Reinforcement Learning for the Petoi Bittle in an Embedded Systems Laboratory

PROTOTYPING AND TESTING MACHINE LEARNING IN AN EMBEDDED APPLICATION

sdmay22-45
Adviser: Dr. Rover
Team Members/Roles:
Amy Wieland - Project Manager
Tyler Ingebrand - Project Manager and Machine Learning Manager
Nathan Bruck - External Hardware/Arduino Manager
Yi Ting Liew - Task Board Manager
Sean McFadden - Machine Learning Manager
Nayra Lujano - Research Manager
Chris Hazelton - Security Manager
Team Email: sdmay22-45@iastate.edu
Team Website: https://sdmay22-45.sd.ece.iastate.edu

Table of Contents

1 Introduction **4**

1.1 Problem Statement 4

2 Design **4**

2.1 Revised Project Design 4

2.2 Requirements & Constraints 4

2.3 Engineering Standards 5

2.4 Security Concerns and Countermeasures 6

2.4.1 Physical Security 6

2.4.2 Cyber Security 6

3 Implementation **6**

4 Testing **9**

4.1 Testing Process 9

4.2 Unit Testing 9

4.3 Interface Testing 9

4.4 Integration Testing 9

4.5 Acceptance Testing 10

4.6 Results 10

5 Related Products & Literature **10**

5.1 Prior Work/Solutions 10

6 Closing Material **11**

6.1 Conclusion 11

6.2 References 11

6.3 Appendices 12

6.3.1 Appendix I - Operation Manual 12

6.3.2 Appendix II - Alternative Versions 12

6.3.3 Appendix III - Other Considerations 12

6.3.4 Appendix IV - Code 17

6.3.5 Appendix V - Lab Module Example 19

List of Figures

Figures

Figure 1. Actor Critic Method 7

Figure 2. Petoi Bittle with Shoes 17

Figure 3. Component Diagram 17

Figure 4. Conceptual Design Diagram 18

1 Introduction

1.1 Problem Statement

The main goals of our project are to successfully demonstrate embedded machine learning on an interesting application and to make recommendations for incorporating embedded machine learning in a course for the CPR E department.

Our team has decided to train a robot to walk via reinforcement learning as our chosen application for our embedded system machine learning project. The machine learning algorithms that are chosen must be applicable to the robot and, thus, will be restricted to the resources of the robot's embedded system. We plan to utilize a combination of virtual environment and physical environment to train the robot throughout the project time duration.

2 Design

2.1 Revised Project Design

Over the course of this semester, our project design has remained, for the most part, consistent with the planned project design developed last semester. The main change to our initial project design was to utilize PyTorch over TensorFlow as part of the learning testbed used to train the robot dog. Initially, TensorFlow was selected over PyTorch because several group members already had previous experience with TensorFlow. However, after implementation began, there was an issue we ran into with TensorFlow, and we were having trouble working around it. We tried to see if we could get something working with PyTorch code, and it produced better results. As we experimented with PyTorch more, we discovered that it was easier to use compared to TensorFlow. With the switch to TensorFlow, we also switched to using PyBullet instead of Mujoco. PyBullet and Mujoco are both physics engines that help simulate how the robot will move. PyBullet offered better integration with PyTorch, and Mujoco was giving issues with loading the URDF file that defined the elements of our virtual robot.

2.2 Requirements & Constraints

Our functional requirements include training the robot virtually using the OpenAI Gym and PyBullet toolkit. Virtual training allows us to utilize more computing power for training, and the physical components of the robot will take no wear or damage. Another requirement is for the inference process to be done locally on the robot. Additionally, the robot should be able to walk stably without having to send information to an external device to process the data. As far as walking, the goal is for the robot to walk at a speed of 0.19ft/s which is the same walking speed the robot has using its default controller walking.

Another type of requirement the team needs to be aware of is the different types of resource and non-functional requirements the project has. First, the team will use the Petoi Bittle Robot Dog as the test bed for our embedded machine learning application. The reasoning for using the Petoi is because it is easily repairable, and we wanted to be able to have a usable device that could easily be repaired and maintained in working condition. With those considerations in mind, the Petoi Bittle is the chosen platform to use for our

application, hence it is a resource we will require. Secondly, we need to ensure that we are using a platform that supports the embedded system usage we are aiming for. Because of this, we will require a resource such as a Raspberry Pi or a type of Microcontroller that allows us to program the Petoi Bittle Robot Dog. With these two resources in mind, we will also need to keep our system somewhat modular. This will allow for the application to be scalable and practical for student projects or other classroom and university applications. Lastly, one of the other most important resource requirements we've identified is being able to use resources a university would have access to for course implementation. This includes using affordable and accessible components and using systems and coding languages the university has access to. We plan to utilize OpenAI Gym (provides training environment), PyBullet (a physics engine), PyTorch, Python, C++, LibTorch, and two open source embedded libraries (pca9685 for servo control and MCU6050 for the IMU).

Other requirements of our project include compiling a list of useful machine learning resources that demonstrate new learning the team has acquired as well as attending keynotes at Imagine 2021 and completing the Coursera course on embedded machine learning. These resources and new learning will help facilitate the development of a machine learning course. Additionally, given our project is more open ended, part of the main initial requirements of the project is to define exactly what our project focus will be and figure out what requirements are necessary for the selected application. One of the main goals of the project is to be able to incorporate what we develop and learn into a course at ISU; therefore, it is important that we develop in a way that is modular so that course implementation is feasible.

