Virtually Trained Self-Balancing Pendulum

By

Henry Thompson
Kishore Adimulam
Mason Ryan

ECE 445 Final Report, Spring 2019
TA: Amr Martini

Group 31
May 1, 2019

Abstract

This paper documents the design and testing of a virtually trained self balancing pendulum system (SBP). The SBP consisted of a single wheel motorized cart on which the inverted pendulum was mounted. This physical system was modeled as accurately as possible in the Unity game engine, which was then used to train a recursive neural network to balance the inverted pendulum through reinforcement learning. This project was intended to show the efficacy of using the Unity game engine for artificial intelligence inference on physical systems.

Contents

1. Introduction

1.1 Objective

There is a growing use for virtual reality as a training environment for AI for applications in the real world. Game engines like Unity have even released machine learning tool-kits for researchers and developers to experiment training reinforcement learning algorithms inside games and simulations. There has been past work in translating these simulation-trained models to physical systems, such as the project done by OpenAI which taught a robot to stack different colored blocks in a specific order only after seeing it once in a virtual simulation [1]. However, the use of game engines to perform similar tasks has been limited, since there are no clear workflows for how someone can use AI models trained in an easy-to-use game engine like Unity to perform inference tasks in a physical system.

Our solution would be to create a self-balancing inverted pendulum system, which would be trained as a simulation in Unity and uploaded into a physical system. The system consists of a cart connected to a one-dimensional track, with an attached pendulum on a hinge. The goal of the system would be to move the cart on the track in either left or right, in order to balance the pendulum in a vertical position. We would create a physical system which replicates all of the attributes of a 3D simulation, and train the agent in the simulation to learn to balance the pendulum using the Python API and Tensorflow. The trained Tensorflow model would then be uploaded into a microcontroller, which would then use the control signals of the agent to operate motors to balance the real physical pendulum. The Kalman filter will be used to reduce the noise from the output of the IMU to get a better estimate of the cart's current acceleration.

1.2 Background

The ability of virtual reality to model physics and interactions between materials positions it to become an ideal tool for simulating environments for artificial intelligence. This allows experiments to be carried out on a much larger scale, at a fraction of the time and resources required to carry out physical tests. Once the low-level controls and modeling for a system has been determined, learning how to carry out more advanced tasks, such as training a robot to jump or to drive a vehicle autonomously, could become possible through using reinforcement learning inside simulations. Since the applications for virtual trained AI agents are numerous, it begs the question how easily such systems can practically be implemented using existing software. One of the most popular current game engines, Unity, has made it easy for developers to train their own AI agents using Tensorflow through the ML-Agents toolkit [2]. Furthermore, TensorFlow Lite has made it possible to deploy machine learning models and perform inference tasks on embedded devices such as the Raspberry Pi [6]. We aim to create a solution where a Tensorflow model trained in a 3D simulation in Unity can be deployed into a microcontroller to balance a physical inverted pendulum.

2. High Level Design

2.1 Block Diagram

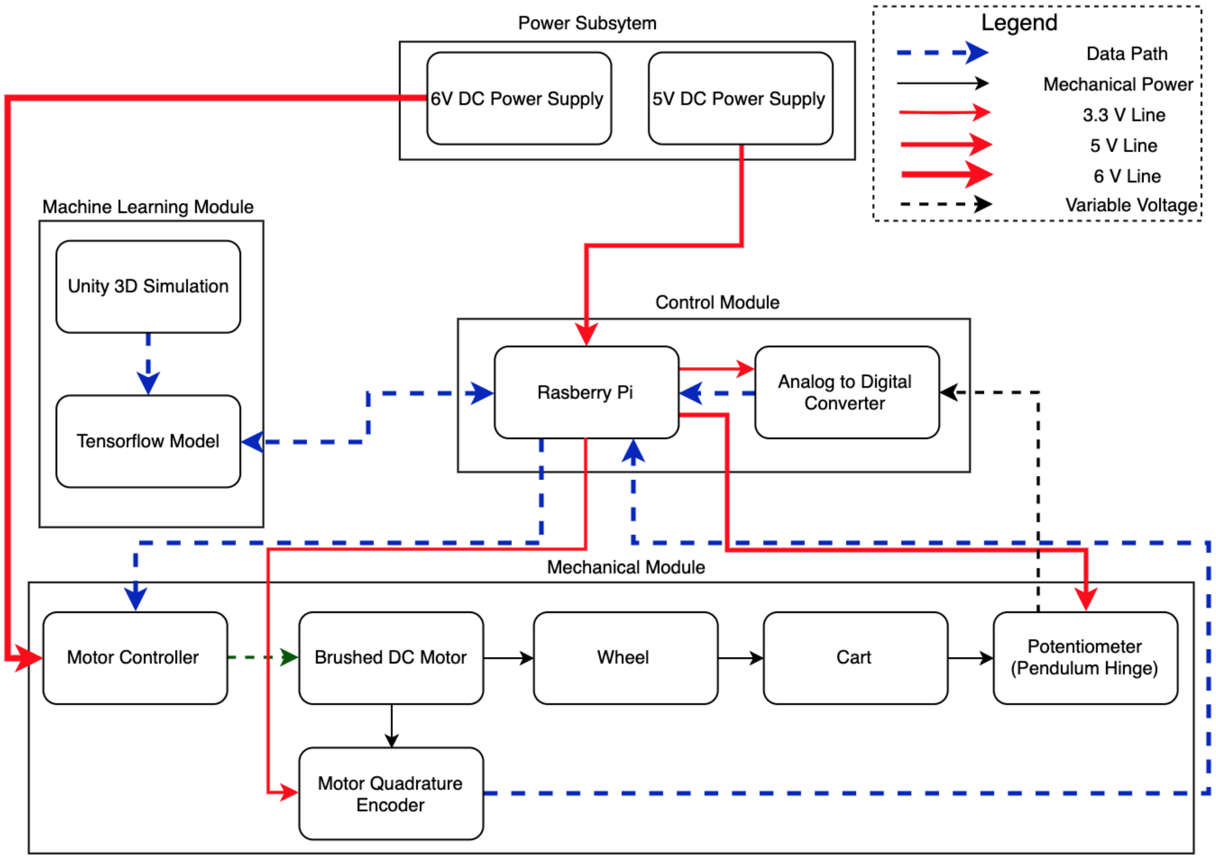


Figure 2.1.1

As seen in Figure 2.1.1, our design will consisted of four major subsystems: machine learning, control, mechanical, and power.

2.2 High Level Design Description

The machine learning subsystem consisted of our Unity simulation and the Tensorflow model produced from the Unity simulation. After the simulation was modeled according to our physical system, an agent was trained to balance the pole using reinforcement learning with the ML-Agents toolkit provided by Unity along with its C# API. Once the simulation was able to consistently able to balance the pole while random impulses tried to knock it done, we took the output tensorflow to run many iterations of the simulation and gather the information of the state/action pairs into text files. The data in these text files was used to train a secondary CNN to learn the relationship between the given state and the action it could take. We chose to use a CNN instead of an RNN since tensorflow lite had incompatibility issues compressing high-level RNN keras modules. This new secondary model was frozen using TFlite and deployed and restored on the Raspberry Pi for inferencing on the physical system. The model was stored in memory on the microcontroller and will take in the current state of the pendulum (position of the cart and angle of the pendulum) as inputs and output the required force on the cart.

The control subsystem consisted of our microcontroller, a Raspberry Pi, and analog to digital converter (ADC). We chose to use this microcontroller since it was able to deploy recent versions of tensorflow lite and hence we would be able to run neural network inferencing on it. The Raspberry Pi stored and ran the CNN as well as a backup PID controller. The ADC was used to convert the variable voltage signal from the potentiometer into a digital signal for the Raspberry Pi. The Raspberry Pi also took in the digital signal from the quadrature rotary encoder (motor encoder) and used it to determine the position of the cart for the CNN and PID controller.

The mechanical subsystem consisted of a motor controller, motor, wheel, cart chassis, pendulum, potentiometer, and quadrature rotary encoder. The motor controller took 3 inputs from the Raspberry Pi: 2 inputs for the polarity to drive the motor at and 1 input for the pulse width modulation (PWM) duty cycle. The motor was a brushed DC gearmotor with the wheel mounted on the gearmotor's drive shaft and the quadrature rotary encoder mounted on the gearmotor's output shaft. The pendulum was mounted on the potentiometer shaft so we could use the potentiometer to measure the pendulum's angular position.

The power system will consisted of a 6 V DC power supply for the motor controller and 5 V DC power supply for the Raspberry Pi. The Raspberry Pi powered the analog to digital converter and potentiometer via its 5 V output pin and powered the quadrature rotary encoder via its 3.3 V output pin.

