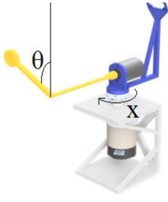


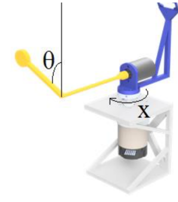


Image Control of the Inverted Pendulum

Nicholas Tan Erik Augustine Sean Fitzgerald

December 16, 2017

1 Introduction

There are many different techniques for the measurement and control of dynamic real-world systems. The traditional approach to motor control problems is to find angular position/velocity with one or more rotary encoders. This information is then typical fed into a control algorithm to administer negative feedback on the observed state.

In applications where encoders are not feasible, other techniques must be used in order to observe the system. This restriction is the motivation for our project – we aim to control the inverted pendulum solely through images of the system. The model will learn how the images are associated with the angular states of the system, and then it will apply different RL techniques to decide the optimal action for the motor in order to keep the pendulum balanced without any prior information about the physical dynamics.

This project compares three different methods to control the inverted pendulum. The first uses a Neural Network to determine the angle of the system from the input images, and then subsequently uses Reinforcement Learning to control and stabilize the system. The second design is the traditional classical control approach that serves as our baseline. These two methods were done successfully in simulation, so we moved onto our third model - physical implementation. For this, we experimented with an adapted form of the above method as well as a new addition with Deep Q Learning [1].

2 Related Work

Our goal was strongly motivated by the company DeepMind and their work on model-free reinforcement learning. Mnih et al. created an algorithm which was capable of online training on a subset of Atari games to achieve super-human level of play ([1],[2]). Many of their techniques were employed in our learning algorithms. More specifically, we chose to implement deep q learning, experience tuples in replay memory, root mean square propagation [4], and online mini-batch training using two separate Neural Networks, updated periodically.

Anderson discusses balancing an inverted pendulum using two neural networks in his paper "Learning to Control an Inverted Pendulum Using Neural Networks" [5]. There, he presents a method that uses one Neural Network (the *action network*) for choosing the preferred action (i.e. moving left or right) and another (the *evaluation network*) for evaluating failure and editing the action network. Both these networks, in Anderson's work, use only a single neuron. We evaluated that, while Anderson had the correct overall approach to model-free learning, his methods were too simple to encapsulate the complexity of the pendulum swing-up problem.

Doya used a machine learning approach to solve the cart-pole balancing problem [6]. This problem, similar to the rotary pendulum, uses a cart that only moves linearly. In her work, Doya shows that the actor-critic reinforcement learning is capable of solving the limited-torque swing up problem, but that it is inefficient when used on the cart-pole swing-up problem. We used Doya's work to convince ourselves that the rotary inverted pendulum swing-up problem is possible to solve with a machine learning approach.

3 Design Methods

3.1 Simulation & Physical Model

The physical system dynamics we used to model the rotary inverted pendulum is based on Cazzolato's work [3]. With the update equations, we approximate each subsequent state of the pendulum with Euler's Method and a timestep of $\Delta t = 0.1$ seconds, representative of a hardware-achievable frame rate (10Hz). The output is a state that corresponds with the pendulum's angle, and its derivative.

Furthermore, we designed and built a physical rotary inverted pendulum. Our setup mounts a camera on the primary arm to take images which are fed into a Raspberry Pi for high level processing. The Raspberry Pi communicates over SPI to an Arduino that handles low level control and tachometer readings.

3.2 Image Recognition Neural Network

To design the architecture of our Neural Network, we took inspiration from the system hardware we worked with. For example, the camera had a resolution of 640x480, so $640 \times 480 = 307200$ became the dimension of the first layer of our Neural Network. The encoder has a resolution of 200 steps per revolution, so we let our output layer have 200 nodes, as a "one hot" encoding for each encoder step.

Our Neural Network design initially began with three fully connected layers (one hidden layer with 1000 nodes), taking a 640×480 image directly as an input. However, during development, we realized that $640 \times 480 = 307200$ nodes in the first layer took too much compute resources to be feasible. Even initializing the weights could not complete in a reasonable amount of time. To mitigate this issue, we used a bigger computer with more memory as well as downsampled our image by a factor of 100. Thus, the first layer of the network took a $64 \times 48 = 3072$ pixel vectorized image that was representative of the angle of the pendulum.

With this three layer network, we were able to train our network to convergence, but we were not achieving the performance we desired. Indeed, the training accuracy was 0.5%. To mitigate against underfitting, we decided to add more features in the form of more layers in our network.

