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

St John's College

Cambridge

May 14, 2019

0 Contents

1 Introduction

1.1 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 import than a prediction of the long term qualitative behaviour of the system. Much 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 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.

1.2 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 discrete time, 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:

$$\begin{aligned} V(x_k, k) &= \underset{u_{k:h-1}}{min}\left(\sum_{i=k}^{h-1} c(x_i, u_i) + J_h(x_h)\right) && (1.1)\\ &= \underset{u_k}{min}\Big(c(x_k, u_k) + V(x_{k+1}, k+1)\Big) && (1.2)\\ u_k^* &= \underset{u_k}{argmin}\Big(c(x_k, u_k) + V(x_{k+1}, k+1)\Big) && (1.3) \end{aligned}$$

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 for the linear case with quadratic costs analytically (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 5/6 dimensions in discrete cases [2].

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 [8].

1.3 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):

$$\begin{aligned} V(x_k) &= \underset{u_k}{max}\left(\sum_{i=k}^{\infty} \lambda^{i-k} r(x_i, u_i)\right) && (1.4) \\ &= \underset{u_k}{max}\, \mathbb{E}\Big[r(x_k, u_k) + \lambda V(x_{k+1})\Big] && (1.5) \\ u_k^* &= \underset{u_k}{argmax}\, \mathbb{E}\Big[r(x_k, u_k) + \lambda V(x_{k+1})\Big] && (1.6) \end{aligned}$$

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 [10, 11]. Furthermore, the advent of neural networks has led to the extension of these to functional approximations from tabula-rasa methods, making the control highly non-linear dynamical systems possible. Notably, Deepmind's recent success with training a robot to gently manipulate objects [12], would not be possible to reproduce using classical or modern control techniques due to dimensional problems.

1.4 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.

1.4.1 Self Play and The Neural Network

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 who the current player was on that move 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.

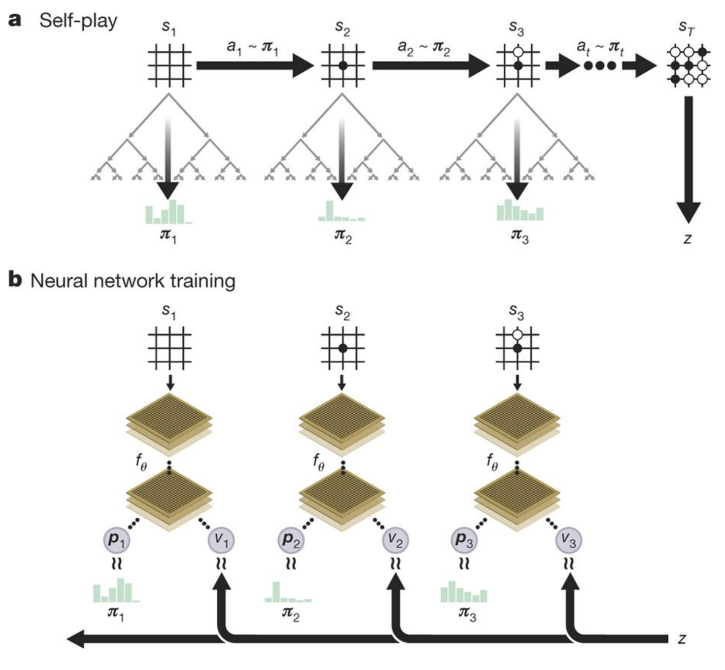

Figure 1.1: A schematic showing how self-play and policy training are performed. Taken from [13].

The loss function for the neural network is given by:

$$\mathcal{L} = (z - v)^2 - \boldsymbol{\pi} \cdot log(\boldsymbol{p}) + c||\theta||^2 \tag{1.7}$$

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, $v \in (-1, 1)$. 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 19-39 residual blocks.