This system design deviated from our proposed design in the two major ways. First, we decided to not use an inertial measurement unit (IMU) to measure cart velocity and acceleration as we determined through previous inverted pendulum projects that the control we desired was possible as long as we knew the angle of the pendulum. In our research, we also saw that implementing a Kalman filter would be necessary to filter out noise from the IMU signal, and integrating the IMU would add a level of complexity to our project that we did not have time to address. As we still wanted to know the position and velocity of the cart to improve the abilities of our control algorithms, we could simply use the motor encoder to get the necessary data. Second, we trained a secondary neural network to learn the state/action relationship learned by the Unity model since the original tensorflow model was unable to be deployed on our Raspberry Pi due to version incompatibility.

3. Machine Learning Module

3.1 Design Procedure

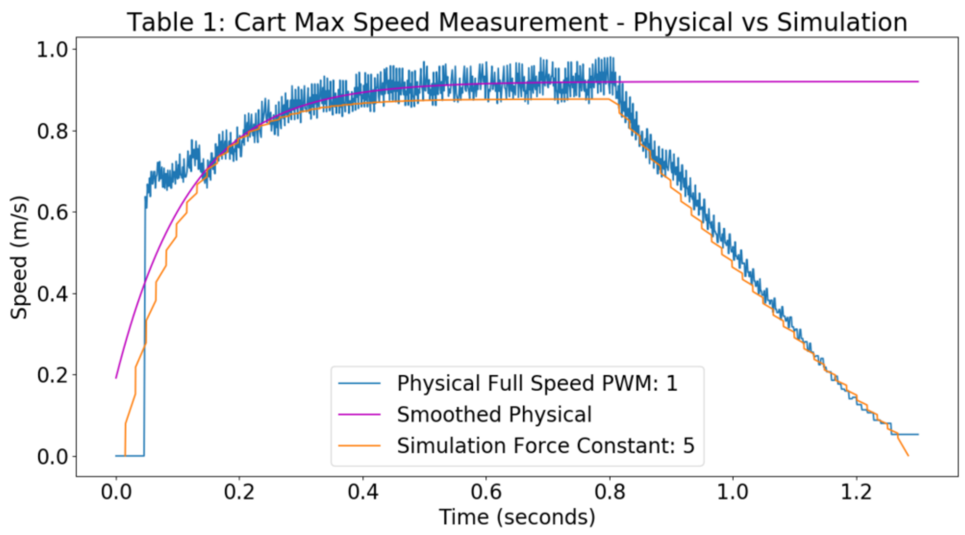
The machine learning module was developed in order to train our cart-pole system entirely in a game engine, without any experience directly in our physical system. The simulation was modeled as closely as possible to our physical system so that that learned model could successfully balance the pendulum in the physical system. Important attributes of our physical system, including the acceleration of the cart as well as the friction of our pendulum hinge were measured and modeled using Unity's physics engine. Training was carried out using a reinforcement learning algorithm, in which we were able to define the rewards of our system given the action it takes in accordance to the observed state of the system.

Post-training, we obtained a Tensorflow model which we used to run inference in the simulation to balance the pendulum given a small impulses to knock it down. Furthermore, we were able to observe the agent's learning over time using tensorboard. We originally hoped to directly the tensorflow model output from Unity to upload into our Raspberry Pi. However, due to version incompatibility, we trained a secondary CNN to learn the state/action relationship from data output from the simulation.

3.2 Unity Simulation

3.2.1 Design Details

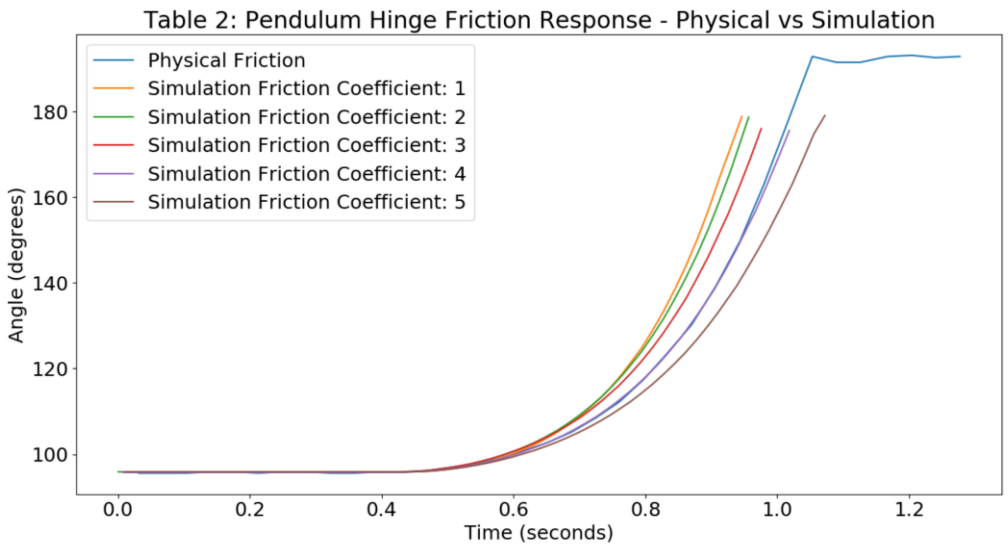
The Unity simulation was built in accordance to the important attributes of our physical system, namely the acceleration of the cart as well as the friction of our pendulum hinge. In order to find these characteristics, we carried out several experiments on our physical system.



In table one, we observe the result of the first experiment which aimed to model the acceleration of the cart. We first applied a max PWM signal to our motor and used our motor encoder to measure the change in velocity over time. We then used the scipy optimization function to fit a best fit line resulting in equation (3.1). This line was used as our baseline for how the velocity over time dynamic for our simulation motor had to function. We determine that a force constant of 5 worked best to accelerate our motor at a given speed similar to that of the physical system.

$$y = 0.7298 - 0.9196e^{-8.2844x} \tag{3.1}$$

The next experiment we carried out was measuring the hinge friction of our pendulum. This was carried out by dropping the pendulum at 5 degree, both in the physical system as well as in the simulation.



Through the plotting of the pendulum angle over time, as shown in table 2, we were able to determine that a friction coefficient of 4 best modeled the simulation pendulum in accordance to our physical system.

Once the dynamics of the system had been modeled, we moved onto defining the reward function for training the reinforcement learning model. A reward of 0.15 was given if the cart was within the center one-third of the track, 0 was given if the cart was within the center two-thirds of the track, -0.7 was given if the cart was outside the center two-thirds of the track, and finally -1.5 was given if the cart exited the bounds of the track, which restarted the simulation as well. Similarly, a reward of 0.1 was given if the pendulum angle was within 15 degrees of vertical, and -1 was given if the pendulum was further out than this angle, which restarted the simulation. The final consideration we took into account before running the training on the model was to account for the error in measurement of our system dynamics. Specifically, for the force constant, air drag constant, and hinge friction constant, a small normal noise was applied to the constants to deviate them from their original measured values. This would allow the agent to learn its policy in accordance to a normal distribution of values, hence increasing the likelihood that it would perform on a physical system which might have been imperfectly measured.

3.2.2 Verification

In order to ensure our Unity simulation works, we used the verification of whether the agent could successfully apply a force to either direction on the cart, a value in the set [-1,1] (inclusive), in order to move the cart and balance the pendulum for a test run of 10 seconds. In our final project, we also added a random impulse at a random time to the pendulum during each test run in order to make it harder for the cart to learn to balance the pendulum and since we would physically also disturb the pendulum in the physical system to see if it could still balance. While our resulting trained model could visually balance the pendulum when viewed in the Unity editor, we also verified the training was complete by viewing the performance of the agent's learning over time in tensorboard.