As for constraints, we are working on a clock and need to have the project done by the end of the spring semester of 2022. We need to be able to get the robot in time and then train the robot.We are working with a budget of $600, which should be plenty as the robot costs $300. Since we are trying to create a class or concepts for a class out of this project, we need to use reusable components such as common coding languages, specifically C++ and Python. The tools that we will be using for training the robot is OpenAI gym. The inputs that we get are IMU readings and servo positions and the

2.3 Engineering Standards

The first engineering standard associated with this project is the UM10204 I^2C bus specification. The chip onboard the robot is the ATMEGA328 and will not likely have the computing power necessary for our application. The board has the ability to switch communication from the ATMEGA328 chip to an I^2C bus controlled by a Raspberry Pi. The next two standards relate to the design and testing of autonomous robots. The IEEE P1872.2 standard for autonomous robots ontology set guidelines for building autonomous systems consisting of robots operating in various environments, and the P2940 standard for measuring robot agility will provide quantitative test methods that are useful to show how well our robot walks. The Petoi Bittle robot dog is powered by a lithium ion battery pack sized for its current peripheral load. However, as we add more functionality, the battery pack may need to be expanded to accommodate the extra load. The standard that will apply to this is the IEEE 1725-2021 standard for rechargeable batteries for host devices such as mobile phones. Lastly, we may want to incorporate wireless communications for control of the robot, which will be accomplished through the Raspberry Pi. The IEEE 802.11 standard for wireless local area networks will guide our use of the wireless network.

2.4 Security Concerns and Countermeasures

2.4.1 Physical Security

First and foremost, for the physical security of our robot, there would be the protection of the robot itself so when it falls over during training or gets hit while walking, it will not fall apart and stop working. Second, on our robot we have a Raspberry Pi which could be accessed if the robot were to be off or not moving.

As for the first countermeasure, we can protect the robot from breaking by making sure everything is securely put together and then we can put a protective layer "Hood" over the electronic parts on the top of the robot. For the second counter measure, we could create a tab to go over the plugs on the Raspberry Pi. This would not stop someone from taking off the tab and getting into the Pi but it would slow down the process. The best way to protect against someone getting into the Pi would be with Cybersecurity.

2.4.2 Cybersecurity

The first concern for cybersecurity would be attackers logging into the Raspberry Pi through SSH. A second concern would be attackers logging into the Pi by hardwiring into it. These are concerns because if an attacker were to gain access to the Pi either virtually or hard wired, then they could change code for the robot. A third concern would be DoS (Denial of Service) and DDoS (Distributed Denial of Service) attacks. This is a concern as an attacker could overload the Pi and disable it which would then disable the robot. A fourth concern would be using the default Pi user on the Raspberry Pi. This is a concern because the default Pi user is the most brute forced password on the planet. A fifth concern would be man in the middle attacks if we are sending information from our computers to the Raspberry Pi. This is a concern because if we are not using the correct services, our traffic may not be encrypted. A sixth concern would be if someone is already in the system, how would we shut down the robot.

For the first concern of logging into the Raspberry Pi through SSH, you can change the port that the SSH service uses so it slows down the attacker because they would have to find the correct port. Another thing that can help with this concern is having the attackers be locked out after so many login attempts. Lastly, we can allow only certain people through the firewall to login so that if you have the wrong IP, you never get the chance to login. For the second concern, we can implement the same login attempt countermeasure. For the third concern, we can limit the amount of traffic coming into the Pi so that these attacks cannot happen. For the fourth concern, we can disable the default Pi user and create a new account to use with a secure password. For the fifth concern, we can use secure services to send information over the wire to the Pi like HTTPS (Hypertext Transfer Protocol Secure) and SFTP (Secure File Transfer Protocol). For the sixth concern, we can have a script that we can log in and run that would shut down the robot and shut out anyone in the system.

See more Raspberry Pi Security information at - https://raspberrytips.com/security-tips-raspberry-pi/

3 Implementation

To begin our walking robot implementation, we have experimented with deep Q networks (DQN) and deep deterministic policy gradient (DDPG) to solve the mountain car problem. This allowed us to practice implementing reinforcement learning before taking on the more challenging task of implementing an algorithm for our robot application.

Through this practice implementation, we have developed a better understanding of the actor critic method. Initially, the neural network will know nothing. The actor will take in a state and then output an action. On the other hand, the critic takes in a state as well as the agent's action and then predicts the value of the next state. The actor will learn based on the critic's suggestion. Utilizing reward, the critic is able to learn and adjust to the desired outcome. As the critic improves, the actor gets better feedback and thus is able to improve. The figure below demonstrates this process.

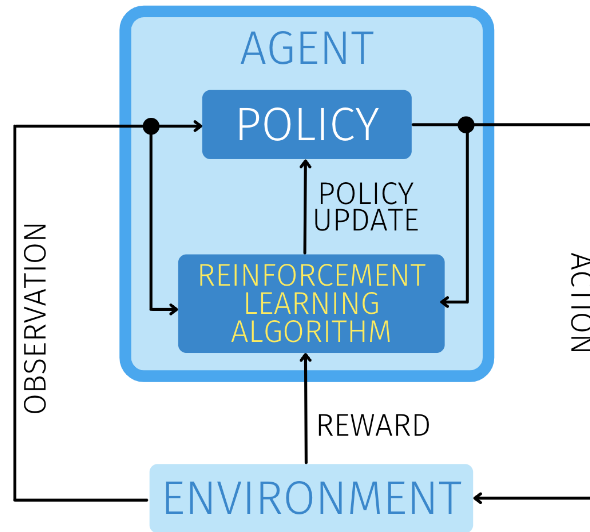

Figure 1: Actor Critic Method

After the practice implementation, there were several main tasks to be completed for our project implementation: establishing communication between the Petoi Bittle board and the Raspberry Pi, constructing a robot model for virtual training, completing Python training, and developing the C++ application.