1.4.2 Monte Carlo Tree Search

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

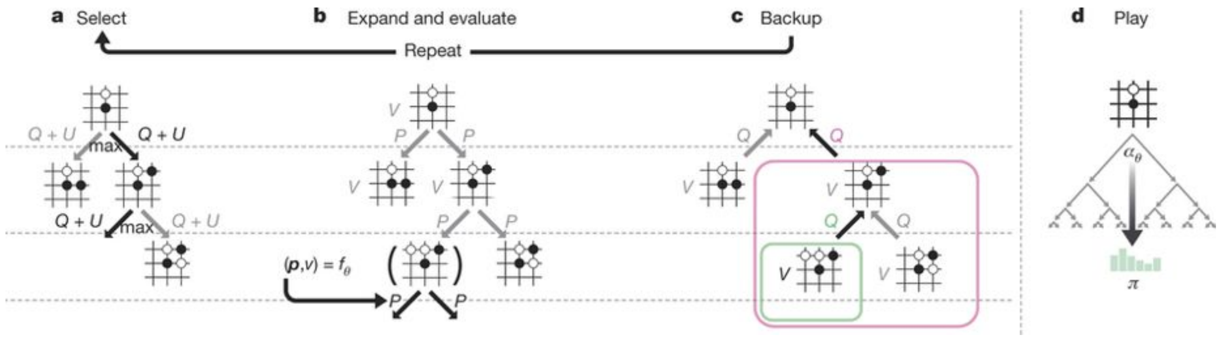

Figure 1.2: A schematic outlining the steps involved in a monte-carlo tree search. Taken from [13].

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- textitp.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 ($\sim$ 1) can be increased to encourage exploration.

$$a_{selected} = \underset{\forall a}{argmax}\Big\{Q(s,a) + c \cdot \boldsymbol{p}_\theta(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}\Big\} \tag{1.8}$$

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.

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s_{t+1}|s_t,a_t} v(s_{t+1}) \tag{1.9}$$

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.

1.4.3 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.

1.5 Ranking and Elo Rating

The Elo rating system is a method of finding players' relative skills in two player games such as chess [16]. The Elo system is based upon every player having an average skill level μ_i, and variation distributed normally about their mean , i.e. $skill \sim \mathcal{N}(skill|\mu_i, \sigma)$. Therefore, the probability of one player beating the other is given by the cumulative density function ($c.d.f.$) of the difference of their Gaussians. This can be approximated as a logistic function, and written as:

$$\mathbb{E}[s_{p1}] = \frac{1}{1 + 10^{(\mu_{p2} - \mu_{p1})/400}} \tag{1.10}$$

Where 400 is a scaling factor such that the means can be used as the player rating, and s is the score. The rule for then updating each player's rating is given by:

$$\mu^{(new)} = \mu^{(old)} + 32(Score - \mathbb{E}[s]) \tag{1.11}$$

where 32 is again an arbitrary constant. New players are given an initial ranking of 1000.

1.6 Summary of Potential Benefits

The application of AlphaZero to dynamical systems a number of potential benefits above traditional optimal control or reinforcement learning. These include:

Robust Disturbance Handling. The Adversary in AlphaZero is effectively worst case scenario disturbance modelling. 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, therefore 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.

This project investigates these.

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.