Table 3: Unity Simulation Agent Cumulative Reward (points) over Time (# of simulations)

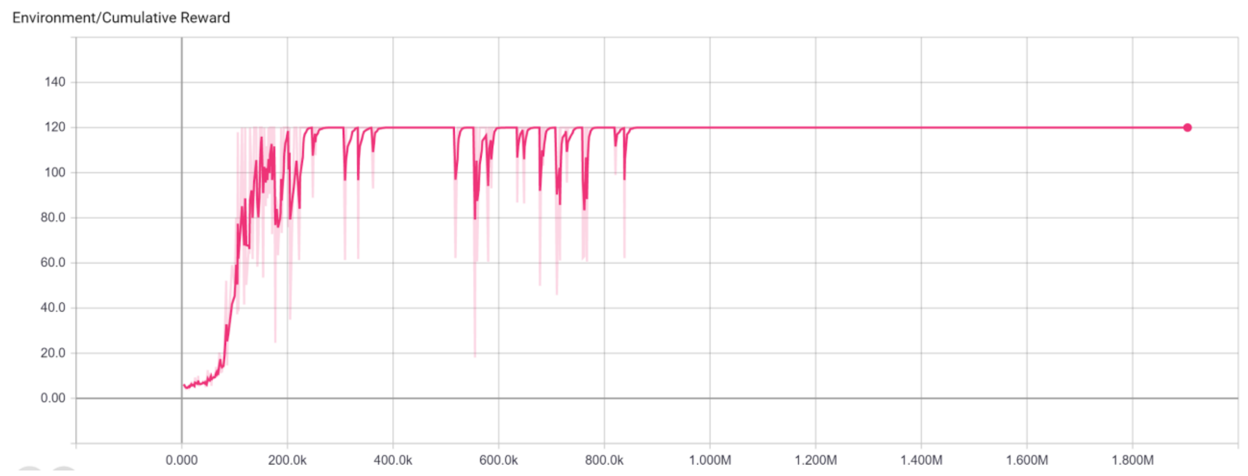
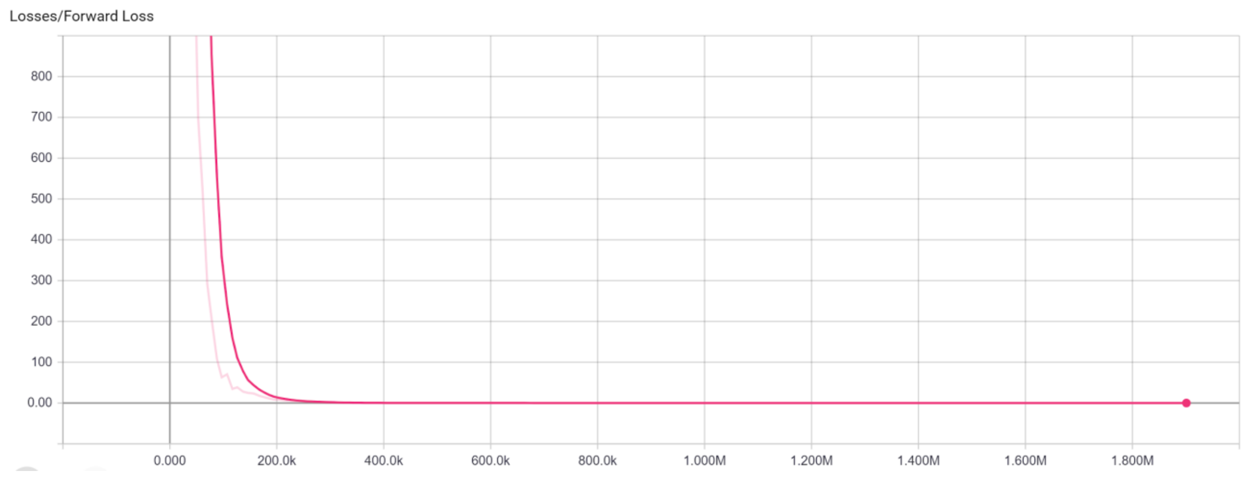
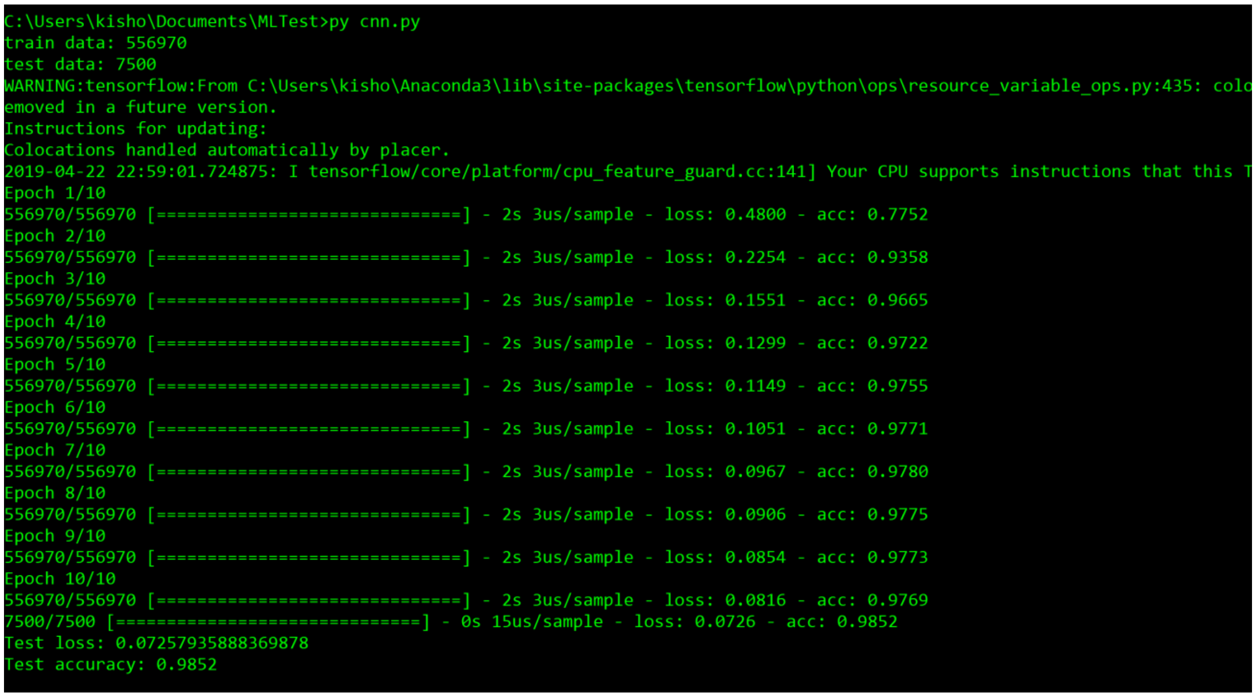


Table 4: Unity Simulation Agent Forward Loss (points) over Time (# of simulations)



In tables 3 and 4, we can see the plots for the agent's cumulative reward and loss over time. Although we ran over 1.8 million simulations for training, after around 900,000 simulations, the agent was constantly getting the full possible reward, as well as zero loss. Through this we further verified that successful training was happening, and that our Unity simulation was able to learn to balance the pendulum given disturbances to the pendulum and given random normal noise to the different measured system constants.



```
C:\Users\kisho\Documents\MLTest>py cnn.py
train data: 556970
test data: 7500
WARNING:tensorflow:From C:\Users\kisho\Anaconda3\lib\site-packages\tensorflow\python\ops\resource_variable_ops.py:435: colo
emoved in a future version.
Instructions for updating:
Colocations handled automatically by placer.
2019-04-22 22:59:01.724875: I tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions that this T
Epoch 1/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.4800 - acc: 0.7752
Epoch 2/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.2254 - acc: 0.9358
Epoch 3/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.1551 - acc: 0.9665
Epoch 4/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.1299 - acc: 0.9722
Epoch 5/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.1149 - acc: 0.9755
Epoch 6/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.1051 - acc: 0.9771
Epoch 7/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.0967 - acc: 0.9780
Epoch 8/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.0906 - acc: 0.9775
Epoch 9/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.0854 - acc: 0.9773
Epoch 10/10
556970/556970 [==============================] - 2s 3us/sample - loss: 0.0816 - acc: 0.9769
7500/7500 [==============================] - 0s 15us/sample - loss: 0.0726 - acc: 0.9852
Test loss: 0.07257935888369878
Test accuracy: 0.9852
```

Figure 3.2.2: Training of Secondary CNN to Mimic Unity Model

While Unity was able to output an adequate tensorflow model given the modeling and training through simulation, we were unable to upload the model into the Raspberry Pi, since Unity ML-Agents ran on TF version 1.7 and TFLite, the compression module for deploying tensorflow models on mobile devices, only accepted models from TF version 1.9 and higher. Furthermore, we were only able to use a CNN using the Keras API for training a secondary neural net to learn the output from the Unity model, since TFLite did not accept most RNN modules given by Keras. We were only able to teach the secondary deployed neural net a binary decision in order for which direction it should move the cart to balance the pendulum. Given further time, we would look into using a lower-level tensorflow API to teach the agent to a higher precision the action it should take to move the cart. Figure 3.2.2 shows the training of this CNN to learn the action to take given the current state from the data output from Unity. The limitations of the CNN are further described in section 8.2.

