, u_t)=17, Ns(x)=16
(x_t, u_t-1, agent_t) = ([0.47, 0.19, 0.2, 1.01], 1, 1), c(x_t) = -0.427, v_pred = -0.498
U_t = -0.969(Q(x)) + 0.181(ucb) = -0.790 Nsa(x_t-1, u_t)=16, Ns(x)=15
(x_t, u_t-1, agent_t) = ([0.47, 0.59, 0.2, 0.23], 0, 1), c(x_t) = -0.322, v_pred = -0.498
U_t = -0.929(Q(x)) + 0.139(ucb) = -0.781 Nsa(x_t-1, u_t)=20, Ns(x)=34
(x_t, u_t-1, agent_t) = ([0.47, 0.17, 0.2, 1.02], 1, 1), c(x_t) = -0.431, v_pred = -0.498
U_t = -0.928(Q(x)) + 0.257(ucb) = -0.632 Nsa(x_t-1, u_t)=7, Ns(x)=6
(x_t, u_t-1, agent_t) = ([0.47, 0.57, 0.2, 0.24], 0, 1), c(x_t) = -0.323, v_pred = -0.498
U_t = -0.841(Q(x)) + 0.195(ucb) = -0.653 Nsa(x_t-1, u_t)=9, Ns(x)=8
U_t = 1.00
U_t = 1.00
U_t = 1.000(
U_t = 0.923(
U_t = 0.768(

Figure 3.2: A zoomed in view of Figure 3.1. The three colours of each node are based on the value of $U = Q(\boldsymbol{x}, u) + ucb$ as in Equation (2.12), the number of state-action visits, $N(\boldsymbol{x}, u)$, and the state-cost, $c(\boldsymbol{x}_t)$. The red nodes follow the true trajectory. `Ns` ($N(\boldsymbol{x})$) is the total number of state visits, this does not equal the action-state visit count, `Nsa` ($N(\boldsymbol{x}, u)$), these differ along the true trajectory due to the re-running of the MCTS simulation.

Figure 3.2 shows the sequential maximisation that occurs during the tree search. Note that the action-value, $Q(\boldsymbol{x}_t, u_t)$, for the adversary is negated and Q is between -1 and 0 for the player. $N(\boldsymbol{x}, u)$ is correlated with the predicted value, V_{pred}, and U, which suggests that the neural network and tree search are working as expected. At the first branching in Figure 3.2 the *ucb* is 17% of U. Intuitively, this is within the correct range as a larger exploration term would cause the MCTS to branch more, diminishing the computational benefit of the structured search, whereas a smaller exploration term inhibits learning. To achieve this ratio, a value of 1 was chosen for $c_{explore}$. The action-state visit counts are 22 and 17, which are markedly higher than the number of simulations per step (10 in this example), suggesting the MCTS is performing a highly structured search. Furthermore, the similar predicted state values, $v_\theta(\boldsymbol{x})$, reflect similar visit counts of each branch.

3.1.2 State and Action Predictions

In this section, the output of the baseline neural network for a specific episode (Figures 3.3 to 3.5) is investigated. From this, the suitability of using a neural network for both the adversary and the player, what the neural networks are modelling and limitations of the model are discussed.

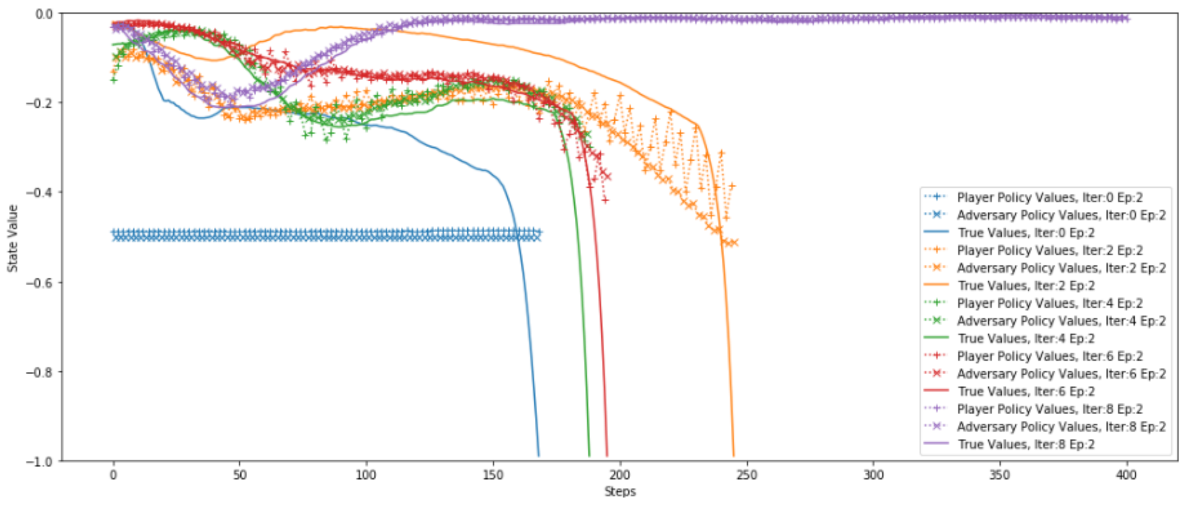


Figure 3.3: The predicted state-values (from both the adversarial and player networks) plotted against the true state-value for a number of iterations.

Figure 3.3 shows that by iteration 4 the player and adversary networks both follow the true value very closely, and by iteration 10 the mean squared error (MSE) is almost zero (Figure A.4). This is likely due to the adversary pushing only in one direction, therefore making the value very predictable. In iteration 2, the adversary values are more “jagged”, alternating between a positive and negative evaluation depending on whether the player (which roughly alternates between pushing left and right when near the equilibrium) has pushed the pendulum further over or not. I.e. by iteration 10, the adversary has learnt to predict the player’s actions, which suggests that the prediction of the state-value is not impeded by the use of two neural networks.

Figures 3.4 and 3.5 show the change in policy action predictions between iteration 2 and iteration 4. By iteration 2 the adversary has already decided on the direction that it will push, however the player is still tuning itself. At iteration 4, the player 4 switches between pushing left and right at a lower frequency.

The output of the predicted values for a small slice of $\dot{x}$ is shown in fig. 3.6. Along all pairs of axes, the data is not linearly separable. Furthermore principal component analysis

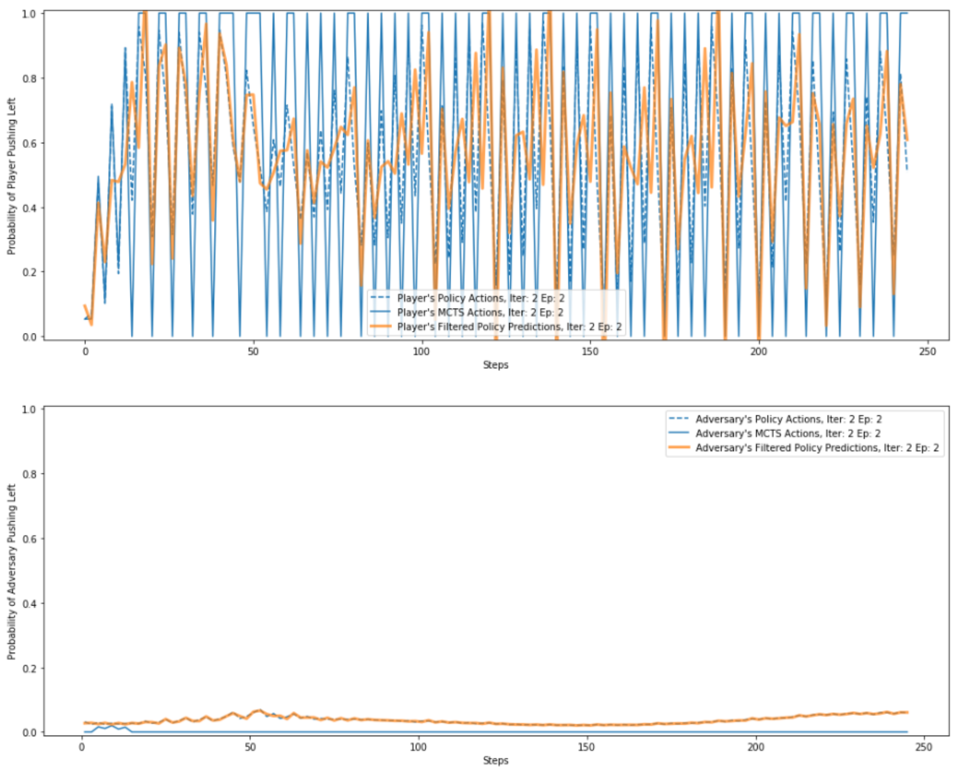


Figure 3.4: MCTS and policy predicted actions vs step for an episode in iteration 2.

(PCA) shows that the predicted values are not linearly separable along the principal axes either (fig. 3.7).