Restructuring our network to have 3 hidden layers improved our accuracy to 95%. We sized the 3 hidden layers with 1000, 700, and 400 nodes to minimize the amount of information compression that happened between each layer.

Lastly, despite our Neural Network achieving good performance in simulation, we again faced a setback with computational resources when transitioning to the physical system. When applying the image recognition Neural Network to the physical model, we became severely time restricted. Each cycle not only predicted the correct action, but also included overhead of processing the image and serial communication with the Arduino. Camera frames were compressed to 32x32 pixels and only the luminance layer of the YUV encoding was used. Similarly, the network structure was compressed to one layer of 50 neurons. To save memory, the output was discretized into the number of angle bins used in the reinforcement learning algorithm (discussed below).

3.2.1 Training the Neural Network

To train our neural net, we acquired a dataset through simulation. We generated mock images of our system by sampling θ from a uniform distribution on the interval $[0, 2\pi)$ and drawing the pendulum at that angle. We then solve for the weights in the neural net that minimize the cross-entropy loss using "Mini-Batch" (batch size of 50 images) Gradient Descent. Forward propogation is computed from the equations below:

1. Input: $z^{[0]} \in \{0, 1, ..., 255\}^{3072}$
2. Calculate 3 hidden layers: $z^{[l]} = W^{[l]} z^{[l-1]} + b^{[l]}$, $a^{[l]} = \sigma(z^{[l]})$
3. Calculate Output: $y = \text{softmax}(z^{[3]}) = \tilde{\theta}_{onehot}$

Backwards propogation is given by first forward propogating through the data and then calculating the gradients from the output to the input. Specifically:

1. Define δ of the output layer: $\delta^{[4]} = \nabla_{z^4} L(\hat{y}, y) = y - \hat{y}$

2. Define δ of the hidden and input layers:
   For l = 3,2,1:
   $\delta^{[l]} = (W^{[l+1]})^T \delta^{[l+1]} \circ g'(z^{[l]})$

3. Compute the gradients of layer l as:
   $\nabla_{W^{[l]}} L = \delta^{[l]} (a^{[l-1]})^T$
   $\nabla_{b^{[l]}} L = \delta^{[l]}$

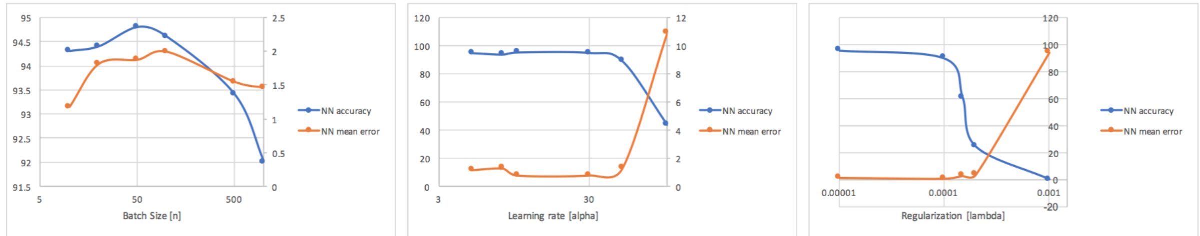
After calculating the gradients of our parameters, we are able to run "Mini-Batch" Gradient Descent:

$$W^{[l]} = (1 - 2\alpha\lambda)W^{[l]} - \alpha \frac{\partial L}{\partial W^{[l]}}$$

$$b^{[l]} = b^{[l]} - \alpha \frac{\partial L}{\partial b^{[l]}}$$

This optimization is repeated until convergence.

To choose our hyperparameters, we ran experiments to see the effect of varying mini-batch size, learning rate, and regularization on performance (in terms of prediction accuracy and mean error).



The batch size was chosen to be 50; the learning rate was chosen to be 10; and the regularization parameter was chosen to be 0.00001.

The neural network used for image recognition on the physical model is smaller and less complex than the one used on the simulation. To make up for the network being less complex and trained on less data, we implemented root mean square propogation in [4] to help our network parameters converge more efficiently. The RMSProp algorithm, although never officially published, uses exponential smoothing to maintain a rolling history of parameter gradients. The learning step size is then weighted by this historical data. The effect is to allow parameters in the same layer to take differently sized steps, understanding that the gradient of one might affect the gradient of another. The specific variant of RMSProp we used is defined by the following equations, where $\lambda \in [0, 1]$ describes the influence (or "momentum") of the learning step.

$$v(w_t) = \lambda v(w_t - 1) + (1 - \lambda)\nabla L(w_t)^2$$
$$w = w - \frac{\eta}{\sqrt{v(w_t)}} \nabla L(w_t)$$

Otherwise, the physical model used the same forward and backpropogation described above, customized to its smaller structure.