The Petoi Bittle came with a NyBoard V1_0, which contained an MPU6050 motion sensor as well as a PCA9685 16-channel PWM controller. The Raspberry Pi communicated with these peripherals using the I2C protocol. The MPU6050 produced accelerometer and gyroscope data which was used to determine the orientation of the robot. To convert the raw IMU data to these values, we used a library designed to interface with the MPU6050 over Raspberry Pi (Hirst). The PCA9685 was used to specify the servo positions for each leg. We used another library to interface with the PCA9685 to set these positions (Sprung).

In order to deploy our trained model onto the Pi, we installed the Python package Torch using a pre-built version of the package designed for the processor on the Pi. We used CMake to build our main application and link Torch and the PCA9685 library.

Before we were able to train our neural network using Python, we needed a virtual model of our robot. Instead of reverse engineering the robot ourselves to construct a virtual model, we utilized a model found online created by a third-party designer (AIWintermuteAI). The model was a collection of meshes organized as a Unified Robot Description Format (URDF) file. We modified this file to alter the standing position of the robot by changing the joint angle values. This put the robot in a more stable starting position.

For virtual training, a PyBullet simulation is converted into a Markov Decision Process (MDP). The state is drawn from PyBullet measurements, and the action is applied to the relevant virtual joints. The state is designed to mimic what we have access to in real life, namely joint positions, roll, pitch, yaw, and acceleration. The action is similarly based on setting joint positions. Gravity is assumed to be 9.8 m/s^2, and the surface is flat. Note the friction coefficient used is the default in PyBullet. It is unclear how this matches real life, more research is needed on friction coefficients in simulation. Once we have converted the PyBullet simulation into a MDP, we are able to use typically reinforcement learning algorithms such as DDPG on it. The output of these algorithms is a neural network, the actor, capable of stable walking in simulation. Given the current state, it outputs an action that will lead to stable walking. This neural network is created using PyTorch, which can conveniently be transferred to C++.

Developing the C++ application involved two main interfaces: embedded and agent. These two components are linked via a buffer that passes the action and state values between them. The policy was trained virtually and uploaded into the agent. Once the policy was in the robot, it no longer interacted with the virtual environment. Figure 3 in Appendix IV shows the interaction between the virtual model, the agent, and the embedded component. By using interfaces for the agent and embedded side, our application has the flexibility to replace an agent or embedded implementation with a different implementation and still use the same main application code.

To develop the agent interface, a setState and getAction method were defined. This interface was then implemented in two different ways. The first implementation was the Stretching Agent. This implementation was used for testing whether the robot could receive values and perform them correctly and provided a way to verify all motors could activate. The Stretching Agent runs the robot joints through a continuous series of motions, incrementally changing the joint angle values up to 45 degrees and then back to -45 degrees. The second implementation was the Neural Network Agent. The Neural Network Agent takes in the neural network model and uses the model to determine the next action for the robot. To implement this, the code has been set up to take in the filename of the model on the command line. This filename is provided to the Neural Network Agent when its constructor is called. Each of the servos are initialized to the starting position and then the setState method gets the current state of the embedded system. The getAction method utilizes the LibTorch dependency to determine what the robot's next action should be based on the model trained in PyTorch. LibTorch allows for loading in a serialized PyTorch model and executing it purely from C++, with no dependency on Python.

To develop the embedded interface, a getState method and setAction method were defined. Two implementations of the embedded interface were also developed. The first implementation was an Embedded Test. This implementation was developed to test that the action and state values were getting passed to and from the embedded side correctly when using the main application loop. The second implementation of the embedded interface was the Petoi Bittle implementation. This implementation moves the servos based on the output of our actor neural network. Additionally, the state of the robot is achieved by getting data through the IMU.

Aside from the agent and embedded interface, the C++ application also had a main application loop and logger implemented. Executing our main application without any command line parameters deployed the Stretching Agent. To deploy our Neural Network Agent, the path to the trained model must be passed in as the first command line argument. Both of these main application loops continuously get the state of the embedded system, determine the next action based on the agent, and then pass this action back to the embedded system. Command line arguments could also be used to free the servos from their last position or start an interactive servo tuning sequence. The servo tuning sequence allowed for making micro-adjustments to servo positions to account for any hardware misalignment. The logger was implemented to provide a way to debug our application. To use the logger, a function name and message are provided, and then the logger prints all messages to a log file.

In Appendix IV, some pseudocode is provided to give an overview of the implementation discussed. Additionally, in Appendix IV, Figure 2 shows a UML diagram of the main components of the C++ application.

4 Testing

4.1 Testing Process

Throughout the development process, we created several tests to ensure our applications were functioning properly. Incremental testing during the whole development cycle made finding errors simpler than trying to test all parts of our final applications.

4.2 Unit Testing

Unit tests were created for both our virtual training and embedded system. The virtual training had unit tests for the PyBullet physics engine and the Petoi Bittle environment. The embedded system used unit tests to verify the servos and IMU were operating correctly.

To test PyBullet, we created a program which loads our model into the engine and launches its GUI. This would show if the Python environment was properly set up and all the packages were installed correctly. This program was also capable of moving the leg joints to specified angles. This was helpful with our integration testing as it allowed us to verify the joint angles in simulation matched what we saw on our physical robot. The Petoi Bittle environment was tested by loading the environment in PyBullet and launching the GUI again. Once the environment was loaded, a random action was assigned to the virtual robot. This allowed us to verify each joint on the virtual robot was able to move.

4.3 Interface Testing

Our servo and IMU unit tests also served as interface tests between the Raspberry Pi and the NyBoard. These tests ensured our I2C communication was set up and functioning correctly.