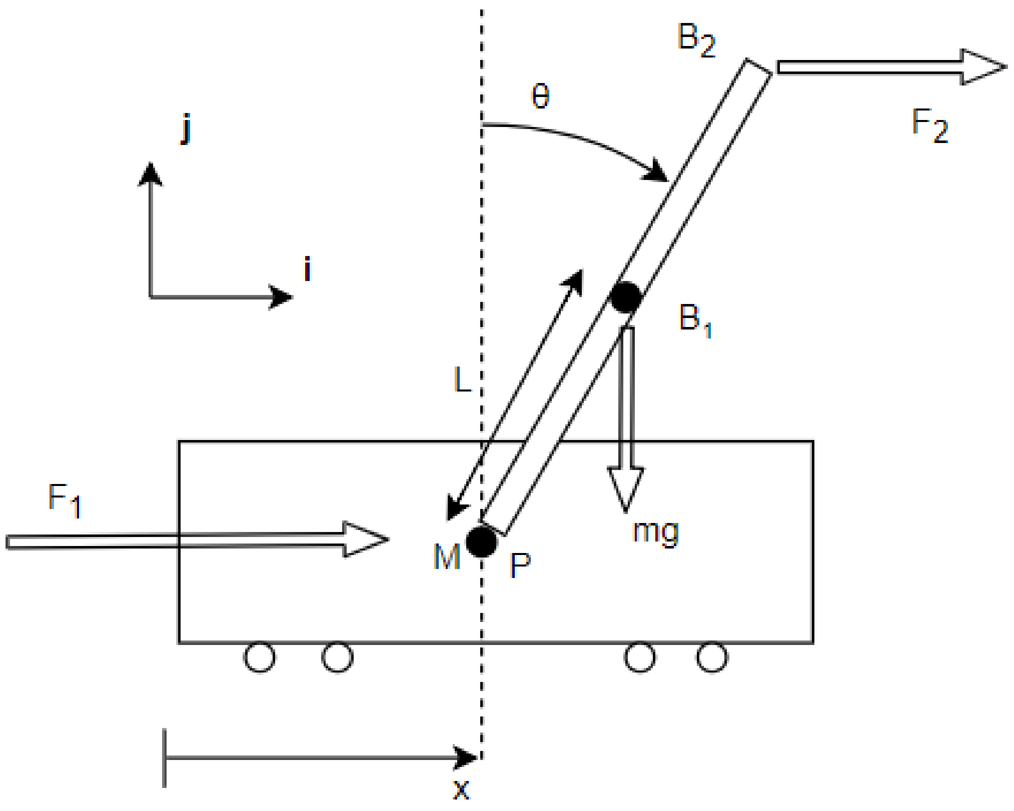

Figure 2.1: A free-body diagram of the inverted pendulum system. For the OpenAI IP the system is in discrete time with a time-step of $\tau = 0.02s$. The other constants are $l = 0.5m$, $m = 0.1kg$, $M = 1kg$, $F = \pm 10N$, $x_{max} = \pm 2.4m$, $\theta_{max} = \pm 12^o$.

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

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

Ignoring second order terms and linearising about $\boldsymbol{x}_e = [x_e, \dot{x}_e, \theta_e, \dot{\theta}_e]^T = [0,0,0,0]^T$:

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

Which, as expected, is unstable since $det(\lambda I - A) = 0 \implies \lambda^2(\lambda^2 + \frac{mg}{lM}) = 0$. Note, for small angles the natural frequency of a non-inverted pendulum is $\omega_n = \sqrt{\frac{mg}{lM}} = \sqrt{\frac{0.1\times 9.81}{0.5\times 1}} \approx 1.40rad/s$. Therefore, the time constant for the system is $\tau \approx 0.70s$.

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 and other extra functionality, whilst providing a rendering function shown in fig. 2.2.

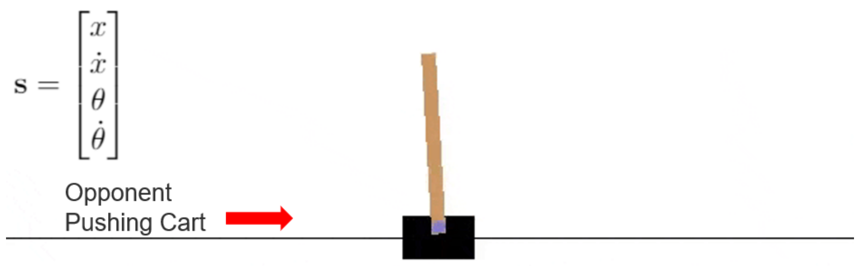
$\mathbf{s} = \begin{bmatrix}x\\ \dot{x}\\ \theta\\ \dot{\theta}\end{bmatrix}$
Opponent Pushing Cart

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/impulse, the 2D state is calculated and a cost, is calculated as in eq. (2.3), where $\boldsymbol{w}^T = [w_1, w_2, w_3, w_4] = [0.25, 0.1, 0.7, 1]$ 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 ***.

$$c(x_t, u_t) = -\frac{1}{\sum_i w_i}\boldsymbol{w} \cdot \left[\left(\frac{x_t}{x_{max}}\right)^2, \left(\frac{\dot{x}_t}{\dot{x}_{max}}\right)^2, \left(\frac{\theta_t}{\theta_{max}}\right)^2, \left(\frac{\dot{\theta}_t}{\dot{\theta}_{max}}\right)^2\right]^T \tag{2.3}$$

better lay-out? x^TQx

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):