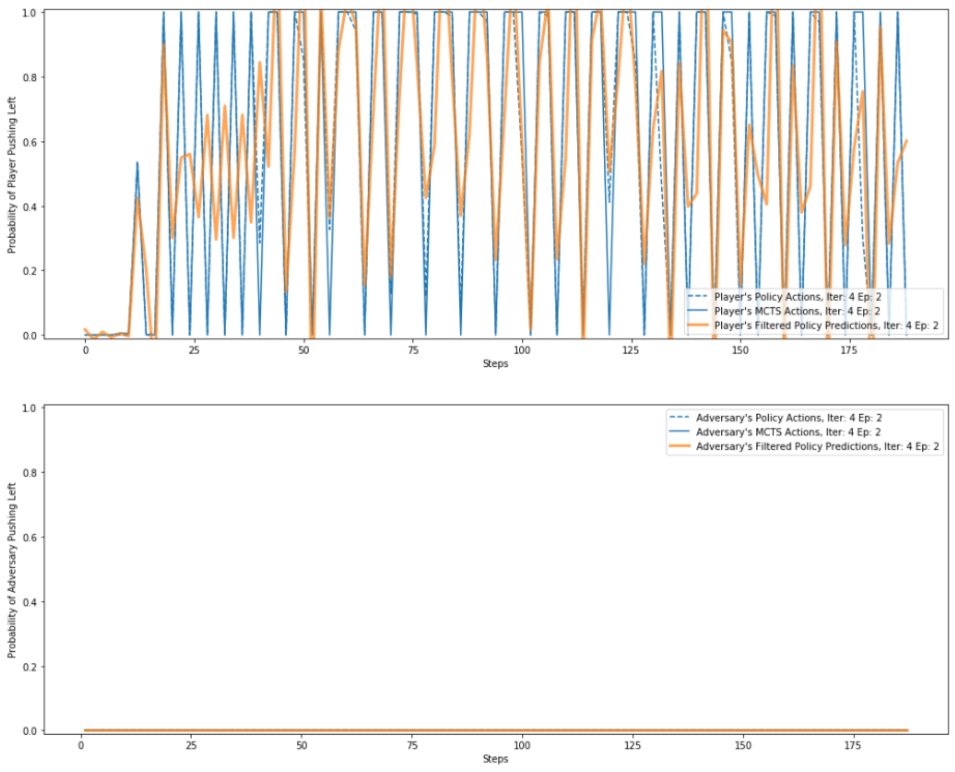


Figure 3.5: MCTS and policy predicted actions vs step for an episode in iteration 4.

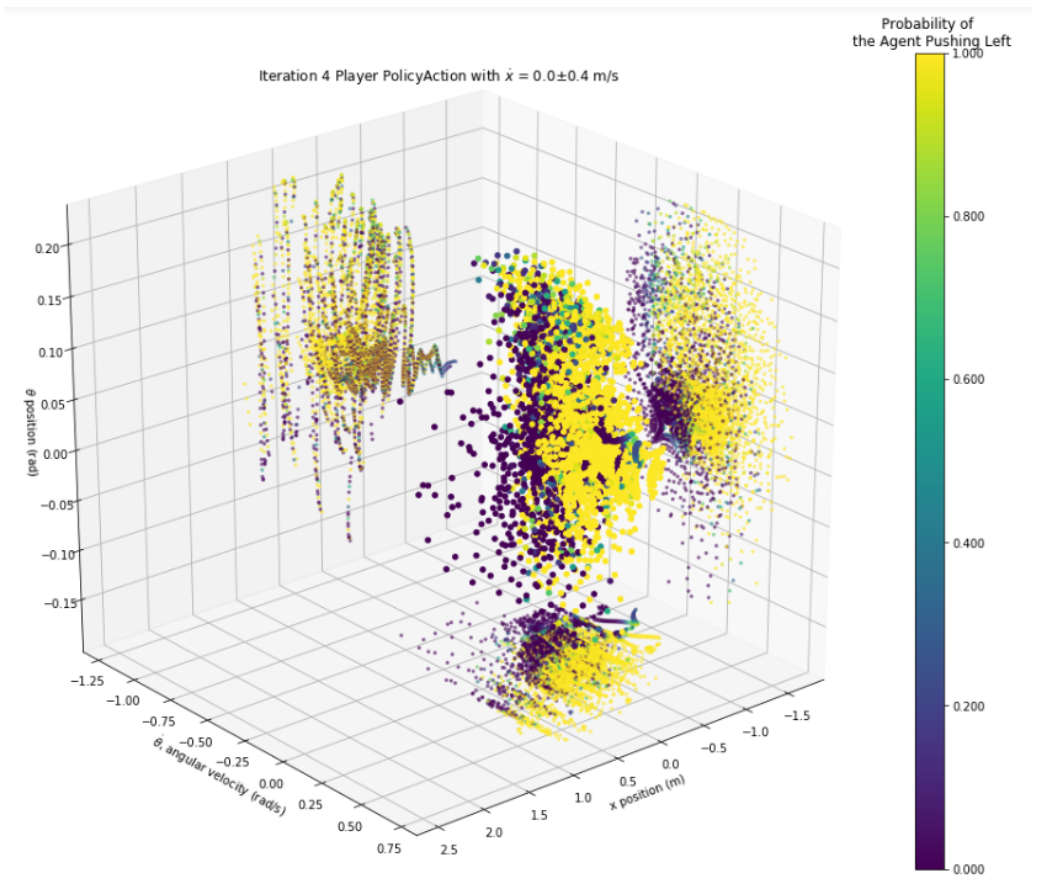


Figure 3.6: A 3D plot of policy-predicted actions during training. The central "blob" is projected onto the $(x, \dot{\theta})$, (x, θ) and $(\theta, \dot{\theta})$ axes.

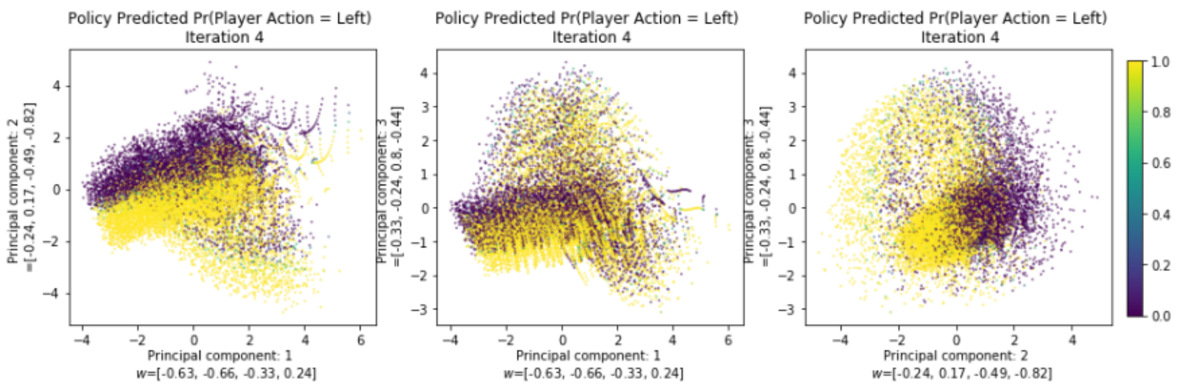


Figure 3.7: Principal Component Analysis of the data in fig. 3.6.

In the optimal control solution, a real valued force would be used to get the pendulum to the equilibrium. Due to the constraint over the actions, the policy must approximate that by repeatedly pushing left/right in order to get an average force that is equivalent to a continuous one. By iteration 4 (fig. 3.5), the neural network matches the MCTS prediction at almost every step and switches quickly between pushing left and right. This implies that a trajectory through the (x, θ) plane would look like a "wave", where the power cycle of the wave in the vicinity of the point is proportional to the instantaneous average power predicted at that point. Additionally, a higher frequency of the wave is related to a faster response time of the system. Note that higher frequencies encode information more densely, and therefore, require deeper neural networks. Portions of this wave are shown in the (x, θ) projection of fig. 3.6.

The instantaneous predicted average power of the policy can be estimated by applying a low-pass filter to the predicted action probabilities (orange line in figs. 3.4 and 3.5). The low-pass filter has a normalised cut-off frequency of $\frac{1/\tau}{2/\delta t}$, where τ is characteristic time of the system and $\frac{2}{\delta t}$ is the Nyquist frequency. For this system, however, this only provides marginal insight as, for this implementation, the time step is almost equal to the characteristic time. Decreasing the time step requires the episode length to be increase such that the real-world episode time is constant at $dt \times S = 0.01s \times 400steps = 4s$. Therefore, decreasing dt causes a significant increase computational time.

Figures 3.4 and 3.5 show that the frequency decreases between iteration 2 and iteration 4; therefore, the neural network is not making use of the highest frequency that it has the capacity to use as in iteration 2. This suggests that against this adversary, a frequency higher than the natural frequency is not necessary for good control. This may be because a higher frequency is needed for more uncertain situations and, since the adversary is only pushing in one direction, does not need to oscillate as quickly.

3.1.3 Power Matching and the Policy

In this section, the problem of "power matching" the player and adversary are discussed. Power matching involves balancing the effect of each agents actions such that they have an equal effect on the system. The efficacy of pre-training the player as a solution to this are analysed. This is followed by a discussion on why the player and adversary act as they do, with an interpretation of the policies they have learnt.

It was found that training the player unopposed indefinitely will cause the player to balance the pendulum within a few time steps from a wide range starting positions, however introducing an adversary at or close to "full power" (i.e. $F_2 \gtrapprox 0.0476F_1$, section 2.2.2) will cause the inverted pendulum to fall immediately. Therefore, an unopposed training episode and an incremental increase in adversary power were introduced. After implementing this, typically, either the adversary is too strong and the episode is over within 20 time steps, or the adversary learns to simply push in one direction only. Figure 3.8 shows this transition occuring over just one episode. Since the adversary was not trained on iteration 0, iteration 1 has MCTS predictions to push both right and left. However, during the training in iteration 2, even with approximately 10% of training examples pushing left, the network trains to push right 100% of the time, and as a result the MCTS also predicts that pushing right is always the best action.

The player does not suffer from this bias and has a far more uniform spread of policy predictions in iteration 2. In iteration 3, the policy predicts either left or right with 100% conviction, however, these are distributed in a roughly 50:50 split, with the proportion of $p(Player\ Action = left) > 0.5$ being slightly higher, to counter the adversary always pushing right. As a result of the adversary being trained to push solely in one direction, the proportion states that are visited are biased in one direction. Visually, this is as if the system is performing the set-point tracking of a reference angle slightly off centre, and therefore slowly moves in one direction until reaching x_{max}.