4. Power Module

4.1 Design Procedure

For our project, the power module serves as the source of power for all the electrical components in our project. In determining the right parts for this module it was important to consider what kind of power we needed. In our case, we needed a 5 V DC power supply to power our Raspberry Pi and 6V DC power supply to power our motor controller.

4.2 5V DC Power Supply

4.2.1 Design Details

For our 5 V DC power supply we chose to use a wall plug that interfaces directly with the Raspberry Pi. The 5 V and 3.3 V pins on the Raspberry Pi were then used to provide power to the potentiometer, motor encoder, and analog to digital converter. With all of the connections in place the 5 V supply should reliably give the sensors and Raspberry Pi a steady voltage and be able to supply draw 1.3 A.

4.2.2 Verification

To verify that our 5 V source was working as intended we used voltmeter to check the voltage coming out of the plug to ensure that it was receiving a constant voltage of 5 V with a very small error. From there, once we assembled the entire circuit, we needed to make sure that each individual sensor was receiving the correct voltage. This was done again by uses the voltmeter at the necessary nodes within our circuit. After testing all the relevant nodes, we determined our 5 V supply was working properly.

4.3 6V DC Power Supply

4.3.1 Design Details

For our 6 V DC power supply, we again chose to use a 6 V wall plug. We originally planned on using a 6 V battery pack, as it would have taken up less space and would be possible to mount it on the cart itself, rather than having wires extend from the outlet. However, we chose to use the wall plug DC supply, as we were already using wall plug for the 5 V source and the output from the wall plug is much more reliable in terms of consistency in the voltage reading and the current being drawn. In our design the 6 V source would be used to supply power to the motor controller, where it would be used to provide a 6 V PWM signal to the motor.

4.3.2 Verification

The process of verifying this supply is very similar to the 5 V supply. We used the voltmeter at the outputs of the supply to ensure that it was supplying a constant 6 V. We then used the oscilloscope to ensure that the motor was receiving a PWM signal with an amplitude of 6 V.

5. Mechanical Module

5.1 Design Procedure

The mechanical module contains all of the components of our project that pertain to the physical cart and pendulum system. All the mechanical pieces, such as the chassis of cart and track, were designed after a previous inverted pendulum project [7] and the CAD drawings and final design can be seen in Appendix A. The CAD files were brought to the machine shop; with which, they were able to make the the physical system. In addition to the machined parts obtained from the machine shop, the mechanical module also consists of several vital components of our project such as the motor controller, potentiometer, and motor encoder.

5.2 Motor Controller

5.2.1 Design Details

For our project we needed a motor controller that would be able to provide a variable PWM signal to our motor with a constant amplitude of 6 V, which is the rating of our DC motor. After some searching we found the VNH5019 Motor Driver which was perfect for our project, as its operating voltage range is within our desired output and its ease of interfacing with our Raspberry Pi. Its H-bridge design also makes it very easy to stop and change the direction of the motor by changing the motor A and B signals.

5.2.2 Verification

To ensure that the motor controller was working as intended we gave it a known PWM signal and motor A and B signals as an inputs and used the oscilloscope to measure the output of the controller. We were then able to verify that the motor controller was working properly as the output generated an appropriate 6 V signal with the right PWM value.

5.3 Potentiometer

5.3.1 Design Details

The potentiometer within the context of our project serves as an angle sensor. For a potentiometer to be used for this application it is important that it is only single turn as we only care about reading values for less than one revolution of the pendulum. It is also very important that the voltage readings across the potentiometer are relatively linear with respect to the angle so we can accurately calculate the the angle of the pendulum based on the voltage across the potentiometer.

5.3.2 Verification

To ensure that the potentiometer provided a linear voltage reading, we moved the pendulum to various angles around the region of its operation and recorded the voltage across the potentiometer at each of these angles. The resulting curve from this experiment turned out to be very linear near its main operating point near the middle and slightly less linear as the potentiometer was closer to its maximum and minimum values. This turned out to be completely acceptable for our project, because we only operate the pendulum at angles very close to the equilibrium, so we only need accurate measurements in this region.

5.4 Quadrature Motor Encoder

5.4.1 Design Details

To calculate the cart's position and velocity we used a quadrature motor encoder. The encoder is attached the motor's output shaft and consists of 6 north and south pole magnets. The encoder contains 2 magnetic sensors that are offset by 90 degrees. These sensors output a high signal when the north pole of the magnet is near it and outputs a low signal when the south pole is close to it. We can then determine the direction the wheel is rotating based on which sensor leads the other. It is also possible to determine the revolutions per minute of the wheel using this sensor by counting the number pulses in a given time frame and then multiplying by the gearbox ratio of the output shafts and the number of north poles on the magnetic encoder. The velocity of the cart can then be determined by converting the units into m/s by calculating the wheel's circumference.

5.4.2 Verification

To verify that we were obtaining accurate measurements for the position and velocity of the cart we ran several tests. To test the position sensor we started the cart in a stationary position and moved it a known distance on the track. To test the velocity sensor we supplied the motor with a constant PWM signal. We had a script record the velocity using the algorithm as explained above. We then confirmed that it was working properly by manually counting the revolutions of the wheel during the interval and comparing the manual calculation for rpm with the scripts output.

6. Control Module

6.1 Design Procedure

The control module's primary goal was to determine the PWM signal and polarity to apply to the motor in order to move the cart left or right and a specific speed. The state of the system was read through the potentiometer which measured the angle of the pendulum, and the motor encoder which recorded the position of the cart on the track. After our Raspberry Pi processed the current state, the control module decided whether to use to PID controller or the tensorflow model stored in memory to balance the pendulum. Our original goal was for the control module use the neural net to balance the pendulum if the angle was within five degrees of vertical, and PID controller if the angle was within 20 degrees of vertical. However, since our neural net was only able to output a binary decision leading to minimal oscillatory control, we primarily used the PID controller for the best performance on the system.

6.2 Raspberry Pi

6.2.1 Design Details

We used a Raspberry Pi for our microcontroller since the Tensorflow Lite framework allowed TF models to be frozen and deployed on mobile devices which included Raspberry Pi. The inputs for the microcontroller were the readings through the GPIO pins from our motor encoder and potentiometer, which determined the position of the cart, as well as the angle of the pendulum.

6.2.2 Verification

One of the requirements of our microcontroller was that it must be able to read the state of the system through the sensors, as well as account for the computation lag of the neural network before it decided which action to take. We were able to solve this problem by introducing a lag to the agent while it was training in the simulation. First we deployed a dummy tensorflow model into the Pi and measured how long it took to compute a value after it was first given an input. When we determined the value was around 10ms, we used this as the introduced lag for the agent in the simulation. Our verification for this problem was whether the Unity simulation could be trained to balance the pendulum given the computation lag on the Pi. After introducing the lag during the training of the simulation, however, the agent was successfully able to learn to balance the pendulum in Unity. This indicated that the same network would be able to perform just as well if it was deployed on the Pi.

6.3 Analog to Digital Converter

6.3.1 Design Details

The analog to digital converter was used in our project to interface the analog voltage signal of the potentiometer to the digital GPIO pins on the Raspberry Pi. For this purpose the Adafruit MCP3008 ADC worked perfectly, as there are existing libraries for the Raspberry Pi for this interface. The 10 bit ADC also provides decent precision as it can determine the right angle within .35 degrees.

6.3.2 Verification

To verify that the analog to digital converter was working properly we connected the potentiometer to one of its serial inputs. We then moved the potentiometer to several known angles and observed the ADCs output. As stated above the ADC was able to distinguish the angle within .35 precision.

6.4 PID Controller

6.4.1 Design Details

The PID controller works as a backup controller for the pendulum, cart system when the tensorflow model fails to balance the cart. The reason we chose to implement this controller to our project is because we wanted some form of security that would be more likely to stabilize the system in case the tensorflow model became unstable. The PID controller works by tuning several gains for proportional, integral, and derivative gains that amplify the error signal to output a PWM signal to the motor. The proportional gain takes the error signal, the current angle subtracted from the equilibrium angle, and just multiplies it by a constant gain. The integral gain multiplies a constant gain by the addition of the current error signal and the sum of all the past error signals. The derivative gain multiplies a constant gain by the difference between the last error signal and the current error signal.

To tune these parameters of the PID controller we used the Ziegler-Nichols tuning method [10]. This method involved initially setting all parameters to 0, then increase the P gain until the system becomes oscillatory. The method than sets this P gain equal to K_u and the oscillation period to T_u. The other PID parameter can be calculated from just these values. For classical PID control the equations are below.

$$P = K_u \times 0.6P$$
$$I = 1.2K_u/T_u$$
$$D = 3K_uT_u/40$$

From these equation we were able to get a good starting point for the parameters of our controller, but since this is just a heuristic and not a fool proof solution for all PID applications we still needed to fine tune the parameters to obtain a controller that operated how we wanted it to.