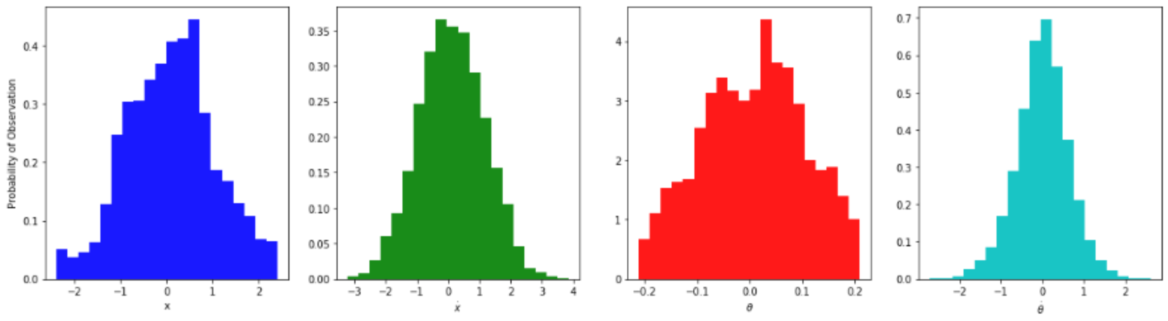

Figure 2.3: Histograms of typical state values. The frequencies greatly depend on the quality of the controller, with better controllers giving much narrower distributions. However, these are typical for a controller of medium efficacy over many episodes where the starting state is randomised (possibly 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. The value function is computed after the episode has completed, and is the discounted future losses at each state. A horizon constraint of $\gamma^k < \frac{1}{20}$ was chosen as it is a standard factor for insignificance. Since the steps taken after a terminal state has been reached, k, must be defined with the game dynamics (the `CartPoleWrapper` class), it is a constant. Therefore γ is calculated as $\gamma < \frac{1}{20}^{\frac{1}{k}}$. The discounted future values are calculated using a geometric series:

$$v_0 = \frac{\sum_{\tau=0}^{k} \gamma^\tau c(x_\tau)}{\sum_{\tau=0}^{k} \gamma_\tau}, \qquad \text{where } \gamma^k < \frac{1}{20} \tag{2.4}$$

Where for simplicity of notation, $v_0 = v(t)$, the state value at step t.

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, following Silver et. al [14], 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.

(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).

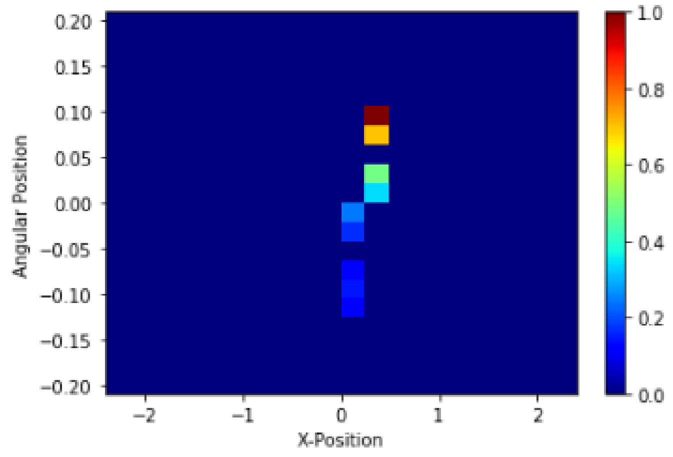

Figure 2.4: An example of a 2D state representation where there are 20 bins and 17 random actions have been taken.

(3) A matrix of discounted previous states, forming a motion history image. An example of this is shown in section 2.1.3, and Algorithm 1 shows the implementation details. This was the chosen representation.