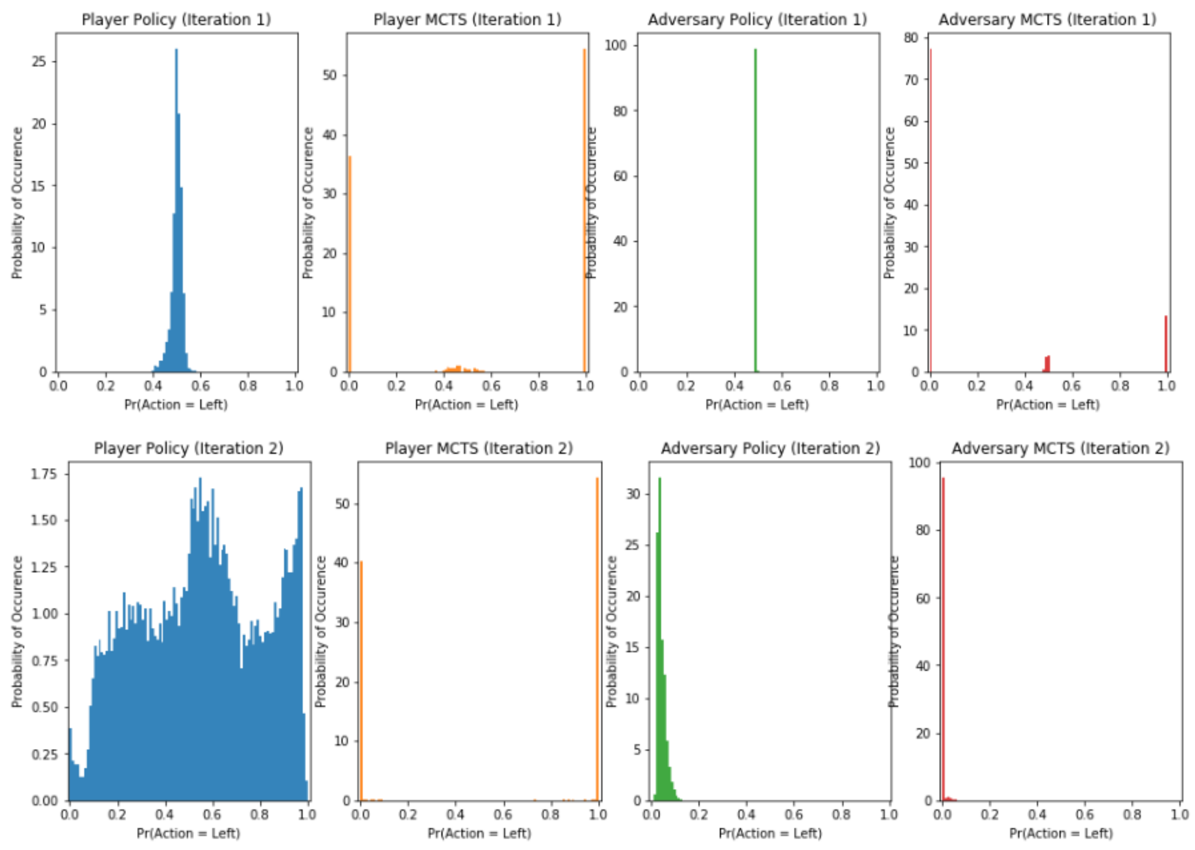


Figure 3.8: Histograms showing the distribution of predicted actions by the MCTS and pure policy for both the player and adversary.

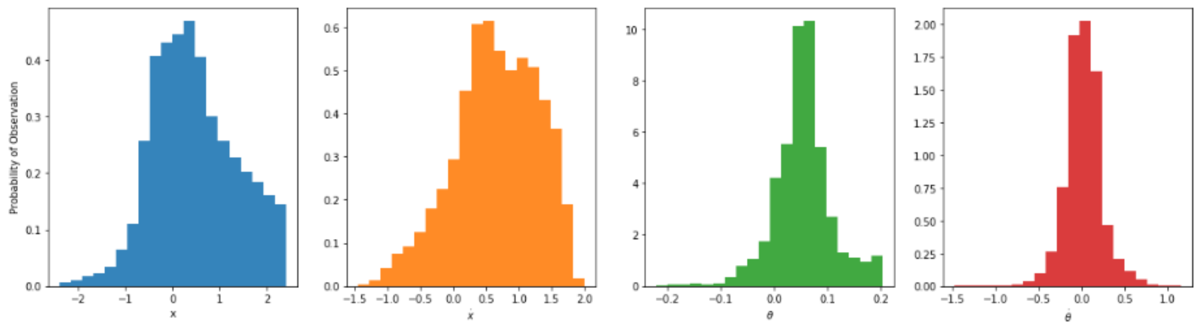


Figure 3.9: Histograms showing the proportion of time that the system was in each state on the second iteration.

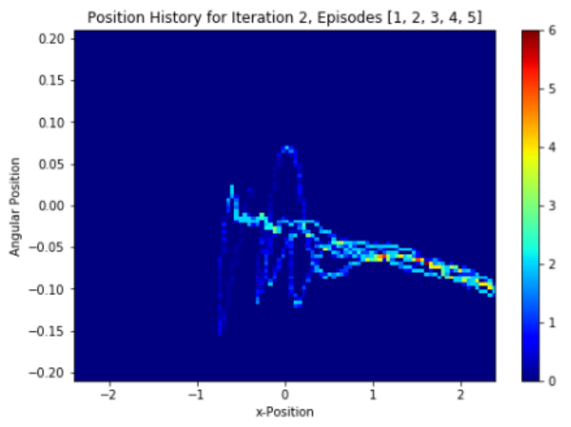
The most likely cause for this bias is the differing abilities between agents due to variations in the pre-training of the player and the power curve of the adversary. Factors such as how much to train the player before introducing an adversary and how strong to make the adversary are difficult to balance. The player can be better trained by increasing the number of training examples it has from unopposed episodes and by increasing the number of unopposed policy iterations. Additionally, the adversary's power, and power curve, can be adjusted by varying γ and F_2 in the adversary handicap formula, $F_2(1 -$

γ^{i-N+1}). However, by training the player more and under-powering the adversary, the adversary learns to push solely in one direction. Pre-training the player too little typically leads to the overtraining of the neural network on a small number of examples, which is difficult to re-train against. Alternatively, under-powering the adversary, or making it too weak for too long seems to cause the adversary to learn that slowly pushing the inverted pendulum to x_{max} is the best option, rather than attempting to push it over quickly.

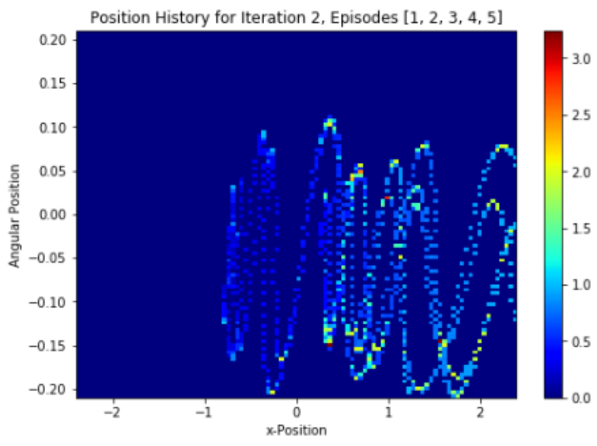
The cost function being weighted as $\boldsymbol{w}^T = [0.4, 0.1, 0.7, 1]$ may have exacerbated this behaviour. A low weight the on x-position compared to the angular position may mean that the player is not prioritising stopping the slow drift to $\boldsymbol{x}_{max}$. Yet from the adversary's perspective, since it is initially learning whilst underpowered, this is the best way to reduce the state value. A possible way in which this could have been prevented would have been to set the x-position cost to zero. In this case, if the adversary solely pushes in one direction, either the player would be able to find an equilibrium by leaning into the adversary's force - minimising the $\dot{x}$ cost, but outputting a constant θ cost; or the player would attempt to minimise the θ cost and as a result move in one direction continuously - outputting a constant $\dot{x}$ cost. Both of these scenarios do not consistently improve the adversary's state value, therefore this could force the adversary to attempt to knock the pendulum over, rather than slowly improving the state-value.

3.1.4 Comparison with $f_\theta(\boldsymbol{x}^{(2D)})$

The training of the convolutional neural network (CNN) results in training episodes with very oscillatory behaviour compared to the the baseline network (Figures 3.10a and 3.10b).



(a) True-State Input



(b) 2-Dimensional State Input

Figure 3.10: Motion history images showing the trajectories of 5 episodes for both neural networks. Higher valued colours represent more recent positions (or positions with multiple visits as the images are computed as a sum of discounted states).

The oscillatory nature of these episodes are due to the player switching between pushing left and right much slower than the characteristic time of the system (fig. 3.11).

This is likely due to the a relatively shallow neural network being used (2 convolutional layers followed by 1 linear layer. See section 2.3.2). A deeper neural network was found to exceed the memory allocation limit of the GPU used. It is possible that this caused a more simple model of the controller to be trained, such that the data became easily linearly separable, as shown in fig. 3.12 where the (x, θ) and $(\theta, \dot{\theta})$ axes have easily separable predictions.