3.3 Reinforcement Learning

To begin structuring our reinforcement learning algorithm, we decided on a state space for the MDP. We chose the state of the MDP to be a discretized version of the physical analog state: s $\in \mathbb{R}^3$ given by $s = [\theta, \frac{d\theta}{dt}, \frac{dx}{dt}]$. To identify the optimal discretization, we ran a small study to show the effects of varying state space size against performance. We found that the reinforcement learning did not need many states to be successful at its task. In particular, we did not see improvement over our baseline of 4 states associated with the pendulum arm compared to using 8, 16, and 32 states. Thus, to save on space and computation, we decided to use 4 states to represent the pendulum arm and similarly, 4 states for the pendulum arm velocity.

Next, we had to decide on an action space. In this case, the actions correspond to how much torque we apply to our motor. Similar to our states, we discretize our actions into five choices: [-2, -1, 0, 1, 2], with each action corresponding to the torque applied to the motor.

We trained the algorithm through repeated trial, error, and observation. Although the true state transition probabilities are unknown to the agent, they are implicitly given by the physics of the system.

The agent is thus able to observe and approximate the state transition matrix and assign values to each state using Value Iteration, which used a discount factor of $\gamma = 0.995$.

To begin, we used a reward system where failures were assigned a negative reward and all other states were assigned 0 reward. In this case, we rely on the state transition matrix to predictively avoid states that would be more likely to lead to failure. We also explored using a reward system where greater rewards were given to states closer to the optimal position of "upright," which gave better results with respect to total time spent without falling over.

Transitioning to the physical model used by the hardware, we added the arm position to the state space (discussed in Section 4.2), yielding:

$$s = \{\theta_{\text{motor}}, \frac{\partial\theta_{\text{motor}}}{\partial t}, \theta_{\text{pendulum}}, \frac{\partial\theta_{\text{pendulum}}}{\partial t}\}$$

We also found we needed to reduce the action space to 3 actions (left, right, or nothing) to allow for additional memory to process the image.

The second important part of reinforcement learning is rewarding the state machine. The swing up and balance problem required a complicated reward system that emphasized the following three objectives, in order of priority:

1. For safety of the test setup, the motor should not perform more than one revolution. It should not hit hard against the physical interlock that prevented full revolution.
2. The pendulum should prefer to be in the upper half of its swing arc, which can only be achieved by swinging left or right (Or hitting the interlock, although this was incorporated into priority 1).
3. When in the upright position, the pendulum should prefer to stay there, rather than quickly swing back around the arc to return to the upright position.

The MDP was thus rewarded only based upon motor and pendulum angle. The back half of the motor swing arc received a weight of -5. The reward for the pendulum swing arc was linearly increased from -0.25 in bottom dead center to 0.75 at top dead center. These two rewards were summed and enforced upon the MDP.

3.4 Deep Q Learning

Deep Q Learning is a reinforcement learning algorithm that is used when only very abstract state information is available. When shown the frames and rewards for several Atari games in [2], Deep Q Learning has successfully executed online reinforcement learning and eventually outperformed human-level play. The algorithm is an adaptation of Q-learning with the following features.

Experience Replay: Every time a decision is made, the state transition, action, and observed reward are stored in "replay memory." A mini-batch is then randomly sampled from all of replay memory for gradient descent. This reduces the contextual bias of the neural network.

Separate networks for learning and prediction: Deep Q Learning maintains two separate neural networks, $Q(s,a)$ and $\hat{Q}(s,a)$. Q is used for prediction and is updated at every gradient descent. $\hat{Q}$ is used as the truth reference for the gradient descent steps on Q. Periodically, the parameters or Q are copied to $\hat{Q}$.

We further modified this algorithm for our swing-up problem. Instead of a neural network that predicted only reward from combined state and action input, we used softmax to predict a reward for all actions from only the input state. This reduced the number of times we ran forward propagation when predicting the preferred action.