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

The state space model for the quantisation of the linearised inverted pendulum (valid for small time steps) can be modelled as (where $\mathcal{WN}$ = White Noise):

$$\boldsymbol{x}_t^{(2D)} = C\boldsymbol{x}_t + \boldsymbol{V}_t \qquad \boldsymbol{V}_t \sim \mathcal{WN}(0, \sigma_v^2 I) \tag{2.5}$$

$$\boldsymbol{x}_t = A\boldsymbol{x}_{t-1} + B\boldsymbol{u}_t \tag{2.6}$$

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:      $\hat{\boldsymbol{x}}_{t-1}^{(2D)}[\hat{\boldsymbol{x}}_{t-1}^{(2D)} < \lambda^{-(T+1)}] \leftarrow 0$        ▷ Only keep $\hat{\boldsymbol{x}}_{t-1}^{(2D)}$ from $t < T$
9:      return $\hat{\boldsymbol{x}}_t^{(2D)} + \lambda\hat{\boldsymbol{x}}_{t-1}^{(2D)}$
10: end function
```

The initial mean squared error, and the propagation of the error (when estimated optimally) are given by eqs. (2.7) and (2.8) (derivation details can be found in appendix A.2).

$$\sigma_0^2 = \sigma_v^2 = \frac{\delta x^2}{12} \tag{2.7}$$

$$\sigma_{n+1}^2 = A\sigma_n^2\left(1 - \frac{A\sigma_n^2}{A\sigma_n^2 + \sigma_v^2}\right) \tag{2.8}$$

If the spectral density of Equation (2.8) is less than 1 then the error will decay to zero, and the neural network (if acting as an optimal filter) should be able to recover $\boldsymbol{x}$ without loss. The number of steps needed for this recovery decreases with smaller bin sizes since $\lim_{\delta x \to 0} \sigma_v^2 = 0$, and σ_v^2 decays with δx^2.

Work out how small vs how many steps (possibly with a second order approx?)

With limited memory and 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.

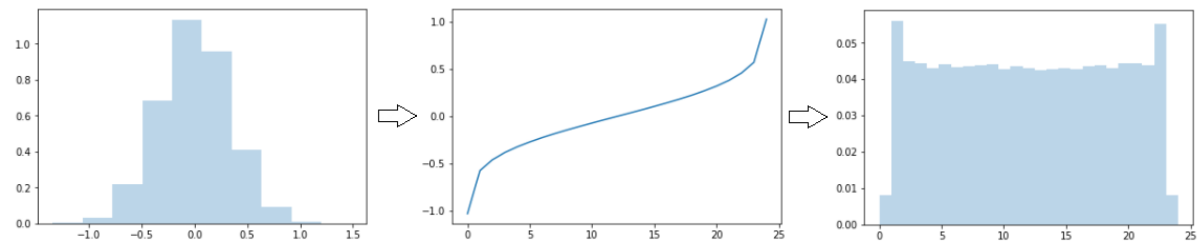

Figure 2.5: Binning of a gaussian distribution with bin edges scaled with an inverse gaussian c.f.d. For this example there are 25 bins.

2.1.4 Discrete vs Continuous Time

For a small enough discrete time step, the simulation will approximate the continuous process very well. Additionally pseudo-continuous actions can be achieved within the constraint of $u \in \{-1, 1\}$ via pulse-width modulation. A time step of 0.02s is 35x smaller than the time constant of the linearised system ($\tau \approx 0.70s$). The positional change per time step is therefore $\sim \frac{1}{35} = 3\%$. Alternatively the average maximum velocities are $3m/s$ and $2rad/s$, thus the maximum positional change per time step is $\dot{x}_{i,max} \times \frac{\tau}{x_{i,max}} = 2.5\%$ and 30% respectively. In practice a 30% change in θ would only occur around θ_{max} and would be uncontrollable, therefore the change can be assumed to be less than this .

can it? Also proof of un-

Therefore, the quick succession of actions can be modelled as a continuous one:

$$F(t) = g(t) * \int_0^\infty u(t)dt \approx g(t) \sum_{n=0}^{\infty} I\tau\delta(t - n\tau) \tag{2.9}$$

work out expression for force? Convolution? Just avg with decay term? Ask vinnicombe

2.2 Self Play and Adversaries

2.2.1 Cost and Value Functions of the Adversary

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 [14], 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 $u_{adv} \in \{-1, 1\}$ was chosen.

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.

`*** Currently method 2 is being used, but subject to change ***`

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.

2.2.3 Episode Execution

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

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
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
```

2.3 Neural Network

2.3.1 Loss Functions and Pareto

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

$$\mathcal{L} = \sum_t (v_\theta(\boldsymbol{x}_t^{(2D)}) - v_t)^2 + c_{pareto} \cdot ||\boldsymbol{p}_\theta(\boldsymbol{x}_t^{(2D)}) - \boldsymbol{\pi}_t||_2^2 \tag{2.10}$$

Where c_{pareto} is a constant. During training the agents move probabilistically throughout, rather than switching to acting greedily after 30 moves as in AlphaZero [14]. As such, mean squared error was used for the action-losses.