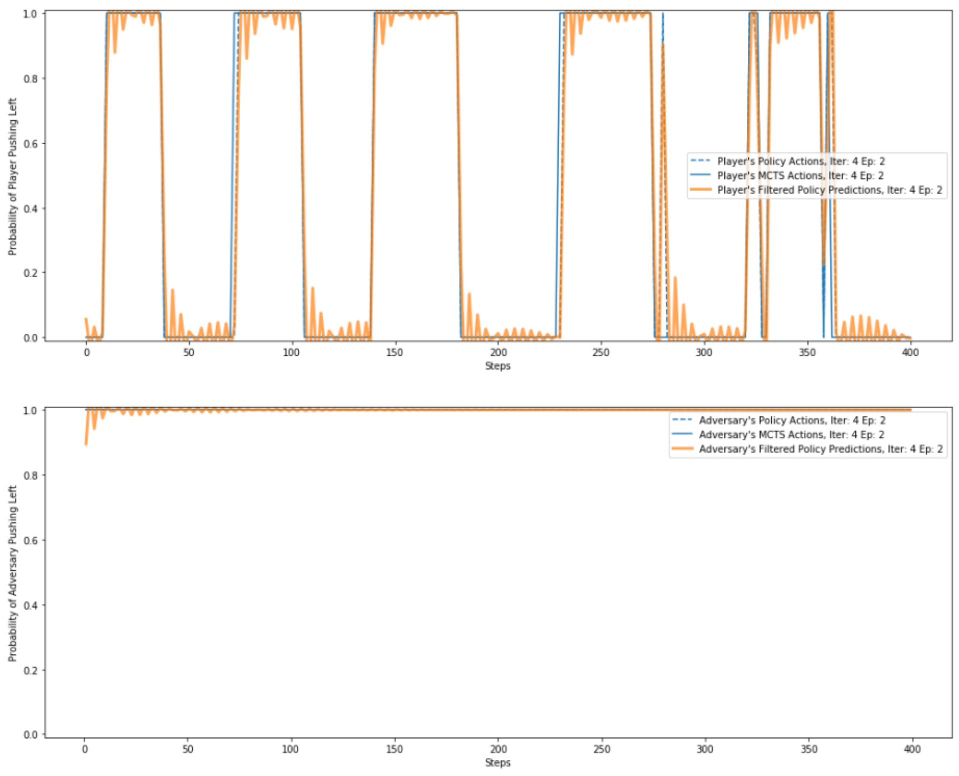


Figure 3.11: MCTS and policy predicted actions vs step for an episode in iteration 4.

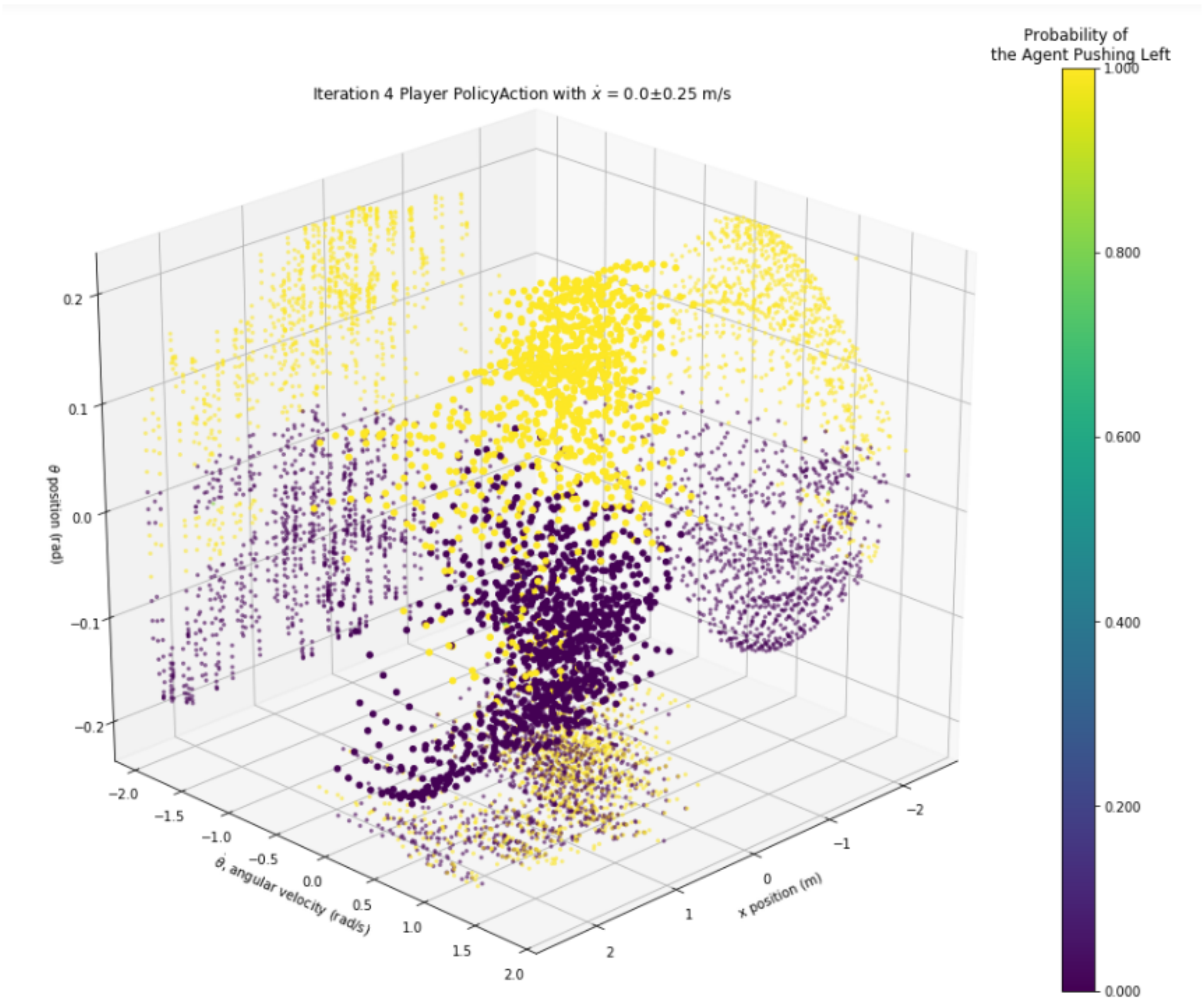


Figure 3.12: A 3D plot of policy-predicted actions after the 4th iteration of training of the CNN.

As the training iterations increased, the oscillations became larger, until iteration 9 where they became large enough that the controller could not stabilise the pendulum at all.

Another possibility for the poor training of the CNN is the relatively small number of training examples used. Each iteration generates less than 16,000 examples (40 episodes each with 400 steps), and at each iteration the last 5 iteration's examples are used for training, which gives a maximum of 80,000 training examples - or 40,000 for each agent. With 40 bins there are 40x40=1600 possible positions. The number of possible 2D-states increases as 1600^n, where n is the number time steps recorded, therefore, even after 2 time steps, there are 64 times more possible 2D-states than training examples.

3.2 Evaluation and Testing

In this section, the results of experiments pitting the player against a number of adversaries is discussed and the performance and robustness of the algorithm is evaluated.

3.2.1 Performance Against a Random Opponent

Against a random adversary with $F_2 = 0.05F_1$, the player's performance improves with the number of MCTS simulations when it has not been trained. However, the performance after training is constant, as shown in fig. 3.14.

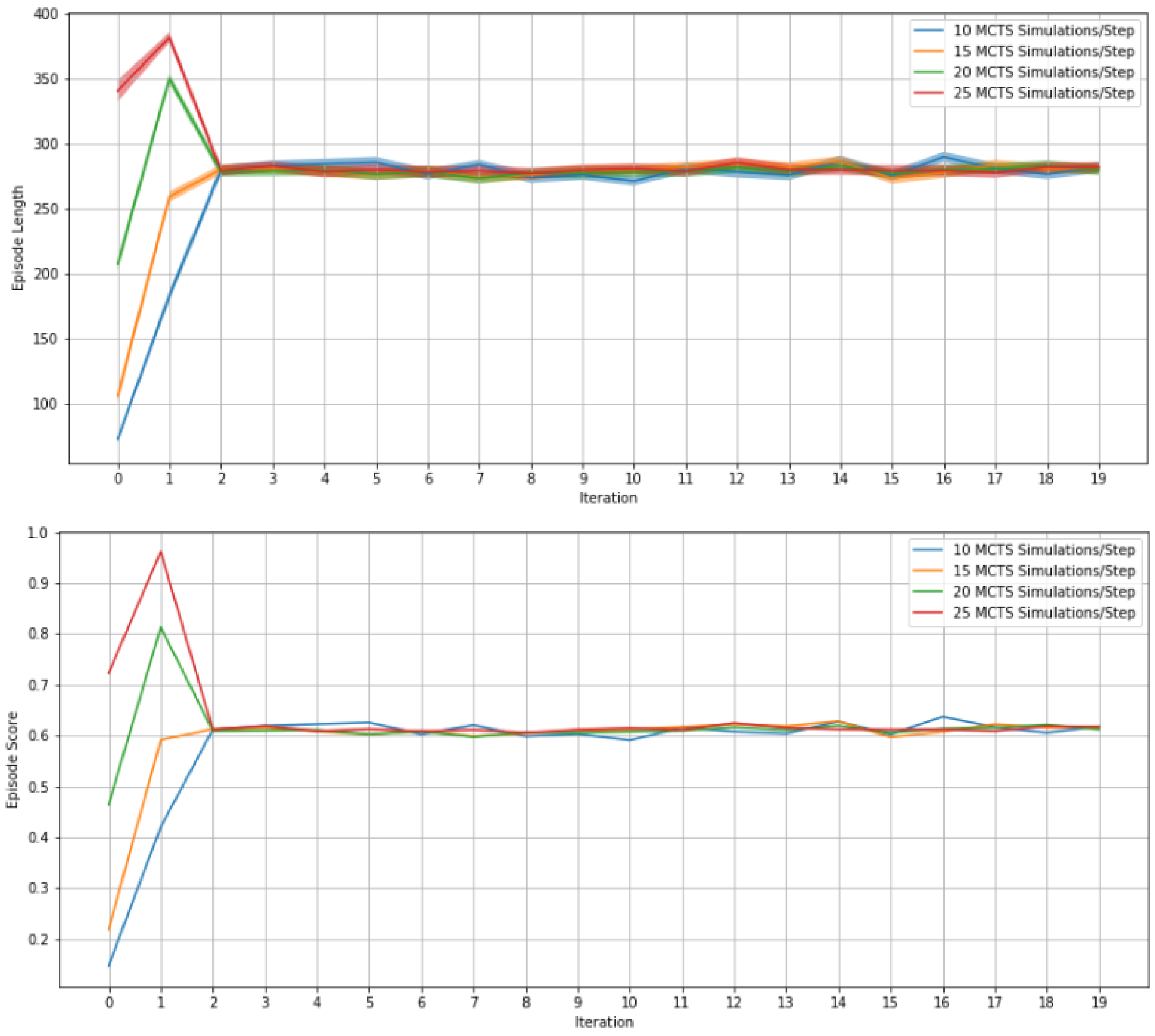


Figure 3.13: The average episode length and score for each iteration over 50 episodes. The standard deviation of episode lengths is also shown as a lighter region around the means. A random adversary is used with $F_2 = 0.05F_1$.