6.4.2 Verification

To verify how our PID controller works we needed to determine where the system converges. In order to determine the cases in which our system converges and at what point it diverges, we ran several tests on our system running the PID controller. While the controller was active we made sure to record the data for the current angle and PWM signal that was supplied to the system. We will assume that the system converges if it reaches an equilibrium point and the velocity and angle does not change until we supply another addition disturbance to the pendulum. We will say the system diverges when the pendulum can no longer reach the equilibrium on the given length of track. From the data we collected we determined that the system can reach this equilibrium if the disturbance produces a PWM signal that is below around 65% and the angle is constrained to within 3.5 degrees of the equilibrium position. If the disturbance results in an error of less than 3.5 degrees the controller tends to be able to stabilize itself to balance the pendulum. If the angle crosses that threshold the controller tends to force the cart to the endpoints of the track before being able to balance the pendulum. Below is a graph of one of these trials that shows this behavior. For the first few disturbances we gave to the system it was still able to balance itself, as the angle and PWM percentage was held within the bounds. At the end however, we produced a disturbance at which the pendulum could no longer balance. The angle of the system at the point of the final disturbance was around 3.5 degrees from the equilibrium which produced a max PWM percentage of around 72.5% which lead to the divergence of the system.

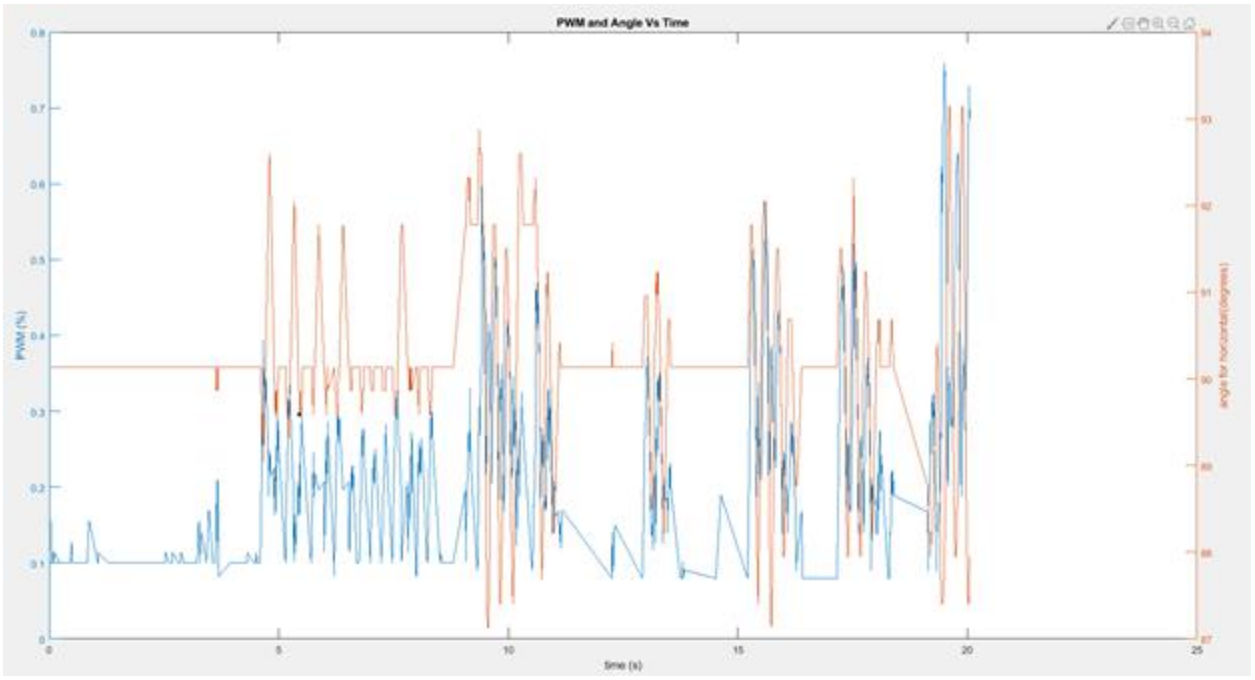


Figure 6.4.1 Graph of PWM and Angle vs Time

7. Costs

7.1 Cost of Parts

The total cost of parts for our final unit was $176.52. A detailed list of parts is in Appendix B.

7.2 Cost of Labor

The cost of labor was calculated as follows: 3 people working 10 hours per week each for the 16 week semester at a wage of $35.00 per hour per person.

$$\text{cost of labor } = 3 \text{ people} \cdot 12 \frac{\text{hours}}{\text{week}} \cdot 16 \text{ weeks} \cdot \frac{\$35.00}{\text{hour}} \cdot 2.5 = \$50{,}400$$

7.3 Total Cost

The total cost of this project was the cost of parts for one completed unit plus the cost of labor. This brings the total cost of our project to $50,576.52.

8. Conclusion

8.1 Accomplishments

Overall, we were able to accomplish a several of the key tasks as well as set up a framework for further work at the final state of our project. Firstly, we were able to create a fully functional PID control system, which was able to balance the pendulum up to angles of 3.5 degrees from vertical given disturbances. This control system had only a single loop, which took in the angle of the pendulum and applied the control equation to output the correct PWM signal to the motor for balancing. Secondly, we were able to create a framework for modeling a physical system and training it in a Unity simulation. Through running tests to determine the dynamics of important physical attributes of the system, we were able to determine the constants needed to most accurately represent these dynamics in Unity. Furthermore, we were able to training the resulting simulation given a random normal noise to the constants, ensuring that the agent would be able to perform over a distribution of different values, hence making it more resilient to deviations in the dynamics of the physics system. Finally, we were able to train a secondary CNN to learn to roughly mimic the originally trained model given the data output from the Unity simulation. We were able to successfully deploy this secondary network onto our Raspberry Pi for minimal control on our physical system.

8.2 Uncertainties and Future Work

The main uncertainty and limitation of our project is the accuracy and precision of our secondary neural network. Our secondary neural network was not able to function up to the standard set by the output of the Unity simulation. The Unity tensorflow model was able to read 2 floating point numbers, the position of the cart and the angle of the pendulum, and output a floating point number which was between -1 and 1 and precise up to four decimal points. Unfortunately, due to Tensorflow Lite's incompatibility with RNN Keras modules, we only had time to use CNN Keras modules, which were more basic and only achieved a 60 percent accuracy predicting the output action rounded to one decimal point. Since this was insufficient to deploy on the physical system, we simply rounded to the nearest integer, -1 or 1, as the value the CNN should output. While this trained up to a 98 percent accuracy (Figure 3.2.2), it allowed limited oscillatory control and has much room for improvement. Further work to improve this would be to develop an RNN using the basic tensorflow API, which would more accurately be able to output a precise floating point output give the current state as well as the past few state of the system. Additionally, the basic tensorflow implementation would be much more compatible with tensorflow lite for deployment on the Raspberry Pi.

8.3 Ethical Considerations

The ethical and safety issues in our project mainly pertain to the safety of the different moving mechanical components in the design. Since there is movement of a cart on a track and the attached swinging pendulum, this poses a safety risk for anyone standing too close or putting their body parts in the system. During operation, the cart and pendulum could cause injury to someone too close to the system. Additionally, our electrical components could cause harm to someone upon being mishandled.

The safety precautions we would take to handle these situations refer to #9 on the IEEE Code of Ethics, to “avoid injuring others, their property, reputation, or employment by false or malicious action“[4]. To prevent the cart from hurting someone’s fingers, we used a lightweight aluminum chassis with blunt ends, as well as bumpers to the end of the track to prevent the cart from flying off. We also ensured that the heavy end of the pendulum was a rounded cylinder and not sharp, to avoid the risk of serious injury if it was to strike anyone standing too close. We also had a safety mechanism to prevent any outputs of the control system which ask for too voltaged by clipping the signals at the max allowed PWM value. In some situations where the pendulum angle was too far from vertical to balance, we cut the signal from the motor completely. In addition, since we were dealing with relatively high voltages to power the motor, we made sure the circuits were properly insulated so that no can get hurt by accidentally coming into contact with the system.

References

[1] Clark, Jack. "Robots That Learn." *OpenAI Blog*, OpenAI Blog, 28 Nov. 2017, blog.openai.com/robots-that-learn/.