`is this actually a good idea?  Why not use greedy?`

Due to the asymmetric nature of the problem, the value and action-losses can be of very different magnitudes. A constant factor, c_{pareto}, was used to prevent this.

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, and another that takes the 2D state, $\boldsymbol{x}^{(2D)}$. 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 2 feedforward layers with *ReLU* activations. The two "heads" split here and have 2 fully-connected layers each, 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*, a mini-batch size of 8, a dropout of 0.3 and batch normalisation. 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 NVIDEA 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 12hrs.

[1]www.pytorch.org

2.4 MCTS

The general MCTS algorithm, outlined in section 1.4.2, was implemented for the inverted pendulum as in Algorithm 3.

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 are multiplied by 10^{12} and converted to 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. For an episode length of 200, the probability of revisiting a state again from a different trajectory is $(S \times L)10^{-12^4}$: impossibly small.

Binning in this manner means that S is limited only in computing power. Having $S > 50$ leads to computing times of days on the NVIDEA GeForce GTX 1050 GPU used for this project (S = 1600 for AlphaZero).

2.4.2 Action Selection

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

$$u^*_{player} = \underset{u\ in\ \mathcal{U}_{player}}{argmax} \left\{ Q(\boldsymbol{x}, u) + c \cdot \boldsymbol{p}_\theta(\boldsymbol{x}, u) \frac{\sqrt{\sum_b N(\boldsymbol{x}, b)}}{1 + N(\boldsymbol{x}, b)} \right\} = \underset{u\ in\ \mathcal{U}_{player}}{argmax}\, U_{player}(\boldsymbol{x}, u) \quad (2.11)$$

$$u^*_{adv} = \underset{u\ in\ \mathcal{U}_{adv}}{argmax} \left\{ -Q(\boldsymbol{x}, u) + c \cdot \boldsymbol{p}_\theta(\boldsymbol{x}, u) \frac{\sqrt{\sum_b N(\boldsymbol{x}, b)}}{1 + N(\boldsymbol{x}, b)} \right\} = \underset{u\ in\ \mathcal{U}_{adv}}{argmax} U_{adv}(\boldsymbol{x}, u) \quad (2.12)$$

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, and at that state the neural network should also return -1. Additionally, when the search reaches 200 steps (the arbitrarily set "end" of the episode), the neural network will not be aware of this fact. Therefore, when any terminal state 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 how "good" their policy is compared to other policies becomes difficult. For example, if the pendulum stays up for longer, is is difficult to determine whether the player has improved because it is controlling the system better, or the adversary is less effective, or both have improved but one improved more. Handicapping the adversary such that it is equal to the player is not practical due to the non-linear relationships between external forces, and gravity, on the dynamics. Furthermore, the inverted pendulum must stay up for at least some amount of time, otherwise the network would not have the training examples to learn from. Therefore handicapping the adversary with $F_2 = \alpha F_1$, and limiting the adversary to push every K steps were both used ($\alpha \sim 0.05$ and $K \sim 20$ initially). The score used is the number of steps the agent stayed up for. For the player this ranges between 0 and 200, and for the adversary $score = 200 - steps$ is used.

does this make sense?