Figure 3.14 shows the first iteration's score and length increase with the number of MCTS simulations per step, as the neural network is trained without an adversary.

However, since the player has a model of what it expects the "best" adversary to do incorporated into its MCTS, if the player expects the adversary to push in only one direction then it will perform poorly against an adversary that does not follow that policy.

The ucb (exploration) term in eq. (2.12) is only $\sim 20\%$ of the Q value when the player is untrained (section 3.1.1). If the player is trained such that its model adversary is predicting $p(left) = 0.01$ then the magnitude of ucb_{left} would need to be $\frac{0.2}{0.01} = 20$ times greater than ucb_{right} to get an exploration term of a similar magnitude. For the number of MCTS simulations per step used in testing, the exploration term never becomes large enough to match the size of the exploitation (Q) term. Therefore, the player acts greedily with respect to its model agent's policy predictions. Due to this, the MCTS trees from iteration 2 onwards do not search along trajectories in which the agent gives a small action probability i.e. the tree search never expands left if the adversary always predicts right. This means that if the random adversary picks left then the MCTS must effectively reset the tree, and furthermore, the value predicted for that branch is not representative of the branch as a whole. Therefore, the performance of the player is defined by the average length of sequential right pushes rather than the depth of the MCTS. This is why the mean score and episode lengths are roughly equal after the adversary is trained and is a major weakness for the players trained against an adversary.

Note that the episode length and score are highly correlated. This is to be expected as the player acts in a similar way for each episode. However, the score cannot necessarily be predicted from the episode length, for example, if an agent that allows oscillatory behaviour was implemented, such as the CNN, $f(\boldsymbol{x}^{(\boldsymbol{2D})})$, it's score will be low, but the average length of an episode will not necessarily be low.

3.2.2 Performance Against Variable F_2 and Trained Adversaries

When the player is pitted against a trained adversary, either the adversary pushes in the same direction as the player's model of the adversary, or it does not. In the former case, the player performs very well lasting to 350 steps on average, whereas in the latter case it performs particularly badly, often falling in less than 20 steps.

When the trained player is pitted against random adversaries with different strengths, the episode length follows a predictable pattern of falling over faster when there is a higher force applied by the adversary. Note, that in order for the player to be able to control for adversaries with varying forces, the player's model of the adversary must also apply that force, otherwise the tree search would not be able to look ahead. This is one of the main drawbacks of this method; there must be perfect knowledge of the adversary and the system must be deterministic.

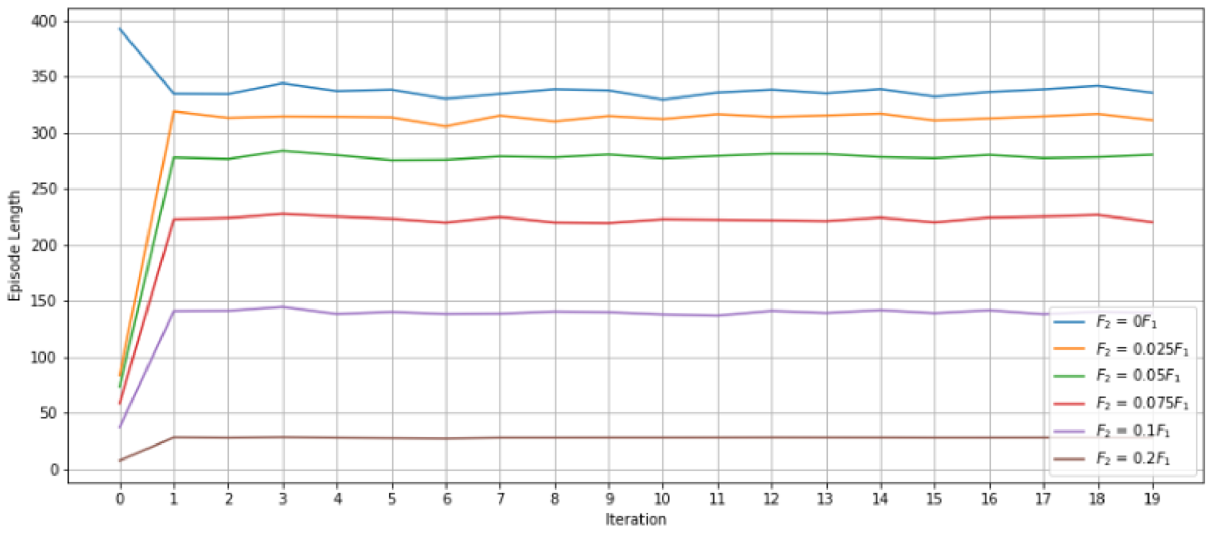


Figure 3.14: The average episode length for each iteration over 50 episodes. A selection of adversary forces, F_2, were used. 15 MCTS simulations per step was used for every run.

4 Conclusions And Future Work

4.1 Conclusions

The aim of this project was to investigate the application of the AlphaZero algorithm to control problems, specifically the control of dynamical systems. The inverted pendulum was chosen as a simple starting dynamical system which also met the requirements of being sufficiently complex such that a trivial solution was not possible.

In order to adapt AlphaZero into a control algorithm a state cost function and a type of receding horizon for the state value were designed. The state value is calculated retrospectively and used for training the neural network. Adding these allowed for the algorithm to deal with continuous states and indefinite horizons with no easily discernable "winner", unlike the board games played by AlphaZero. In addition to this, two different methods of state representation were considered: the true state, $\boldsymbol{x}^T = [x,\ \dot{x},\ \theta,\ \dot{\theta}]$, and a 2-dimensional (2D) state, $\boldsymbol{x}^{(2D)}$. The true state had the benefit of being continuous with a low number of dimensions, which made it more ideal for this specific problem. However, the 2D state did not rely on measurements of the system velocities. Relying purely on positional information allows for a more generalisable state space representation as in real systems, states are often unobservable. Furthermore, the binning of the states was proven to be lossless in the limit of infinite time, assuming a deterministic model and that the neural network could emulate an optimal Kalman Filter.

The policy was represented as two neural networks, paired at each iteration. It was found that the two networks predicted similar values for each state suggesting that the use of a network for each the player and adversary did not hinder learning.

Balancing the effect of each agents actions such that they have an equal effect on the system, or "power matching" was one of the more difficult problems faced. The adversary was chosen to act at the tip of the pendulum so that the adversary would be forced to learn a different policy than the player. However, due to the effect of gravity and a different point of action, the adversary and player had unequal effects on the system. It was found that simply choosing the force of the adversary to cause the same angular acceleration of the pendulum ($F_2 = 0.05F_1$) would lead to the pendulum falling very quickly, meaning that minimal training examples were recorded; or, if a smaller force was chosen, the adversary was too weak to push over the pendulum. In order to counter this, the adversary force was incrementally increased. In the case of the adversary being too weak, the adversary invariably learnt to push solely in one direction, it was also found that pre-training did not improve this. This was postulated to be the most efficient way for the adversary to maximise the cost of the system. Due to the adversary learning this, robustness to general disturbances was severely hindered. It was found that the player's internal model of the adversary could not deal with more general adversaries.

Against a random adversary, the performance of the player was found to be independent of the depth of the Monte-Carlo tree search, and was defined by the average length of sequential right pushes. This was due to the resetting of the search when the adversary pushed in a direction unexpected by the player.

Due to computational limitations, the episode length and time step were limited such that the time step was only marginally faster than the characteristic time of the system. In an ideal simulation, the time step would be many times smaller than the characteristic time, allowing for the controller to approximate continuous actions (pulse width modulation). It was found, even with the limited depth of the neural networks used, that the policies could model rapid changes in output with respect to the input space well. This suggests that there is potential for this method to work well on many problems.

The potential benefits of successfully adapting AlphaZero were: robust disturbance handling, the ability to control non-linear systems, and a wide range of possible applications. It has been shown that MCTS can be used to improve the control of a system to great effect, and there is no limitation for this due to system non-linearities. This method can model any system given that it is deterministic and the player has a perfect knowledge of the action space of the adversary. However, more work is needed for this to be considered "robust control".

4.2 Recommended Future Work

This project has shown that the application of AlphaZero to control problems is feasible, however, there are a number of areas in which further work needs to be done.

The simplest improvement for this project is to use more computing power. The neural networks and number of training examples in this project were limited primarily due to the specifications of the GPU used. Therefore, a more powerful computer would allow data to be collected more quickly, and deeper neural networks to be designed.