[2] Juliani, Arthur. "Introducing: Unity Machine Learning Agents Toolkit – Unity Blog." *Unity Technologies Blog*, 19 Sept. 2017, blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/.

[3] OpenAI. "Proximal Policy Optimization." *OpenAI Blog*, OpenAI Blog, 20 July 2017, blog.openai.com/openai-baselines-ppo/.

[4] "IEEE Code of Ethics." *IEEE - Advancing Technology for Humanity*, www.ieee.org/about/corporate/governance/p7-8.html.

[5]"Average Entry-Level Electrical Engineer Salary", Payscale, www.payscale.com/research/US/Job=Electrical_Engineer/Salary/6fd28da9/Entry-Level

[6] "TensorFlow Lite | TensorFlow." *TensorFlow*, www.tensorflow.org/lite.

[7] "Inverted Pendulum Project." *Andy's Log*, 23 Jan. 2016, andreweib.wordpress.com/inverted-pendulum-project/.

[8] Unity-Technologies. (n.d.). Unity-Technologies/ml-agents. Retrieved from https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Feature-Memory.md

[9] Friction and Friction Coefficients. (n.d.). Retrieved from https://www.engineeringtoolbox.com/friction-coefficients-d_778.html

[10]The Michigan Chemical Process. (10/16/2007). PID Tuning Classical. Retrieved from https://web.archive.org/web/20080616062648/http://controls.engin.umich.edu:80/wiki/index.php/PIDTuningClassical#Ziegler-Nichols_Method