We also tested the agent interface by implementing a stretching agent. This agent helped us ensure the control flow of data from the NyBoard data to the agent and back was operational. The stretching agent has a clear expected behavior (to move all joints to 45 degrees and then back to -45 degrees continuously), so we could ensure from this that values were being passed to the embedded side correctly based on the resulting behavior. Additionally, logging was used to identify the values that were being passed to the embedded side to check that the values matched up with the expected behavior.

When implementing the neural network agent, we needed to verify that the correct values were being produced from the forward pass of our neural network to obtain the action. This was done by checking the values that were outputted by the neural network model in C++ and comparing them to the output from the model in Python. If these values matched, then the model in C++ is producing the actions to pass to the robot correctly.

4.4 Integration Testing

Once the applications passed all of the unit and interface testing, we moved on to integrating the components together. This was done by running our neural network agent in our main application loop. If everything was functioning properly, the robot would get data from the IMU and its current servo location

and pass it to the neural network agent. The agent would receive this state, run the values through one forward pass of the network, and output the new servo angles for each joint. This process should repeat until the main application is terminated. This testing showed some flaws in our final system, which is discussed in the results section.

4.5 Acceptance Testing

For our project, we are conducting a proof of concept of a ML walking robot. Our goal was to demonstrate to Dr. Rover, our client, a successful NN model and a physical robot that can learn how to walk based on said model. These are both items that can be physically demonstrated to Dr. Rover.

Another goal of our project was to propose a way to implement machine learning into an undergraduate course. Therefore, we set out to create a development process that could be followed along in a classroom setting. For example, we could present modules that mirror the development process that we followed to reach our final product. We would take aspects that we deemed were most valuable to us during our learning process and make suggestions as to how these ideas could be incorporated into an undergraduate lab. We would need our development process and implemented system to be reliable and maintainable in order to be utilized from semester to semester. For an example of the types of lab activities students could do related to this project, see Appendix V.

4.6 Results

By the end of the second semester, we were able to successfully pass all of the unit and interface tests. Our virtual model functioned properly in the PyBullet engine, and there was successful communication between the Raspberry Pi and the NyBoard. While we got promising results of the robot walking in our virtual training, the model did not meet our distance or speed goals set for the robot.

When elevated on a stand, the robot moved its legs in a walking motion. This motion did not translate very well when the robot was on the ground. The robot often rolled over on its side when going to take its first step. After making some modifications to the feet on the robot, the robot could move forward several inches before either tipping over or hitting its front on the ground. These results were likely caused by differences between our virtual environment and the real world. The differences that likely exist and what changes we could have made, given more time, are discussed in Appendix III.

5 Related Products & Literature

5.1 Prior Work/Solutions

Reinforcement learning is an area of research that is continually evolving and advancing. This model of machine learning has many applications and will continue to be an area of focus in machine learning. As reinforcement learning will continue to be an important tool in the future, our project aims to integrate reinforcement learning concepts into the classroom. Teaching machine learning at the undergraduate level can be somewhat challenging because it requires students to make “linkages between complex concepts in linear algebra, statistics, and optimization” (Sahu et al., 2021). In a preliminary study by Sahu et al. titled, “Integrating machine learning concepts into undergraduate classes”, methods to best teach machine learning were explored. The researchers explored teaching machine learning in a side-by-side method and as stand alone workshops. In the side-by-side approach, the machine learning concepts were integrated into a pre-existing signals and system course. The researchers found that “while students like the side-by-side delivery better, the workshops showed improved student learning” (Sahu et al., 2021). Based on this study,

we determined it would be useful to implement machine learning concepts in a CPRE course by constructing labs, like workshops, that students can complete side-by-side with embedded machine learning concepts being taught in the classroom. We have developed several lab materials, located in Appendix V, as examples of labs that could be performed by students.

We also looked at literature to identify robot abilities in applications similar to ours. One particular area we were interested in was the speed at which the robot could walk. In a study conducted by Haarnoja et al., a deep reinforcement learning algorithm was applied to a four legged robot. The robot in this study was able to “walk forward at a speed of 0.32m/s” (Haarnoja et al. 2019). The study also mentioned that this was comparable to the default controller gait provided by the manufacturer (Haarnoja et al. 2019). We decided to set a similar goal for our robot based on its default controller gait. Our robot walked at a speed of 0.19ft/s using the default controller gait, so we set a goal to have our robot walk at a similar speed using the machine learning algorithm.

6 Closing Material

6.1 Conclusion

Over the course of this semester, we successfully conducted reinforcement learning virtually and deployed the model on the Petoi Bittle. We fully developed a C++ application which is responsible for passing the actions determined by the neural network model to the embedded side of the application. The transition from the virtual environment to the real world proved to perform differently due to the environment differences. This was something that was expected, and we hoped to have enough time to fine tune the reinforcement learning algorithm to achieve a satisfactory walk in the real world. See Appendix III for recommendations on next steps for the project.

6.2 References

- AIWintermuteAI (2021) Bittle_URDF [Source code]. https://github.com/AIWintermuteAI/Bittle_URDF/.
- C. Sahu, B. Ayotte and M. K. Banavar, "Integrating machine learning concepts into undergraduate classes," *IEEE* , 2021, pp. 1-5, https://ieeexplore.ieee.org/abstract/document/9637283?signout=success.
- Haarnoja, Tuomas et al. “Learning to Walk via Deep Reinforcement Learning”. *arXiv preprint,* arXiv:1812.11103v3 [cs.LG], 19 Jun. 2019, https://arxiv.org/pdf/1812.11103.pdf.
- Hirst, R. (2012). PiBits [Source code]. https://github.com/richardghirst/PiBits/tree/master/.
- Lillicrap, Timothy P., et al. "Continuous control with deep reinforcement learning." *arXiv preprint arXiv:1509.02971*, 2015.
- Seno, Takuma. “Welcome to Deep Reinforcement Learning Part 1 : DQN.” *Medium*, Towards Data Science, 21 Oct. 2017, https://towardsdatascience.com/welcome-to-deep-reinforcement-learning-part-1-dqn-c3cab4d41b6b.