The most pressing area of interest is the problem of "power matching". In the discussion, it was postulated that the reason for the adversary learning to push solely in one direction was due to the choice of cost function. It has been shown in other control schemes for the inverted pendulum that the adversary should be able to push the pendulum over without overpowering the player [9]. This can be done by pushing at the system's resonant frequency, which takes advantage of inherent lag in the system. Therefore, a simple improvement may be to introduce a first order lag, $F_t = \lambda F_{k-1} + (1 - \lambda)u_k$.

Recently, Deepmind released a blog post on a new algorithm, *AlphaStar*, which defeated the world champion in StarCraft II [17]. Due to the complexity of StarCraft, the algorithm was designed to keep a record of different adversarial strategies, and was deemed “good” if it could defeat all of them. It is speculated that there are multiple strategies to push over the IP, one being to slowly push it to one side, and another being to take advantage of mistakes made by the controller (player). Making a record of multiple adversaries may provide a method to overcome this.

5 References

[1] M.C. Smith, I Lestas, *4F2: Robust and Non-Linear Control* Cambridge University Engineering Department, 2019

[2] G. Vinnicombe, K. Glover, F. Forni, *4F3: Optimal and Predictive Control* Cambridge University Engineering Department, 2019

[3] S. Singh, *4F7: Statistical Signal Analysis* Cambridge University Engineering Department, 2019

[4] Arthur E. Bryson Jr, *Optimal Control - 1950 to 1985.* IEEE Control Systems, 0272-1708/95 pg.26-33, 1996.

[5] I. Michael Ross, Ronald J. Proulx, and Mark Karpenko, *Unscented Optimal Control for Space Flight.* ISSFD S12-5, 2014.

[6] Zheng Jie Wang, Shijun Guo, Wei Li, *Modeling,Simulation and Optimal Control for an Aircraft of Aileron-less Folding Wing* WSEAS TRANSACTIONS on SYSTEMS and CONTROL, ISSN: 1991-8763, 10:3, 2008

[7] Giovanni Binet, Rainer Krenn and Alberto Bemporad, *Model Predictive Control Applications for Planetary Rovers.* imtlucca, 2012.

[8] Raković, Saša, *"Robust Model-Predictive Control.* Encyclopedia of Systems and Control, pg.1-11 ,2013.

[9] Russ Tedrake, *Underactuated Robotics.* MIT OpenCourseWare, Ch.3, Spring 2009.

[10] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction (2nd Edition).* The MIT Press, Cambridge, Massachusetts, London, England. 2018.

[11] I. Carlucho, M. De Paula, S. Villar, G. Acosta . *Incremental Q-learning strategy for adaptive PID control of mobile robots.* Expert Systems with Applications. 80. 10.1016, 2017

[12] Yuxi Li, *Deep Reinforcement Learning: An Overview.* CoRR, abs/1810.06339, 2018.

[13] Sandy H. Huang, Martina Zambelli, Jackie Kay, Murilo F. Martins, Yuval Tassa, Patrick M. Pilarski, Raia Hadsell, *Learning Gentle Object Manipulation with Curiosity-Driven Deep Reinforcement Learning.* arXiv 2019.

[14] David Silver, Julian Schrittwieser, Karen Simonyan et al, *Mastering the game of Go without human knowledge.* Nature, vol. 550, pg.354–359, 2017.

[15] David Silver, Thomas Hubert, Julian Schrittwieser et al, *A general reinforcement learning algorithm that masters chess, shogi and Go through self-pla.* Science 362:6419, pg.1140-1144, 2018.

[16] S. Thakoor, S. Nair and M. Jhunjhunwala, *Learning to Play Othello Without Human Knowledge* Stanford University Press, 2018 `https://github.com/suragnair/alpha-zero-general`

[17] Google Deepmind, *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II.* `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game -starcraft-ii/`

A Appendices

A.1 Inverted Pendulum Dynamics Derivation

The state space equations for the Inverted Pendulum can be found using d'Alembert forces. Define the distance and velocity vectors to the important points:

$$\begin{aligned}
&\boldsymbol{r}_P = x\boldsymbol{i} \\
&\boldsymbol{r}_{B_1/P} = Lsin\theta\boldsymbol{i} + Lcos\theta\boldsymbol{j} \\
&\boldsymbol{r}_{B_1} = (x + Lcos\theta)\boldsymbol{i} + L\dot{\theta}sin\theta\boldsymbol{j} \\
&\dot{\boldsymbol{r}}_{B_1} = (\dot{x} + L\dot{\theta}cos\theta)\boldsymbol{i} - L\dot{\theta}sin\theta\boldsymbol{j}
\end{aligned}$$

Linear Momentum, $\boldsymbol{\rho} = \sum_i m_i \dot{\boldsymbol{r}}_{i/o} = m\dot{\boldsymbol{r}}_{B_1} + M\dot{\boldsymbol{r}}_P$:

$$\boldsymbol{\rho} = \begin{bmatrix} (M+m)\dot{x} + mL\dot{\theta}cos\theta \\ -mL\dot{\theta}sin\theta \\ 0 \end{bmatrix}$$

Moment of momentum about P, $\boldsymbol{h}_P = \boldsymbol{r}_{B_1/P} \times m\dot{\boldsymbol{r}}_{B_1}$:

$$\begin{aligned}
\boldsymbol{h}_P &= -mL(L\dot{\theta} + \dot{x}cos\theta)\boldsymbol{k} \\
\therefore \dot{\boldsymbol{h}}_P &= -mL(L\ddot{\theta} + \ddot{x}cos\theta - \dot{x}\dot{\theta}sin\theta)\boldsymbol{k}
\end{aligned}$$

Balance moments using $\dot{\boldsymbol{h}}_P + \dot{\boldsymbol{r}}_P \times \boldsymbol{\rho} = \boldsymbol{Q}_e$ and $\boldsymbol{Q}_e = \boldsymbol{r}_{B_1/P} \times -mg\boldsymbol{j} + \boldsymbol{r}_{B_2/P} \times F_2\boldsymbol{i}$:

$$\dot{\boldsymbol{h}}_P + \dot{\boldsymbol{r}}_P \times \boldsymbol{\rho} = \begin{bmatrix} 0 \\ 0 \\ -mL(\ddot{x}cos\theta + L\ddot{\theta}) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -L(mgsin\theta + 2F_2cos\theta) \end{bmatrix} = \boldsymbol{Q}_e$$

And also balance linear momentum using $\boldsymbol{F}_e = \dot{\boldsymbol{\rho}}$:

$$\dot{\boldsymbol{\rho}} = \begin{bmatrix} (m+M)\ddot{x} + mL(\ddot{\theta}cos\theta - \dot{\theta}^2 sin\theta) \\ -mL(\ddot{\theta}sin\theta + \dot{\theta}^2cos\theta) \\ 0 \end{bmatrix} = \begin{bmatrix} F_1 + F_2 \\ R - (M+m)g \\ 0 \end{bmatrix} = \boldsymbol{F}_e$$

Finally write the system dynamics in terms of $\ddot{\theta}$ and $\ddot{x}$:

$$\ddot{\theta}\big(M + msin^2\theta\big)L = \left(\frac{2M+m}{m}F_2 - F_1\right)cos\theta + g(M+m)sin\theta - mL\dot{\theta}^2 sin\theta cos\theta \quad (A.1)$$

$$\ddot{x}(M + msin^2\theta) = F_1 - F_2 cos(2\theta) + msin\theta(L\dot{\theta}^2 - gcos\theta) \quad (A.2)$$

Simplifying by substituting in constants, the full state space equation are:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \frac{\left(\frac{2M+m}{m}F_2 - F_1\right)cos\theta + g(M+m)sin\theta - mL\dot{\theta}^2 sin\theta cos\theta}{(M+msin^2\theta)} \\ \dot{\theta} \\ \frac{F_1 - F_2cos(2\theta) + msin\theta(L\dot{\theta}^2 - gcos\theta)}{L(M+msin^2\theta)} \end{bmatrix} = \begin{bmatrix} f_1(\boldsymbol{x}, F_1, F_2) \\ f_2(\boldsymbol{x}, F_1, F_2) \\ f_3(\boldsymbol{x}, F_1, F_2) \\ f_4(\boldsymbol{x}, F_1, F_2) \end{bmatrix} \quad (A.3)$$

Using Lyapunov's indirect method, the linearised equations about the equilibrium, $\boldsymbol{x}_e = [x_e, \dot{x}_e, \theta_e, \dot{\theta}_e]^T = [0, 0, 0, 0]^T$, are:

$$\begin{bmatrix} \delta\dot{x} \\ \delta\ddot{x} \\ \delta\dot{\theta} \\ \delta\ddot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_1}{\partial \dot{x}}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_1}{\partial \theta}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_1}{\partial \dot{\theta}}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_2}{\partial x}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_2}{\partial \dot{x}}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_2}{\partial \theta}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_2}{\partial \dot{\theta}}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_3}{\partial x}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_3}{\partial \dot{x}}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_3}{\partial \theta}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_3}{\partial \dot{\theta}}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_4}{\partial x}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_4}{\partial \dot{x}}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_4}{\partial \theta}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_4}{\partial \dot{\theta}}\Big|_{\boldsymbol{x}_e} \end{bmatrix} \begin{bmatrix} \delta x \\ \delta\dot{x} \\ \delta\theta \\ \delta\dot{\theta} \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial F_1}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_1}{\partial F_2}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_2}{\partial F_1}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_2}{\partial F_2}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_3}{\partial F_1}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_3}{\partial F_2}\Big|_{\boldsymbol{x}_e} \\ \frac{\partial f_4}{\partial F_1}\Big|_{\boldsymbol{x}_e} & \frac{\partial f_4}{\partial F_2}\Big|_{\boldsymbol{x}_e} \end{bmatrix} \begin{bmatrix} \delta F_1 \\ \delta F_2 \end{bmatrix} \quad (A.4)$$

$$\begin{bmatrix} \delta\dot{x} \\ \delta\ddot{x} \\ \delta\dot{\theta} \\ \delta\ddot{\theta} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{mg}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{(m+M)}{ML}g & 0 \end{bmatrix} \begin{bmatrix} \delta x \\ \delta\dot{x} \\ \delta\theta \\ \delta\dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{M} & -\frac{1}{M} \\ 0 & 0 \\ -\frac{1}{ML} & \frac{2M+m}{mML} \end{bmatrix} \begin{bmatrix} \delta F_1 \\ \delta F_2 \end{bmatrix} \quad (A.5)$$

The eigenvalues are given by $det(\lambda I - A) = \lambda^2(\lambda^2 - \frac{(m+M)}{ML}g) = 0$. Therefore, the system is unstable about $\boldsymbol{x}_e$ due to the right half plane pole, $\lambda = \sqrt{\frac{(m+M)}{ML}g}$. Additionally, the time constant of this unstable system is $\tau = \sqrt{\frac{ML}{g(m+M)}}$. Note, if $M >> m, \tau \rightarrow \sqrt{\frac{L}{g}}$, which is the time constant for a simple pendulum.