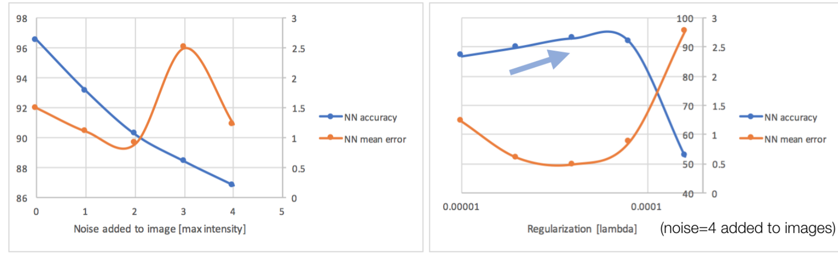
To fully explore the physical model in a short period of time, an annealed ϵ-greedy strategy was used when choosing an action for each state: The system chooses a preferred action using the learning algorithm. The system then chooses to use either a random action (with probability ϵ) or the preferred action (with probability $(1-\epsilon)$). The value of epsilon is linearly annealed from 1 at the beginning of the session to 0.05 around $\frac{1}{50}T$ through the session, where T is the total allotted learning time.

4 Experiments, Discussion, and Results

4.1 Neural Networks

We investigated the effects of adding noise to our pendulum images (during simulation training) on the performance of the Neural Net. As expected, the ability of the Neural Net to accurately predict angle

decreased the more noise we added. However, we could mitigate the loss incurred from this noise by increasing the regularization applied (within reason). This parameter helped to better generalize the model and avoid over-fitting.



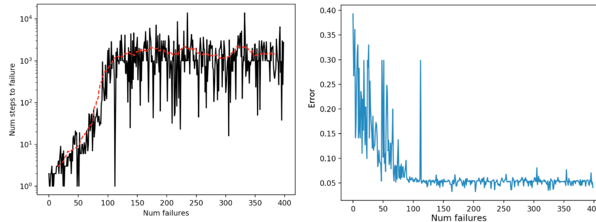
4.2 Reinforcement Learning

The two metrics we used to benchmark our project were time spent upright, and deviation from a "perfectly" upright position (error).

The two main difficulties we found during our work was feature selection and determining the number of states to use in Reinforcement Learning. To begin, we tried using only the pendulum angle and velocity to control the system. With only this information encoded the pendulum would only stay balanced for up to 30 seconds. We were able to greatly improve performance by encoding the motor arm velocity into our states. With this addition we were able to balance for over 150 seconds on average, and had an average error of 2.96^o from the upright position. However, this performance boost necessitated a hardware upgrade to read both the encoders on the motor as well as the pendulum arm.

We also found that the choice of states could greatly improve performance and the convergence time. We investigated various assignments and extractions of states from underlying parameters. For example, is it worthwhile to assign a state based on the parameter values, or is it sufficient to only use the sign of the velocity? We found that minimally increasing the state size would give optimal results, but would give diminishing returns as the state size got too large.

Control Method	Mean Time to Failure	Mean Error
Observe θ through NN	150 seconds	2.96^o
Observe θ directly	232 seconds	2.71^o
Discrete P Controller	5 seconds	8.09^o



Time to failure vs number of failures and Error vs number of failures

5 Conclusions and Future Work

Our baseline algorithm, which used actual pendulum encoder speed to define system state, was able to swing up and balance the pendulum for only a little less than one second. During this time balancing, it exhibited the quivering characteristic observed in the simulator when balancing a pendulum. However, it became clear that angle fidelity near the top of the pendulum arc is very important in the final balancing stage of the swing up problem. Neither the image recognition neural network nor the Deep Q Network were able to achieve this much success. Both, however, converged to swinging the pendulum as high as possible while avoiding the interlock that prevents full revolution.

While we were unsuccessful in getting the Deep Q Network to balance the pendulum, we believe with additional work it is possible. Our next attempt would be to start with a less general method and slowly generalize the algorithm into a Deep Q Network configuration that is able to achieve convergence.

Lastly we would like to explore integrating the two aspects of our learning. Explicitly, we would like to explore feeding the pendulum images directly into the Reinforcement Learning algorithm as a representation of the state of the system.

6 Contributions

All team members contributed equally to this project.

7 References

[1] V. Mnih et al., "Playing Atari with Deep Reinforcement Learning." 2013.

[2] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529–533, 2015.

[3] B. Cazzolato et al., "On the Dynamics of the Furuta Pendulum" 2011

[4] G. Hinton, N. Srivastava and K. Swersky, Cs.toronto.edu. [Online]. Available: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[5] C. Anderson, "Learning to control an inverted pendulum using neural networks", IEEE Control Systems Magazine, vol. 9, no. 3, pp. 31-37, 1989.

[6] K. Doya, "Reinforcement Learning in Continuous Time and Space," Neural Computation, vol. 12, no. 1, pp. 219–245, 2000.