Appendix A: CAD Models, Schematics, and Layouts

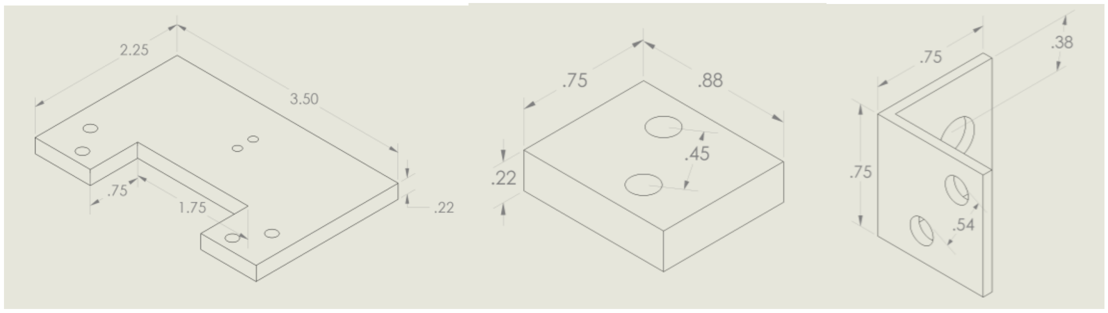


Figure A1: Cart Chassis, Rail Spacer, and Track Holder

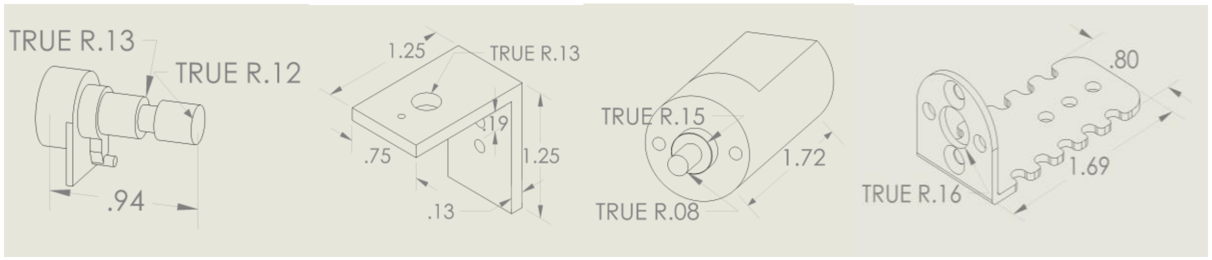


Figure A2: Potentiometer, Potentiometer Holder, Motor, and Motor Bracket

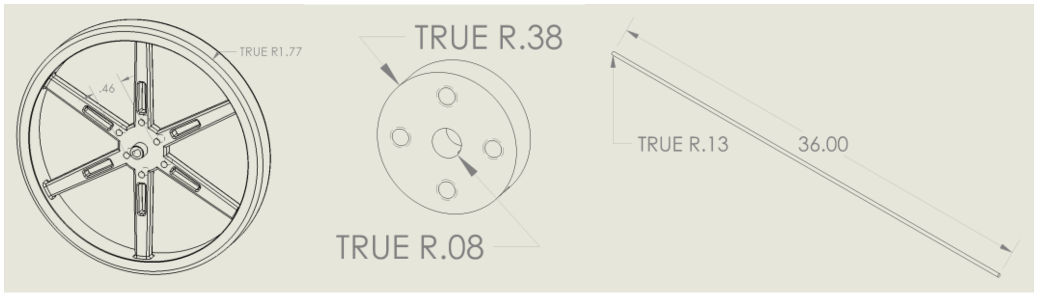


Figure A3:, Wheel, Wheel Hub, and Track

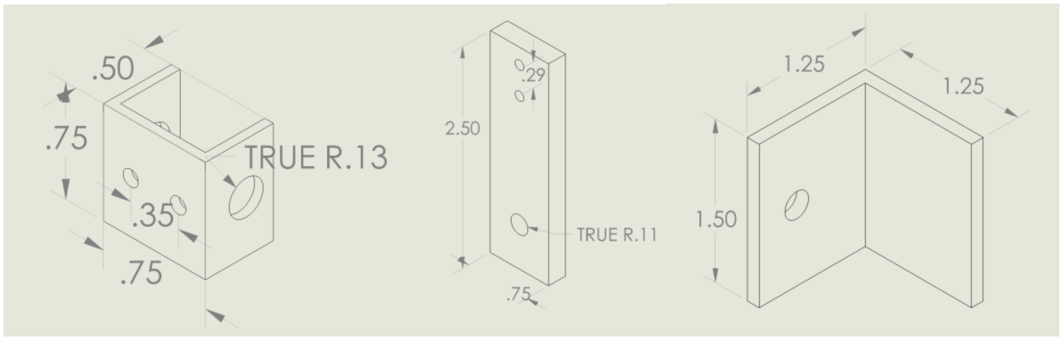


Figure A4: Pendulum Holder, and Potentiometer Connector, and Outer Track Holder

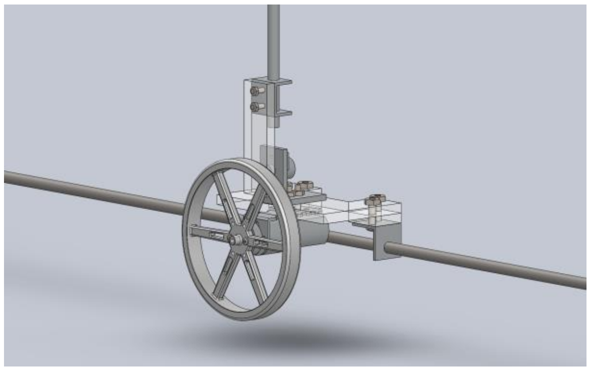


Figure A5: Prototype 3D Model

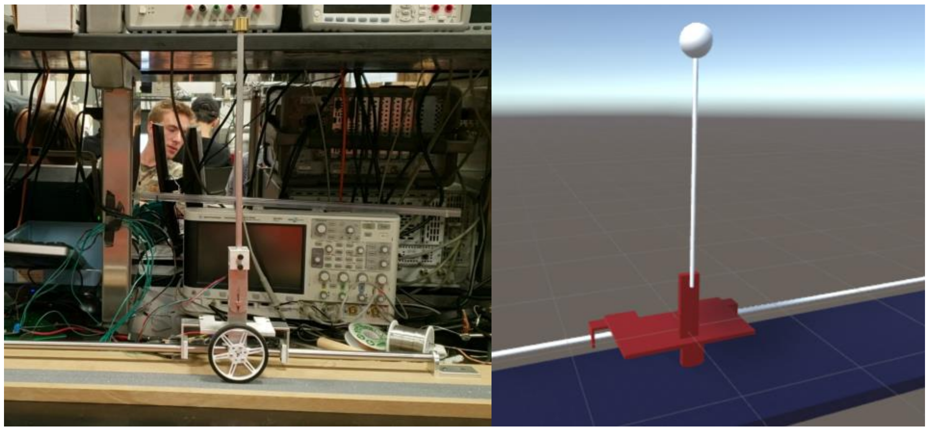


Figure A6: Final Physical Design and Unity Simulation

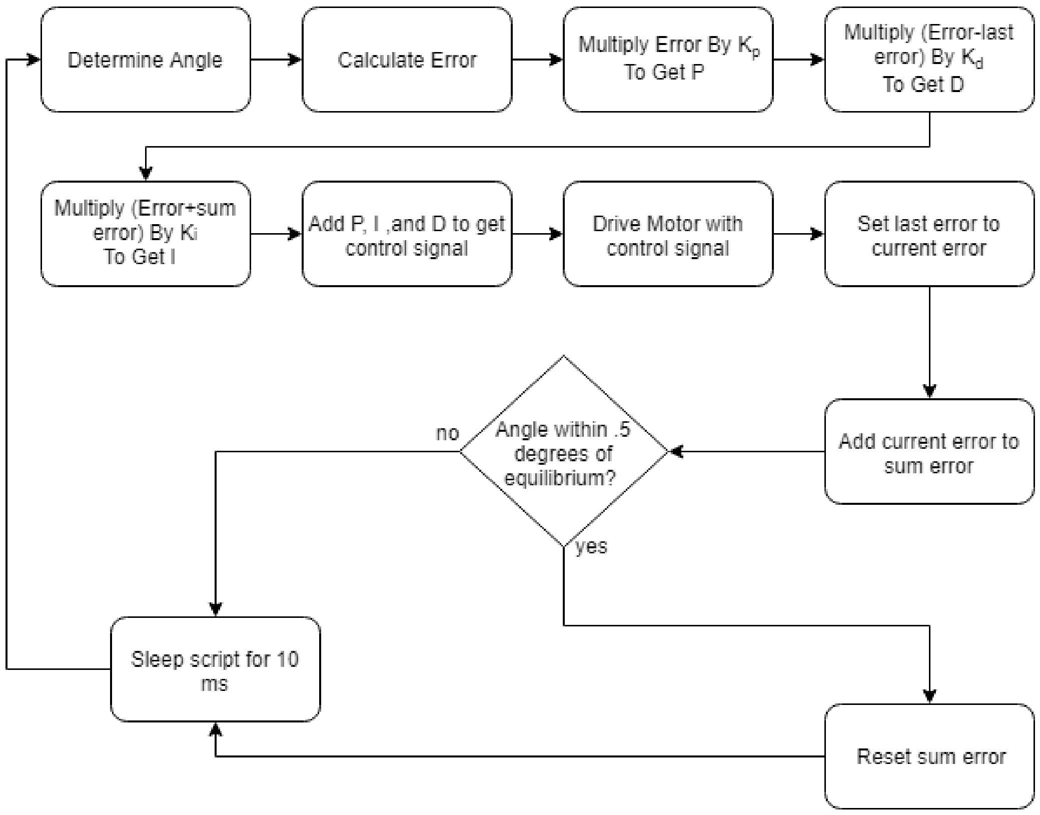

Determine Angle
Calculate Error
Multiply Error By K_p To Get P
Multiply (Error-last error) By K_d To Get D
Multiply (Error+sum error) By K_i To Get I
Add P, I ,and D to get control signal
Drive Motor with control signal
Set last error to current error
Add current error to sum error
Angle within .5 degrees of equilibrium?
no
yes
Sleep script for 10 ms
Reset sum error

Figure A7: PID Software Flowchart

Appendix B: Parts List

Part Description	Part	Cost ($)
6V DC motor	Pololu 63:1 Metal Gearmotor 20Dx43L mm 6V CB with Extended Motor Shaft, #3714	22.95
Motor Encoder	Pololu Magnetic Encoder Pair Kit, #3499	8.95
Motor Brackets	Pololu 20D mm Metal Gearmotor Bracket Pair, #1138	6.95
Motor Driver	Pololu Dual VNH5019 Motor Driver Shield Pololu VNH5019 Motor Driver Carrier, #1451	24.95
Wheel	Pololu wheel (2 pack), #3691	7.95
Wheel Mount	Pololu Universal Aluminum Mounting Hub for 4mm Shaft, 2 Pack, #1081	6.95
6V Power supply	Triad Magnetics WSU060-3000 6 V, 3 A Power Supply	12.45
Raspberry Pi	Raspberry Pi 3 with 2.5A Micro USB Power Supply	42.99
Potentiometer	5 kΩ Single Turn 3 Pin Rotary Taper Potentiometer	12.36
Analog to Digital Converter	Adafruit MCP3008	7.49
Grip Tape	Pusdon Anti Slip Non Skid Safety Tape, Gray, 2-Inch x 15Ft (51mm x 4.75m)	7.95
Cart, rail, pendulum, mount	Custom built by ECE Machine Shop	0
	Total, with 9% Sales Tax	176.52

Appendix C: Core Programs

C1. PID Controller

```
from gpiozero import MCP3008
from gpiozero import PWMOutputDevice
import RPi.GPIO as GPIO
from gpiozero import OutputDevice
from read_RPM import reader
import time
import pigpio
import signal
import sys

GPIO.setmode(GPIO.BCM)

pot = MCP3008(0)

PWM = PWMOutputDevice(15)

a = OutputDevice(14)
b = OutputDevice(18)

kp = .1
kd = .06
ki = .015

PID = 0
error = 0
last_error = 0
sum_error = 0
angle = 0

multValue = ((0.7957987298485589-0.15192965315095264)/180)
setpoint = (pot.value - 0.15192965315095264)/multValue
print(setpoint)

def Drive_motors(control):
    if control>0:
        a.on()
        b.off()
        if abs(control) > 1:
            PWM.value = 1
        else:
            PWM.value = abs(control)
    elif control<0:
        a.off()
        b.on()
        if abs(control) > 1:
            PWM.value = 1
        else:
            PWM.value =  abs(control)
    else:
        PWM.value = 0;

def signal_handler(sig, frame):
    PWM.value = 0
    a.off()
    b.off()
    sys.exit(0)
signal.signal(signal.SIGINT, signal_handler)

while True:
    angle = (pot.value - 0.15192965315095264)/multValue
    error = setpoint - angle
    P = error*kp
    D = kd*(error-last_error)
    sum_error = sum_error + error
    I = ki*(sum_error)
    PID = P+I+D
    Drive_motors(PID)
    last_error = error
    if abs(error) < .6:
        sum_error = 0
    #print('Setpoint: ' + str(setpoint) + ", angle: " + str(angle) + ', Error: ' + str(error)+ ", I control: " +
str(I))
    time.sleep(.01)
```

C2. Tensorflow to Raspberry Pi Interface

```
import random
import numpy as np
import time
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

num_files_to_read = 150
LR = 0.01

MODEL_NAME = 'cartModel-{}.model'.format('2conv-basic') # just so we remember which saved
model is which, sizes must match




def create_training_data():
        lines = []
        headers = ["target","current","output"]

        for i in range(2,num_files_to_read):
                with open(f"./agent_content/agent_content{i}.txt", "r") as filestream:
                        for line in filestream:
                                currentline = [float(item) for item in line.split(",")]

                                action = 1 if (currentline[0]- currentline[1]) > 0.0 else 0

                                lines.append([np.array(currentline[:2]), action])
                                #lines.append(currentline)



        #print(lines)
        random.shuffle(lines)

        train_x = []
        train_y = []

        test_x = []
```

```
	test_y = []


	for i in range(len(lines)-500): #go over new sequential data
		seq, target = lines[i]
		train_x.append(seq)
		train_y.append(target)

	for i in range(len(lines)-500, len(lines)): #go over new test data
		seq, target = lines[i]
		test_x.append(seq)
		test_y.append(target)

	return np.array(train_x), train_y, np.array(test_x), test_y






train_x, train_y, test_x, test_y = create_training_data()

print(f"train data: {len(train_x)}")
print(f"test data: {len(test_x)}")




model = Sequential([
    Dense(128, input_shape = (2,), activation=tf.nn.relu),
	Dense(32, activation=tf.nn.relu),
    Dense(2, activation=tf.nn.softmax)
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


model.fit(train_x, train_y, epochs=5)



test_loss, test_acc = model.evaluate(test_x, test_y)
```

```
print('Test loss:', test_loss)
print('Test accuracy:', test_acc)


# Save tf.keras model in HDF5 format.

keras_file='models/HDF5_Models/cnn_model.h5'

model.save(keras_file)

converter = tf.lite.TFLiteConverter.from_keras_model_file(keras_file)
converter.target_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,tf.lite.OpsSet.SELECT_TF_OPS]
converter.post_training_quantize=True
tflite_model = converter.convert()
open("models/TFLite_files/quantized_cnn_model.tflite", "wb").write(tflite_model)
```