It can be proved that the inverted pendulum system is controllable by showing:

$$rank[\boldsymbol{B}\ \boldsymbol{AB}\ \boldsymbol{A^2B}\ \boldsymbol{A^3B}] = 4 \quad (A.6)$$

Therefore for any initial condition we can reach $\boldsymbol{x}_e$ in finite time under these linear assumptions.

A.2 Propagation of Quantisation Error

The state space model for the quantisation of the linearised inverted pendulum can be written as:

$$\boldsymbol{x}_t^{(2D)} = C\boldsymbol{x}_t + \boldsymbol{V}_t \qquad \boldsymbol{V}_t \sim \mathcal{U}\left(\begin{bmatrix} \frac{1}{\delta x} \\ \frac{1}{\delta\theta} \end{bmatrix}\right) \tag{A.7}$$

$$\boldsymbol{x}_t = A\boldsymbol{x}_{t-1} + B\boldsymbol{u}_t \tag{A.8}$$

Where A and B are the linearised system dynamics (valid for small time steps), and C is the linear transformation to a 2D state space, with quantisation noise $\mathbf{V}$.

Assuming the quantisation bin sizes, δx and $\delta\theta$, are small and that x and θ are independent within the bin, the quantisation noise can be modelled as uniform random variables with covariance, $cov(\boldsymbol{V},\boldsymbol{V}) = \mathbb{E}[\boldsymbol{V}\boldsymbol{V}^T]$:

$$= \mathbb{E}\begin{bmatrix} x^2 & x\theta \\ \theta x & \theta^2 \end{bmatrix} = \begin{bmatrix} \int_{-\delta x/2}^{\delta x/2} x^2 \cdot \frac{1}{\delta x} dx & 0 \\ 0 & \int_{-\delta\theta/2}^{\delta\theta/2} \theta^2 \cdot \frac{1}{\delta\theta} d\theta \end{bmatrix} = \begin{bmatrix} \frac{\delta x^2}{12} & 0 \\ 0 & \frac{\delta\theta^2}{12} \end{bmatrix} \tag{A.9}$$

For simplicity, let $\delta x = \delta\theta$, and therefore, $cov(\boldsymbol{V},\boldsymbol{V}) = \sigma_v^2 I$.

Kalman filtering can be used to find an optimal estimate $\hat{\boldsymbol{x}}_n = K[\boldsymbol{x}_n|\boldsymbol{y}_{n-k:n}]$ using the algorithm (derived in [3]).

Algorithm 6 Multivariate Kalman Filtering

```
1: Given: $\hat{\boldsymbol{x}}_n = K[\boldsymbol{x}_n|\boldsymbol{y}_{1:n}]$ and $\Sigma_n = \mathbb{E}[(\boldsymbol{x}_n - \hat{\boldsymbol{x}}_n)(\boldsymbol{x}_n - \hat{\boldsymbol{x}}_n)^T]$

   Prediction:
2: $\bar{\boldsymbol{x}}_{n+1} = K[\boldsymbol{x}_n|\boldsymbol{y}_{1:n}] = A\hat{\boldsymbol{x}}_n + B\boldsymbol{u}_{n+1}$
3: $\bar{\Sigma}_{n+1} = \mathbb{E}[(\boldsymbol{x}_{n+1} - \bar{\boldsymbol{x}}_{n+1})(\boldsymbol{x}_{n+1} - \bar{\boldsymbol{x}}_{n+1})^T] = A\Sigma_n A^T + \Sigma_w$    ▷ $\Sigma_w = \mathbf{0}$

   Update:
4: $\tilde{\boldsymbol{y}}_{n+1} = \boldsymbol{y}_{n+1} - C\bar{\boldsymbol{x}}_{n+1}$    ▷ Calculate Innovation residual
5: $S_{n+1} = \Sigma_v + C\bar{\Sigma}_{n+1}C^T$    ▷ Calculate Innovation Covariance
6: $\hat{\boldsymbol{x}}_{n+1} = \bar{\boldsymbol{x}}_{n+1} + \bar{\Sigma}_{n+1}C^T S_{n+1}^{-1}\tilde{\boldsymbol{y}}_{n+1}$    ▷ $\Sigma_v = \sigma_v^2 I$
7: $\Sigma_{n+1} = (I - \bar{\Sigma}_{n+1}C^T S_{n+1}^{-1}C)\bar{\Sigma}_{n+1}$
```

Thus we can find the optimal linear estimate of the covariance of Equation (A.7) from algorithm 6 line 7:

$$\Sigma_{n+1} = (I - \bar{\Sigma}_{n+1}C^T S_{n+1}^{-1}C)\bar{\Sigma}_{n+1} \tag{A.10}$$

$$= (I - \bar{\Sigma}_{n+1}C^T(\sigma_v^2 I + C\bar{\Sigma}_{n+1}C^T)^{-1}C)\bar{\Sigma}_{n+1}, \qquad where\ \bar{\Sigma}_{n+1} = A\Sigma_n A^T \tag{A.11}$$

For the univariate case this reduces to:

$$\sigma_{n+1}^2 = \bar{\sigma}_{n+1}^2 (1 - \frac{C^2 \bar{\sigma}_{n+1}^2}{\sigma_v^2 + C^2 \bar{\sigma}_{n+1}^2}) \tag{A.12}$$

$$= A^2 \sigma_n^2 (1 - \frac{C^2 A^2 \sigma_n^2}{\sigma_v^2 + C^2 A^2 \sigma_n^2}) \tag{A.13}$$

It is obvious from this that as $\sigma_v^2 \to 0$, $\sigma_{n+1}^2 \to 0$. Furthermore, if the spectral radius, $\rho\left(A^2(1 - \frac{C^2 A^2 \sigma_n^2}{\sigma_v^2 + C^2 A^2 \sigma_n^2})\right) < 1$, then the mean squared error from quantisation will reduce to zero. Since $0 < ||\frac{C^2 A^2 \sigma_n^2}{\sigma_v^2 + C^2 A^2 \sigma_n^2}||_2^2 < 1$ if $\sigma_v^2 > 0$, a sufficient condition is that $A^2 \leq 1 \implies \rho(A) \leq 1$.

Using Gelfand's Theorem, $\rho(M_1 ... M_n) \leq \rho(M_1) ... \rho(M_n)$, and the eigenvalue identity, $(M + cI)v = (c + \lambda)v$. The MSE of the multivariate case can be reduced to:

$$1 > \rho\left((I - \bar{\Sigma}_{n+1} C^T (\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)^{-1} C) \bar{\Sigma}_{n+1}\right) \tag{A.14}$$

$$> \rho(\bar{\Sigma}_{n+1}) \left[1 - \rho(\bar{\Sigma}_{n+1} C^T (\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)^{-1} C)\right] \tag{A.15}$$

Therefore, if $\rho\left(\bar{\Sigma}_{n+1} C^T (\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)^{-1} C)\right)$ is less than one, and $\rho(\bar{\Sigma}_{n+1}) < 1$, then this is true. This is equivalent to showing that if $0 < \kappa(\Sigma_n) \kappa(A)^2 \kappa(CC^T) \leq 1$ and $\rho(AA^T) < 1$, where $\kappa(\cdot)$ denotes the condition number, then the MSE will decay:

$$\begin{aligned}
\rho\left(\bar{\Sigma}_{n+1} C^T (\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)^{-1} C)\right) &\leq \rho(\bar{\Sigma}_{n+1}) \rho(CC^T) \rho((\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)^{-1}) \\
&\leq \rho(\bar{\Sigma}_{n+1}) \rho(CC^T) \frac{1}{|\lambda_{min}(\sigma_v^2 I + C \bar{\Sigma}_{n+1} C^T)|} \\
&= \rho(\bar{\Sigma}_{n+1}) \rho(CC^T) \frac{1}{|\lambda_{min}(C \bar{\Sigma}_{n+1} C^T)| + \sigma_v^2} \\
&< \frac{|\lambda_{max}(\bar{\Sigma}_{n+1}) \lambda_{max}(CC^T)|}{|\lambda_{min}(C \bar{\Sigma}_{n+1} C^T)|} \\
&< \frac{|\lambda_{max}(\bar{\Sigma}_{n+1}) \lambda_{max}(CC^T)|}{|\lambda_{min}(\bar{\Sigma}_{n+1} CC^T)|} \\
&< \frac{|\lambda_{max}(\bar{\Sigma}_{n+1}) \lambda_{max}(CC^T)|}{|\lambda_{min}(\bar{\Sigma}_{n+1}) \lambda_{min}(CC^T)|} \\
&= |\kappa(\Sigma_n)| \kappa(A)^2 \kappa(CC^T)
\end{aligned}$$

where the above uses the results $\lambda_i(X \Sigma X^*) = \lambda_i(\Sigma X^* X)$, where Σ is symmetric (this equality also holds for spectral radii) and $\lambda_{min}(\bar{\Sigma}_{n+1} CC^T) > \lambda_{min}(\bar{\Sigma}_{n+1}) \lambda_{min}(CC^T)$. Note that $\kappa(\Sigma_n) = 1$. Therefore sufficient conditions for the decay of the covariance are:

$$(1) \qquad \rho(AA^T) \leq 1 \implies |\lambda_{max}(AA^T)| \leq 1 \tag{A.16}$$

$$(2) \qquad \kappa(A)^2 \leq 1 \implies |\lambda_{min}(A)| = |\lambda_{max}(A)| \tag{A.17}$$

$$(3) \qquad \kappa(CC^T) \leq 1 \tag{A.18}$$

For the linearised dynamics derived in eq. (A.5) the eigenvalues of AA^T, λ, are given by $\lambda^2(1-\lambda)^2 = 0$, and the eigenvalues of A are $\lambda = (0,\ 0,\ \pm\sqrt{\frac{(m+M)}{ML}g})$. Therefore (1) and (2) are true.