Over one policy iteration, there are 4 different policies: the two current player and adversary policies, $f^{(player)}_{\theta,\, curr}$ and $f^{(adv)}_{\theta,\, curr}$, and the challenger policies, $f^{(player)}_{\theta,\, chal}$ and $f^{(adv)}_{\theta,\, chal}$. The improvement of the player can be determined by pitting the current player against the current adversary, and then the challenging player against the current adversary; and vice-versa for the adversary improvement[2]. This gives the four equations in eq. (1.10).

$$\mathbb{E}[s^p_{curr}] = \frac{1}{1 + 10{}^\wedge\left(\frac{\mu^a_{curr} - \mu^p_{curr}}{400}\right)} \qquad \mathbb{E}[s^a_{curr}] = \frac{1}{1 + 10{}^\wedge\left(\frac{\mu^p_{curr} - \mu^a_{curr}}{400}\right)} \tag{2.13}$$

$$\mathbb{E}[s^p_{chal}] = \frac{1}{1 + 10{}^\wedge\left(\frac{\mu^a_{curr} - \mu^p_{chal}}{400}\right)} \qquad \mathbb{E}[s^a_{chal}] = \frac{1}{1 + 10{}^\wedge\left(\frac{\mu^p_{curr} - \mu^a_{chal}}{400}\right)} \tag{2.14}$$

$$\tag{2.15}$$

where $\mathbb{E}[s^p_{curr}] = \mathbb{E}[s(f^{(player)}_{\theta,\, curr})]$ denotes the score for the current player, and $\mu^p_{curr} = \mu(f^{(player)}_{\theta,\, curr})$ is the rating of the current player. The expected scores can be estimated via monte-carlo. Rearranging gives the relation between successive agent ratings as:

$$\mu^p_{chal} = \mu^p_{curr} + 400log_{10}\left(\frac{E[s^p_{chal}]}{1 - E[s^p_{chal}]} \cdot \frac{1 - E[s^p_{curr}]}{E[s^p_{curr}]}\right) \tag{2.16}$$

[2]Note, if the ratings were correctly updated in the last policy iteration then $\mathbb{E}[\mathbb{E}[s^{agent}_{curr}] - \mu^{agent}_{curr}] = 0$

Algorithm 3 MCTS (based on S.Nair's implementation [15])

```
1: function getActionProb($\boldsymbol{x}_{root}, \boldsymbol{x}^{(2D)}, agent$)
2:     for i in nMCTSSims do                    ▷ Simulate nMCTSSims episodes from $\boldsymbol{x}_{root}$
3:         $reset(\boldsymbol{x}_{root})$
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

10: function MCTSSearch($\boldsymbol{x}^{(2D)}, agent$)
11:     if $\boldsymbol{x}$ is terminal then
12:         $\pi, v \leftarrow f_\theta(\boldsymbol{x}^{(2D)})$
13:         return $v$                       ▷ Use $v_{nnet}$ whether fallen or past the step limit
14:     end if
15:
16:     if $\boldsymbol{x} \notin Tree$ then                                    ▷ Expand Leaf Node
17:         $\pi, v \leftarrow f_\theta(\boldsymbol{x}^{(2D)})$
18:         $N(\boldsymbol{x}, \cdot) \leftarrow 0$
19:         $P(\boldsymbol{x}, \cdot) \leftarrow \pi$
20:         return $v$
21:     end if
22:
23:     if $agent = player$ then                             ▷ Get best action using UCB
24:         $u^* \leftarrow \underset{u \in \mathcal{U}_{player}}{argmax} U_{player}(\boldsymbol{x}, u)$
25:     else
26:         $u^* \leftarrow \underset{u \in \mathcal{U}_{adv}}{argmax} U_{adv}(\boldsymbol{x}, u)$
27:     end if
28:     $\boldsymbol{x}, c \leftarrow step(\boldsymbol{x}, u)$                         ▷ Take next MCTS simulated step
29:     $\boldsymbol{x}^{(2D)} \leftarrow getState2D(\boldsymbol{x}, \boldsymbol{x}^{(2D)})$
30:     $agent \leftarrow nextAgent(agent)$
31:     $v \leftarrow MCTSSearch(\boldsymbol{x}^{(2D)}, agent)$                  ▷ Recursion to next node
32:
33:     $Q(\boldsymbol{x}, u^*) \leftarrow \frac{N(\boldsymbol{x},u^*)Q(\boldsymbol{x},u^*)+v}{N(\boldsymbol{x},u^*)+1}$              ▷ Backup Q-values up the tree
34:     $N(\boldsymbol{x}, u^*) \leftarrow N(\boldsymbol{x}, u^*) + 1$
35:     return $v$
36: end function
```

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

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

[10] 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

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

[12] 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.