- Sprung, R. (2019). PCA9685 [Source Code]. https://github.com/Reinbert/pca9685.

6.3 Appendices

6.3.1 Appendix I - Operation Manual

6.3.1.0 Python Virtual Simulation Setup

1. Download and install Python3.7
2. Install Torch, Gym, Numpy, etc using pip
3. Register the Petoi Bittle Environment using “pip install -e path/to/PetoiBittleEnvironment. This registers the environment with your python installation so it can be called as an external
4. Run main.py using the flags listed in the file. Specify “PetoiBittle-v0” as the environment. Use the flag --store_models to save the models.
5. Run eval.py to test the models. Use --visualize to render a simulation, use --pause to step through it frame by frame and print the state and action values
6. When you are happy with your model, use ConvertModelToCPP.py to convert the model from Python to C++. Copy the C++ model to the Petoi Bittle along with the verification text file to make sure it does not get corrupted.

6.3.1.1 Rasperry Pi Setup & Running the C++ Application

Torch Installation Process

7. Download and install Python3.7
8. Create and activate a Python virtual environment
9. Download numpy, torch, and gym using pip
10. Download Python 3.7 onto the Raspberry Pi
    a. wget https://www.python.org/ftp/python/3.7.0/Python-3.7.0
11. Unzip the download in the home directory and run the following commands to install Python.

```
sudo tar zxf Python-3.7.0.tgz
cd Python-3.7.0
sudo ./configure
sudo make -j 4
sudo make altinstall
```

12. Make 3.7 default Python version
    a. This may not be necessary as long as you can create a virtual environment with Python 3.7.

```
vim ~/.bashrc
alias python3='/usr/local/bin/python3.7'
source ~/.bashrc
```

13. Create the Python 3.7 virtual environment and activate it.
    a. If successful, your shell prompt should start with (bittle_env)

```
sudo apt-get install python3-venv
python3 -m venv bittle_env
source bittle_env/bin/activate
```

14. Install numpy using wheel
    a. Wheels allows us to use a pre-built version of packages instead of having to build them from source. This saves us a lot of time. The wheel must be for the same python version (cp37) and architecture (armv7l). The Raspberry Pi 3 uses an ARMv8, but it can operate in an ARMv7 compatible mode (source: https://pi.processing.org/technical/).

```
pip install
https://www.piwheels.org/simple/numpy/numpy-1.17.1-cp37-cp37m-linux_armv7l.w
hl#sha256=e17c4b3b8b2e00ec1d9af5ddae2602b325b453465c0d9bfb0df7046b0989a760
```

    b. To learn more about wheels: https://www.raspberrypi.com/news/piwheels/
    c. Original source: https://www.piwheels.org/project/numpy/
15. Install torch using wheel
    a. Original source: https://github.com/Kashu7100/pytorch-armv7l

```
pip install
https://github.com/Kashu7100/pytorch-armv7l/blob/main/torch-1.7.0a0-cp37-cp3
7m-linux_armv7l.whl
```

16. Install other torch dependencies
    a. This dependency prevents errors when importing torch

```
sudo apt-get install libopenblas-dev
```

17. Test Torch Installation
    a. Test the installation by importing torch in Python. If no output is given, then the library was successfully installed

```
python3
>>> import torch
```

Install Embedded Libraries

1. Clone the below repositories to the home directory and follow the installation process given on GitHub.
    a. pca9685 (Servo control): https://github.com/Reinbert/pca9685
    b. MCU6050 (IMU): https://github.com/alex-mous/MPU6050-C-CPP-Library-for-Raspberry-Pi

Build Using CMake

1. Download CMake

```
sudo apt install -y cmake
```

2. Create CMake file for libraries
    a. In order for CMake to properly find and link the wiringPi and pca9685 libraries, two files must be added to the Pi (`FindWiringPi.cmake` and `FindWiringPiPca9685.cmake`).
    b. After creating these files and filling them with the contents below, move them to /usr/share/cmake-#.##/Modules, where cmake-#.## is the version of CMake you are using.

FindWiringPi.cmake:

```
find_library(WIRINGPI_LIBRARIES NAMES wiringPi)
find_path(WIRINGPI_INCLUDE_DIRS NAMES wiringPi.h)
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(WiringPi DEFAULT_MSG WIRINGPI_LIBRARIES
WIRINGPI_INCLUDE_DIRS)
```

FindWiringPiPca9685.cmake:

```
find_library(WIRINGPIPCA9685_LIBRARIES NAMES wiringPiPca9685)
find_path(WIRINGPIPCA9685_INCLUDE_DIRS NAMES pca9685.h)
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(WiringPiPca9685 DEFAULT_MSG
WIRINGPIPCA9685_LIBRARIES WIRINGPIPCA9685_INCLUDE_DIRS)
```

3. Build
    a. To build the application, navigate to the build directory (sdmay22-45/CppApplication/build).
    b. This directory contains a script (run_cmake_all.sh) that will run all of the necessary commands to generate a Makefile and build the main application.
    c. **The Python virtual environment must be activated to execute these commands**. Otherwise, CMake will not be able to find the path where Torch was installed.
    d. The commands in run_cmake_all.sh can also be executed separately.
    e. After building, a Makefile will be generated in the build directory. This Makefile can be used to rebuild the application if any changes are made using make.
4. Setup CMake

```
cmake -DCMAKE_PREFIX_PATH=`python3 -c 'import
torch;print(torch.utils.cmake_prefix_path)'` ..
```