Therefore, the covariance decays to zero if $\rho(CC^T) \leq 1$ in the linearised multivariate case and the 2D state becomes a lossless estimate of the state. Given that C is a 2x4 projection matrix of the form

$$C = \begin{bmatrix} c_{11} & 0 & 0 & 0 \\ 0 & 0 & c_{23} & 0 \end{bmatrix} \implies CC^T = \begin{bmatrix} c_{11}^2 & 0 \\ 0 & c_{23}^2 \end{bmatrix} \tag{A.19}$$

By the choice of $\boldsymbol{x}^{(2D)}$ as square, $c_{11} = c_{23}$. By the same logic as above, $\kappa(CC^T) = 1$.

The rate of decay for the univariate case can be determined with a first order approximation about $\sigma_v^2 = 0$:

$$\sigma_{n+1}^2(\sigma_v^2) = A^2\sigma_n^2(1 - \frac{C^2A^2\sigma_n^2}{\sigma_v^2 + C^2A^2\sigma_n^2}) \tag{A.20}$$

$$\approx \sigma_{n+1}^2(0) + \delta\sigma_v^2 \frac{\partial\sigma_{n+1}^2(\sigma_v^2)}{\partial\sigma_v^2}|_{\sigma_v^2=0} \tag{A.21}$$

$$= \frac{\delta\sigma_v^2}{(AC\sigma_n)^2} \tag{A.22}$$

A.3 Neural Network Losses

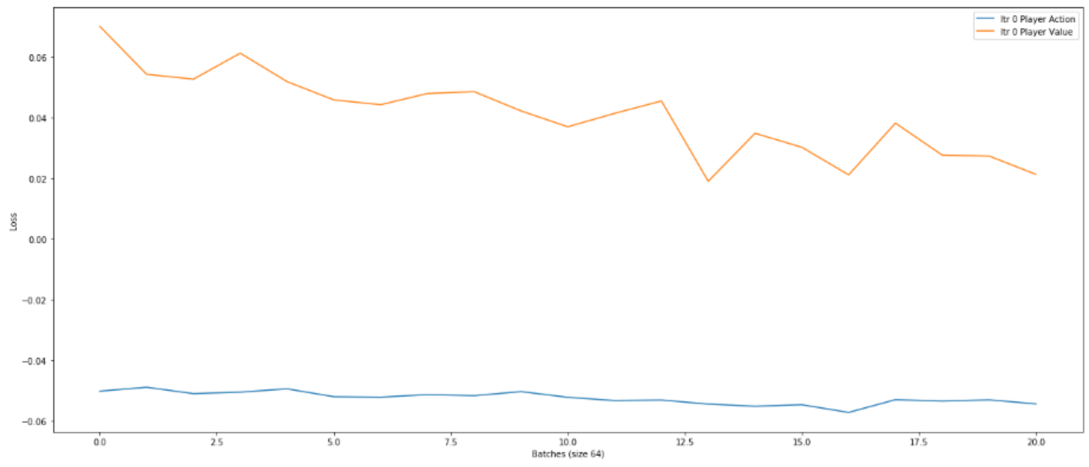


Figure A.1: The losses for both the player and adversary neural networks after one set of training examples.

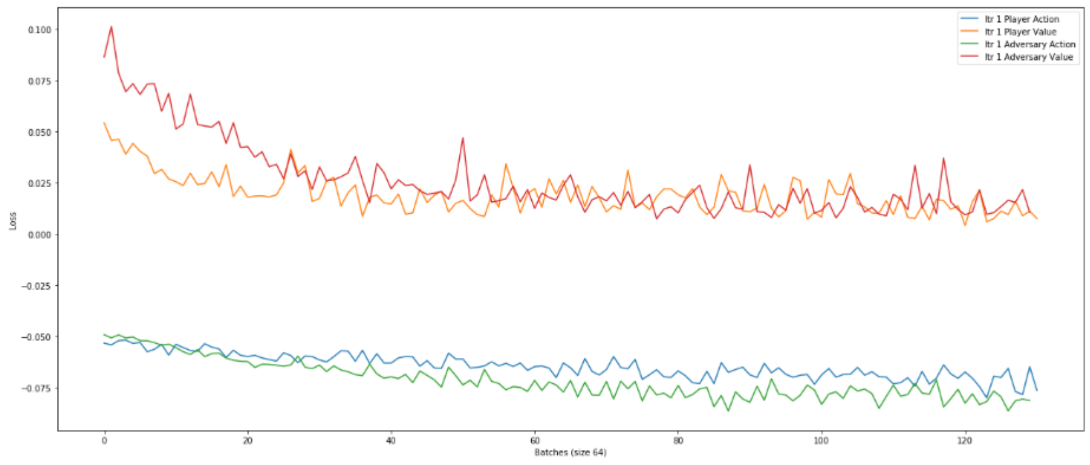


Figure A.2: The losses for both the player and adversary neural networks after two sets of training examples.

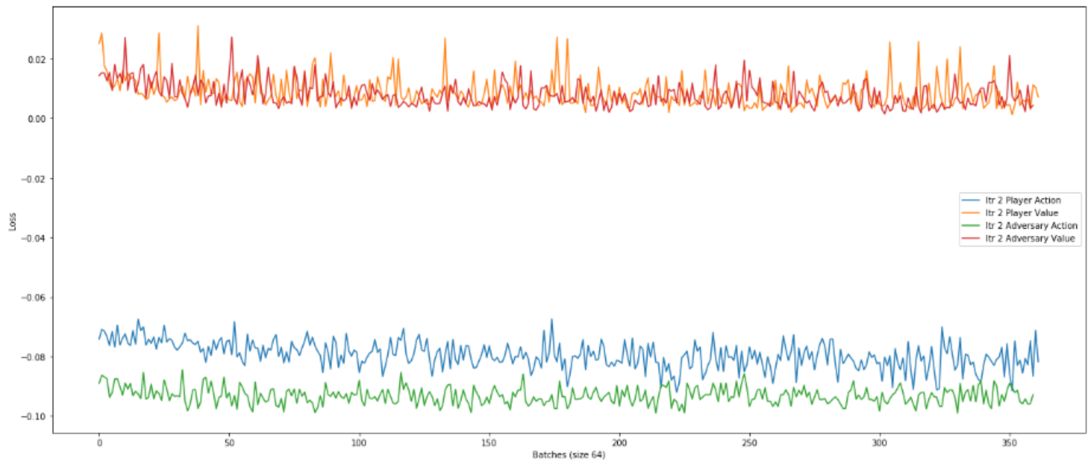


Figure A.3: The losses for both the player and adversary neural networks after three sets of training examples.

A.4 Retrospective Risk Assessment

Being a computer-based project, the risks are minimal. In the original risk assessment the states risks were eye-strain from looking at a computer screen for too long, and back pain from sitting in a single position for too long. Neither of these risks occured over the course of the project.

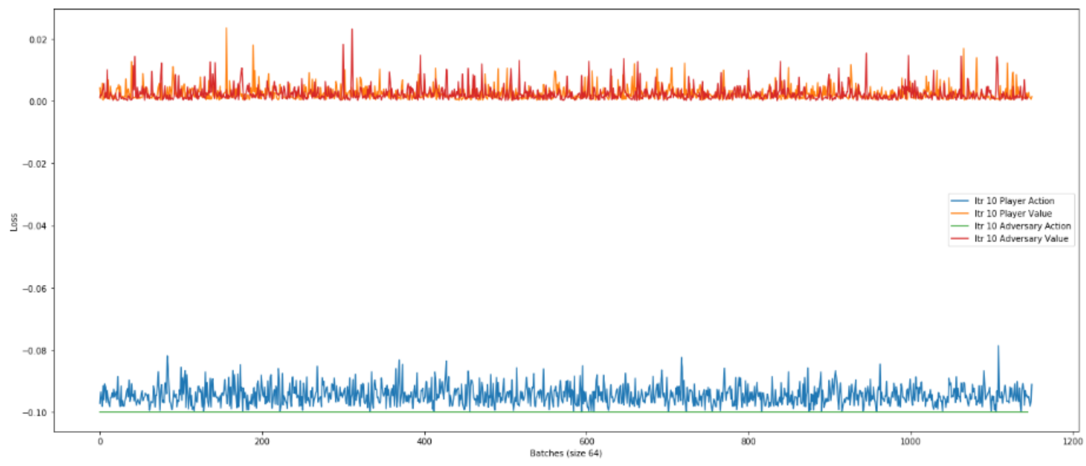


Figure A.4: The losses for both the player and adversary neural networks after ten sets of training examples.