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

[14] 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.

[15] 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`

[16] HowlingPixel.com, *Elo Rating System.* `https://howlingpixel.com/i-en/Elo_rating_system` acc: 11/03/2019. published 2019

A Appendices

A.1 Inverted Pendulum Dynamics Derivation

We can find the state space equations for the Inverted Pendulum using d'Alembert forces. Firstly we 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, $\boldsymbol{h}_P = \boldsymbol{r}_{B_1/P} \times m\dot{\boldsymbol{r}}_{B_1}$:

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

We can 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 $\boldsymbol{F}_e = \dot{\boldsymbol{\rho}}$:

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

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

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

$$\ddot{x}(M + msin^2\theta) = F_1 + F_2cos(2\theta) + msin\theta(L\dot{\theta}^2 - gcos\theta) \quad (A.2)$$

Simplifying this for our problem by substituting in constants, we can write the full state space equation:

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

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

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

Therefore for any initial condition we can reach $\boldsymbol{x}_e$ in finite time under these linear assumptions. However, for $\theta \approx 0$ we need a more sophisticated model.

A.2 Propagation of Quantisation Error

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

$$\boldsymbol{x}_t^{(2D)} = C\boldsymbol{x}_t + \boldsymbol{V}_t \qquad \boldsymbol{V}_t \sim WN(0, \sigma_v^2 I) \quad (A.5)$$

$$\boldsymbol{x}_t = A\boldsymbol{x}_{t-1} + B\boldsymbol{u}_t \quad (A.6)$$

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 **V**.

*** This derivation here is for 1D and also follows the derivation from 4F7, which does not take into account actions (B=0), and has additional noise on the second equation. However, because actions are deterministic, it should have no effect on the propagated MSE. Could fairly easily do a full derivation for this, but is it that necessary? Also σ_v^2 is not true, as this is only a 1D error. Finally, possible spanner... Limiting the memory of the system to only t-k could place limits on the accuracy? Should do full derivation if time allows. Note, is this even the correct way to go about this? Can we assume a neural network will act as a kalman filter?***

Assuming the quantisation bin size, δx, is small, the quantisation noise can be modelled as uniform random variable with a mean squared error:

$$\sigma_v^2 = \mathbb{E}[V_n^2] = \int_{-\delta x/2}^{\delta x/2} u^2 \cdot \frac{1}{\delta x} du = \frac{\delta x^2}{12} \tag{A.7}$$

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

Algorithm 4 Kalman Filtering

```
1: Given: $\hat{X}_n = K[X_n|Y_{n-k:n}]$ and $\sigma^2 = \mathbb{E}[(X_n - \hat{X}_n)^2]$

   Prediction:
2: $\bar{X}_{n+1} = K[X_n|Y_{n-k:n}] = f_n \hat{X}_n$
3: $\bar{\sigma}_{n+1}^2 = \mathbb{E}[(X_{n+1} - \bar{X}_{n+1})^2] = f_n \sigma_n^2 + \sigma_w^2$        ▷ $\sigma_w^2 = 0$

   Update:
4: $I_{n+1} = Y_{n+1} - g_{n+1}\bar{X}_{n+1}$
5: $\hat{X}_{n+1} = \bar{X}_{n+1} + \frac{g_{n+1}\bar{\sigma}_{n+1}^2}{g_{n+1}^2\bar{\sigma}_{n+1}^2 + \sigma_v^2} I_{n+1}$        ▷ $\sigma_v^2 = \frac{\delta x^2}{12}$
6: $\sigma_{n+1}^2 = \bar{\sigma}_{n+1}^2 \left(1 - \frac{g_{n+1}^2\bar{\sigma}_{n+1}^2}{g_{n+1}^2\bar{\sigma}_{n+1}^2 + \sigma_v^2}\right)$
```

Thus we can find the optimal mean squared error of the next value from algorithm 4 line 6:

$$\sigma_{n+1}^2 = f_n \sigma_n^2 \left(1 - \frac{f_n \sigma_n^2}{f_n \sigma_n^2 + \sigma_v^2}\right) \tag{A.8}$$