C3: Unity Simulation Reinforcement Learning

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using MLAgents;
using UnityEngine.SceneManagement;
using System.Diagnostics;
using System.Linq;
using System.Timers;
using System;

public class SystemAgent : Agent
{



    public Rigidbody pendulum_body;
    public Rigidbody cart_body;

    public int count;

    System.Random rnd;

    public int episode;
    public List<string> lines;

    //constants
    public float origHingeConstant = 4f;
    public float hingeConstant;

    public float origSpeedConstant = 5f;
    public float speedConstant;

    public float origDragConstant = 14f;
    public float dragConstant;


    public float motorDrag = 0.015f;
    public float pwm;
    public float lastForce;

    //UPDATED VARS

    public GameObject current_system;
    public GameObject system_prefab;
    Vector3 cart_orig_loc;

    public GameObject pendulum;
    public GameObject motorAgent;
    public GameObject wheel;
    public GameObject rail;

    public float maxVel = 0;
    public int lag = 0;
    public int impulseTime;
    public bool fallen;

    public Stopwatch timer;


    // Use this for initialization
    void Start()
    {
        //cart_body = GetComponent<Rigidbody>();
        //pendulum_body = pendulum.GetComponent<Rigidbody>();

        // print("fixed time unit: " + Time.fixedDeltaTime);

        cart_orig_loc = new Vector3(0f, 0f, 0f);



        count = 0;
        episode = 0;



        rnd = new System.Random();

        current_system = Instantiate(system_prefab, cart_orig_loc, Quaternion.identity);
        //pendulum =
current_system.GetComponentInChildren<Transform>().Find("Pendulum").gameObject;
```

```
        //motorAgent =
current_system.GetComponentInChildren<Transform>().Find("MotorAgent").gameObject;


        fallen = false;

        timer = new Stopwatch();

        timer.Start();

    }

    public override void AgentReset()
    {

        //impulse will happen in random time between 4-8 seconds
        impulseTime = (int) (rnd.NextDouble()*400 + 400);

        //break current prefab
        Destroy(motorAgent);
        Destroy(pendulum);
        Destroy(current_system);

        //create current system
        current_system = Instantiate(system_prefab, cart_orig_loc, Quaternion.identity);

        motorAgent = current_system.transform.GetChild(0).gameObject;
        pendulum = current_system.transform.GetChild(1).gameObject;



        cart_body = motorAgent.GetComponent<Rigidbody>();
        pendulum_body = pendulum.GetComponent<Rigidbody>();


        //set the constants with normally distributed noise

        double u1, u2, randStdNormal;
        //speedConstant - varys between 4.5-5.5, orig is 5
        u1 = 1.0 - rnd.NextDouble();
        u2 = 1.0 - rnd.NextDouble();
        randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0 * Math.PI * u2); //random
normal(0,1)

        speedConstant = (float)(origSpeedConstant + (0.2 * randStdNormal));

        //dragConstant - varys between 13-15, orig is 14
        u1 = 1.0 - rnd.NextDouble();
        u2 = 1.0 - rnd.NextDouble();
        randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0 * Math.PI * u2); //random
normal(0,1)

        dragConstant = (float)(origDragConstant + (0.5 * randStdNormal));
        cart_body.drag = dragConstant;

        //hingeConstant - varys between 3-5, orig is 4
        u1 = 1.0 - rnd.NextDouble();
        u2 = 1.0 - rnd.NextDouble();
        randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) * Math.Sin(2.0 * Math.PI * u2); //random
normal(0,1)

        hingeConstant = (float)(origHingeConstant + (0.5 * randStdNormal));
        pendulum_body.drag = hingeConstant;



        print("created system:" + episode);




        string[] content = lines.Select(i => i.ToString()).ToArray();


//System.IO.File.WriteAllLines(@"C:\Users\kisho\Documents\MLTest\pendulum_agent_content\age
nt_content" + episode + ".txt", content);

        //print("episode: " + episode + ", lines written: " + lines.Count);

        lines = new List<string>();
        episode += 1;
        count = 0;
        fallen = false;

        timer = new Stopwatch();
        timer.Start();

    }

    public override void CollectObservations()

    {

            AddVectorObs(pendulum.transform.eulerAngles.x);
            //AddVectorObs(pendulum.GetComponent<Rigidbody>().angularVelocity);

            AddVectorObs(motorAgent.transform.position.z);
            //AddVectorObs(motorAgent.GetComponent<Rigidbody>().velocity.z);

    }

    public override void AgentAction(float[] vectorAction, string textAction)
    {

        //cart: 336g
        //pendulum: 140g
        //pendulum: 140setg
        //pend_bracket: 40g
        //
        //mlagents-learn config/trainer_config.yaml --run-id=pend_lstm_1 --train


        string new_state = pendulum.transform.eulerAngles.x + "," + motorAgent.transform.position.z
+ "," + vectorAction[0];

        print("Angle: " + pendulum.transform.eulerAngles.x + ", Position : " +
motorAgent.transform.position.z + ", Action: " + vectorAction[0]);


        //add new state to recording
        lines.Add(new_state);
        //Control Signal -> (0,0,-1) or (0,0,1)
        //Max speed of cart with speed scale 0.005f: 0.866 m/s
        //Max speed of cart in physical system: 0.88 m/s -> 0.73 m/s

        //delay the actions to take place every 0.02 seconds (since time step + NN processing takes 0.02
seconds)


        if (lag == 1) {

            //get action from agent and apply force to cart
            pwm = vectorAction[0];
            lastForce = speedConstant * (1f - (float)Math.Pow(1 - pwm, 2f));
            Vector3 controlSignal = new Vector3(0f, 0f, lastForce);

            cart_body.AddForce(controlSignal);

            lag = 0;


            //motor torque drag, for when motor is turned off
            if (Math.Abs(lastForce) < 0.0001)
            {
                cart_body.drag = 0;
                if (cart_body.velocity.z >= 0.03)
                    cart_body.velocity = new Vector3(0f, 0f, cart_body.velocity.z - motorDrag);
                else if (cart_body.velocity.z <= -0.03)
                    cart_body.velocity = new Vector3(0f, 0f, cart_body.velocity.z + motorDrag);
            }
            else {
                cart_body.drag = dragConstant;
            }


        }
        else
        {
            lag += 1;
        }
        //apply an impulse between 4 and 8 seconds
        if (count == impulseTime)
        {

            int disturbance = rnd.Next(-5,6);

            float applied = disturbance * 0.01f;

            //print(applied);


            //pendulum.GetComponent<Rigidbody>().AddForce(new Vector3(0f, 0f, disturbance *
intensity), ForceMode.Impulse);
            //pendulum.GetComponent<Rigidbody>().AddForce(new Vector3(0f, 0f, applied),
ForceMode.Impulse);
            pendulum.GetComponent<Rigidbody>().AddForce(new Vector3(0f, 0f, -0.01f),
ForceMode.Impulse);

            // print("force applied: " + applied);
            //print("pendulum V: " + pendulum.GetComponent<Rigidbody>().velocity.z);

        }

            count += 1;




        //print("velocity: " + cart_body.velocity.z);
        if(cart_body.velocity.z > maxVel)
        {
            maxVel = cart_body.velocity.z;
            //print(maxVel);
        }



        //if cart_body goes too far, reset
        //between 0 - 0.15, reward = 0.1
        //between 0.15 and 0.25, reward = 0
        //between 0.25 and 0.36, reward = -1
```

```
//past 0.36, reward = -1.5

if (Mathf.Abs(motorAgent.transform.position.z) > 0.36)
{

   Destroy(motorAgent);
   Destroy(pendulum);
   Destroy(current_system);


   //print("destroyed system for POSITION");

   //bad episode
   lines = new List<string>();

   SetReward(-1.5f);
   Done();
}
else if (Mathf.Abs(motorAgent.transform.position.z) > 0.25) {
   SetReward(-0.7f);
}
else if (Mathf.Abs(motorAgent.transform.position.z) <= 0.15)
{
   SetReward(0.1f);
}


//float difference = Mathf.Abs(pendulum.transform.position.z - transform.position.z);


//Conservative approach
//if within 25 degrees of vertical, reward = 0.1
//else, reward = -1.0
if (pendulum.transform.eulerAngles.x < 300 && pendulum.transform.eulerAngles.x > 60)
{


   Destroy(motorAgent);
   Destroy(pendulum);
   Destroy(current_system);

   //print("destroyed system for FALLING");
```

```
            //bad episode
            lines = new List<string>();

            SetReward(-1.0f);
            Done();

        }
        else
        {
            SetReward(0.1f);

        }

    }

}
```