5. Build Application

```
cmake -build . -config Release
```

6. Run Application
    a. Run the main application using the executable inside the main directory.
    b. If no command line arguments are given, the Stretching Agent runs by default.
    c. To run the Neural Network Agent on a trained model, pass the path to the saved model as the first command line argument. Trained models can be added to the Pi from your local machine using 'scp' if you are connected via SSH.
    d. Running the application with '-f' as the first argument frees the servos from their last set position.
    e. Running the application with '-t' as the first argument starts an interactive tuning sequence to adjust the position of the servos. The robot legs should be perpendicular to the body. To adjust this, input an offset for each servo until it is perpendicular. To save an offset, enter 's'. The tuning sequence should only need to be done if the legs are removed from the robot.

```
./main /path/to/model
```

7. Run Unit Tests
    a. In order to run unit tests for the servos or IMU, include `ServoUnitTest.h` or `IMUUnitTest.h` respectively. Next, instantiate the unit test in the main application and use the function `runUnitTest()`.

6.3.2 Appendix II - Alternative Versions

The initial version of our project involved utilizing TensorFlow and MuJoCo as development tools. These tools were initially chosen because members of the team had prior experience with them. However, in the beginning stages of the development process, TensorFlow was producing issues that we were having trouble locating the root cause. As a result, we experimented with PyTorch to see if this would work instead. PyTorch worked successfully with what we were trying to implement in TensorFlow, and we found PyTorch was easier to work with even though the team did not have prior experience with it. Additionally, MuJoCo was having trouble loading in our URDF file. Therefore, we tried PyBullet as an alternative. This was able to load in the URDF file, and therefore, PyBullet was the newly selected physics engine for our project.

6.3.3 Appendix III - Other Considerations

Throughout the implementation process, there were many different learning lessons. We learned a lot about the difficulties in transitioning from a virtual training environment to the real world environment. Due to these difficulties, we were not able to achieve the level of walking stability and speed that we had hoped. If

we had more time, there are several things that we would pursue to achieve the desired walking stability and speed. The first recommendation for project continuation would be to accelerate the development cycle by using a GPU for reinforcement learning. For our project, we utilized a VM provided by the university that does not come with a GPU. The compute cluster that was available to us has a GPU, but the GPU is too old for the newer PyTorch. Secondly, we would improve the physics simulation to better match reality than it currently does. Utilizing more accurate friction coefficients and exact robot part size, shape, and density would greatly improve the physics simulation and make virtual training more closely mimic reality. The current software used for the physics simulation, PyBullet, is relatively new so with time, as the software improves, it will become easier to more accurately simulate the real world. In our simulation, the default friction coefficient is used and every part is considered the same weight. These assumptions may contribute to some of the differences realized between the simulation and real world environment. Lastly, we would recommend to consider a robot that is designed to have reinforcement learning applied to it from the ground up. With our robot, we are not able to obtain torque values or accurately measure exact leg position. As a result, we must assume that the robot accurately carries out the action it is told to without an explicit way to check if this is actually true. Having more sensors and servos with encoders to verify the state is correct could help achieve further walking stability and improve our training algorithm.

As we developed the training model, we observed some interesting behaviors from the robot. At one point, the training model deployed on the robot produced a somewhat stable result, but instead of progressing forward as you would when walking, the robot sort of bounced back and forth in place, so it looked like it was dancing before eventually falling over. This was funny to watch. At another point, if the robot got to a position where it was close to laying down, it would choose to lay down and quit trying to walk so that it didn't have to risk getting a negative reward. It was really neat to see how the robot would react to the different reward functions that were put in place. Even though the robot did not successfully walk as we wanted, the team still learned a lot about machine learning and how you would even begin to implement a more complicated learning algorithm as well as how this can integrate with an embedded system.

As a final attempt to improve the stability of the robot with the current model, a team member 3D printed shoes for the robot so that each leg could have a wider base. This seemed to improve the robot's stability and allowed the robot to remain standing for a longer period of time.

For future work on lab development, finding ways to have the training happen faster is a plus and is well-welcomed. Another potential area for future work is to create more labs with additional focuses besides these three. This could potentially be looking more critically at how the environment is set up and exploring different setups for the environment. This could help students know how to set up their own machine learning applications and can help understand more of a "follow-through" process.

Figure 2: Petoi Bittle with Shoes

6.3.4 Appendix IV - Code & Diagrams

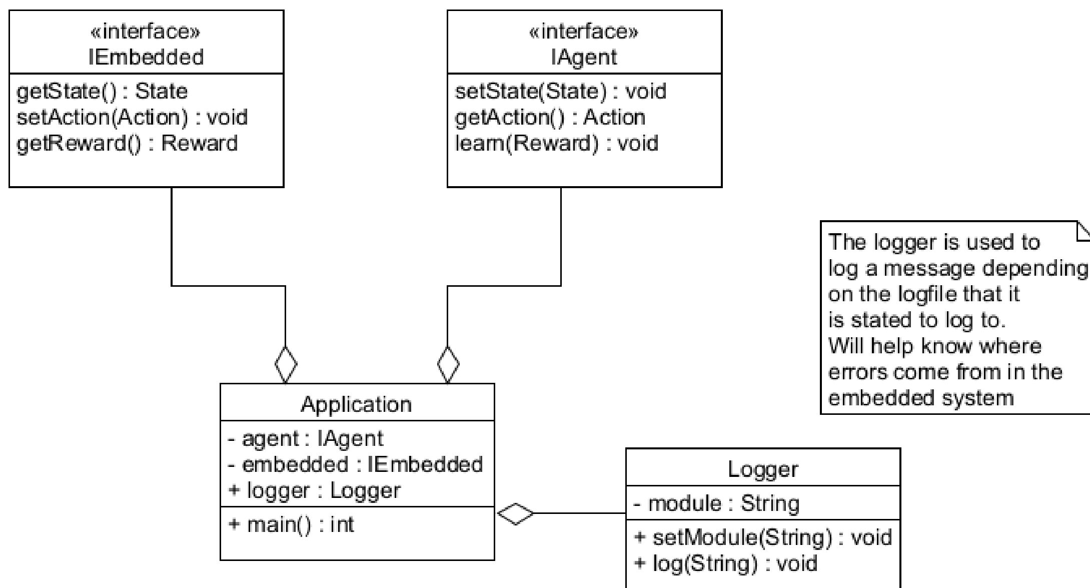

Figure 3: Component Diagram

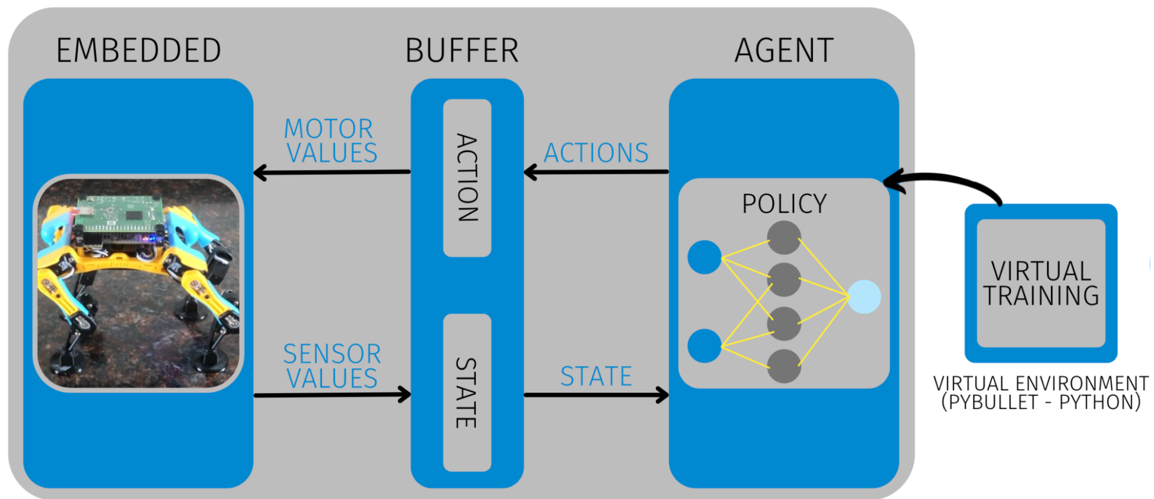

Figure 4: Conceptual Design Diagram

Pseudocode for C++ Application:

```
Class IAgent (Interface for the agent)
	Constructor(String model) //needs file location of NN if using one
	Destructor
	setState(State s)
	getAction()
	learn(Reward r) //this one most likely isn't needed unless we want to
learn on the robot

Class IEmbedded (Interface for the embedded side)
	Constructor()
	Destructor()
	getState()
	setAction(Action a)
	getReward() //this one most likely isn't needed unless we want to
learn on the robot

Class Application (IE the main)
main()
	Init embedded
Init agent
	while(1)
		State s = embedded.getState()
		agent.setState(s)
		Action a = agent.getActions()
		embedded.setActions(a)
```

6.3.5 Appendix V - Lab Module Example

Our team has included three lab modules to cover three critical points of understanding: Basic RL concepts, Actor and Critic relationships, and the Reward Function. Two of the three labs involve the Petoi Bittle and the introductory lab uses the MountainCar simulations we experimented with early in the project. The labs have been crafted to have specific goals and objectives at the beginning of them tailored to target the three critical points of understanding mentioned above.

Each of these labs are meant to be completed over several weeks. Our team believes it is important for students to engage with training the model, even though it is time consuming. Thus, for a class structure, since training takes a long time, fewer labs that included critical and denser concepts was considered more beneficial than shorter and smaller labs that barely scratched the surface. This approach will also allow students to participate more in-depth with the provided code and Petoi Bittle simulation and application. An excellent closing lab could be to help guide students to the final solution of the real-world application from the digital simulation or guide them in exploring how to do this.

The first lab is focused on the Basic RL Concepts. The goal of this lab is to ensure students can identify the fundamental concepts in the code provided. It is structured to give them a guided start to knowing how to use the code our team has developed to aid their understanding. They should be able to analyze and interpret gathered information from the software and be able to identify the Bellman equations in code by the end of the lab. This lab is not meant to be as long as the other labs, but we still believed it was a good start in order to get students familiar with the course and some of the basic concepts. There is some training required in this lab, but it is not for the Petoi Bittle but instead for the MountainCar problem. We found this problem an easy and accessible place to start compared to throwing students into the complexities of the Petoi Bittle walking simulation.

The second lab focuses on the Actor-Critic relationship. This was another fundamental concept our team identified and thought was important for students to understand. The goal of this lab is to have students interact, analyze, modify and understand the relationship between the actor and critic. This lab is used with the Petoi Bittle functionality. Students work to map out the actor-critic relationship in the code and, just as in the previous lab, the students will train the agent, monitoring the change in relationship between the actor and critic. This is to help students understand the impact of the decisions made in the model training and help them solidify their understanding of this delicate relationship.

The third lab developed focuses on the reward function. This was yet another fundamental concept our team identified would be beneficial to have a lab document on. This one is the longest lab we developed as it requires many different training instances. This was necessary to be able to identify and see the relationship between always rewarding, never rewarding, and then finding a middle between those. This lab is meant to be the launching point for the final lab we mentioned above.

These are example labs and depending on the class structure, they can be longer or shorter. If opting for shorter labs, the labs can be broken up so one or two sections of each lab are due each week (based on length and complexity). This can help monitor student progress more regularly and ensure students do not fall significantly behind. This may be beneficial due to the length of the simulations. The downside of the shorter labs may result in students not being able to engage with more complicated ideas that longer labs can elaborate on. Another solution we came up with for the longer training times is to find a resource for larger processing power, whether that be through GPUs or a server cluster. Although this would be expensive for the class, this could be a tool to do more with the lab content in a shorter period of time. This would result in students being able to train many models quickly and focus on more in-depth concept application.

The following links may be useful at various points in the labs:

- How to create MDPs - https://gym.openai.com/docs/#environments

- Information on different RL algorithms - https://ychai.uk/notes/2019/04/02/RL/SpinningUp/RL-taxonomy/
- How to use PyBullet (simulator) - https://usermanual.wiki/Document/pybullet20quickstart20guide.479068914/html#pf15
- How to create a custom MDP - https://gerardmaggiolino.medium.com/creating-openai-gym-environments-with-pybullet-part-2-a1441b9a4d8e
- Info on DDPG - https://spinningup.openai.com/en/latest/algorithms/ddpg.html
- General info on how NNs work - https://www.youtube.com/watch?v=aircAruvnKk&ab_channel=3Blue1Brown
- Past work on trying to train RL algorithms on robots in simulation - https://www.youtube.com/watch?v=Wypc1a-1ZYA&start=381&ab_channel=MATLAB

Understanding Basic RL Concepts

In this lab, there will be experiments with the actor and critic network, and an application of the Bellman equations discussed in the lecture. At the end of the lab, the student should be able to identify applications of the Bellman equations in the models provided, successfully experiment with the provided RL model and devise conclusions based on that model, be able to execute the use of the software tools provided, and analyze and interpret gathered from the lab materials.

The student will be using the MountainCar files provided in order to train and experiment with these concepts.

1. Recall that the Bellman Equation represents the sequence of rewards based on the actions that are taken. Where in the code does the Bellman Equation appear? What do the values represent?

2. In simple terms, what is the "goal" used to optimize the critic? Is this an effective goal? Why or why not?

3. You will use the `main.py` script to train the DDPG agent MountainCarContinuous-v0. You can visualize the learned policy by using the `eval.py` script. Note that training the model should not take more than a few minutes to produce useful results. Provide a brief description of your agent's behavior.

4. How could you go about changing the goal of the critic? How would this improve your training model? Run a few training models to see if you can get the car to move up the hill. **Train the agent.** Document proof of your effort and experiments for this task. If you were unable to complete the task, what behavior did you observe that was unwanted? How would you go about remedying it?

Training the agent:

Note that training this agent will likely require many hours for complex models. It may be a good idea to have multiple trainings happening at the same time. Each model that is saved will be stored in a directory called `pytorch_models`. This can help reduce the wait for each part. In order to train the model, you will have to run the following command:

```
python3.7 main.py --env_name MountainCar-v0 --save_models
```

Note that you may need to set the environment by using the following:

```
pip install -e path/to/Environment
```

Which will be the Petoi Bittle python environment for the model.

Testing the Models:

In order to test the results that you have, the following command will be run:

```
python3.7 eval.py --visualize model_name
```

Note that you will have to insert the name of the model you have trained to input it into the program correctly. It is assumed to be in the `pytorch_models` directory. You can also use the `--pause` flag in order to step through the simulation frame by frame and print out the state and action values.

Actor-Critic Evaluation

In this lab, there will be experiments and exploration with understanding the structure and functionality of the actor and critic behavior. At the end of the lab, the student should have a greater understanding of how the actor and critic model to interact, be able to analyze, modify and understand the relationships between the actor and the critic.

In Reinforcement Learning (RL) it is important to define the functionality of the actor and critic well in order to have a solid understanding of their roles and how they contribute to the end goal. For this lab, you will be using the Petoi Bittle files provided in order to analyze, modify, and understand the relationships between the actor and critic.

1. Using your notes from class and the code provided, what is the role of the actor? How is this reflected in the code? What roles does the actor have? Please list concrete examples from the code provided as well as a brief description of how they interact.

2. Using your notes from class and the code provided, what is the role of the critic? How is this reflected in the code? What roles does the critic have? Please list concrete examples from the code provided as well as a brief description of how they interact.

3. How many inputs are there to the actor network? How many outputs? What clues does it give you to know it is the actor-network?

4. How many inputs are there to the critic network? How many outputs? What clues does it give you to know it is the critic network?

5. There are many different inputs that affect the agent's behavior. In this next section of the lab, you will be using evaluating and or editing `eval.py` and `DDPG.py`. Use the information from the information you've gathered above and draw a diagram showing how the Actor and Critic interact in `DDPG.py`.

6. Take a look at the `train` function located in `DDPG.py`. This is where the model is ultimately trained using the information from the actor and critic networks that you have examined. Notice that there are several different steps that occur before the updating of the models. Come up with 3 different experiments that either target the critic or actor models. These should be executed by commenting out code, we will

analyze the change in training in this method. **Train the agent.** Describe what your experiment is targeting (actor or critic training) and what the results are. Please provide thorough documentation of the results you see. This can be done through images, explanations, and screenshots. How much does each change affect the outcome? Note: Look at the notes below for training and testing the module to help you.

Reward Function Analysis

In this lab, there will be experiments with the reward function to see how it impacts the behavior. Note that training the agent takes a lot of time. The earlier you begin the better. At the end of the lab, the student should have a greater understanding of the relationship between the reward function, analyze and gather data from the model through experimentation, and draw conclusions of the effect of the model. Note that there are sections in the bottom of the lab to be used as reference to help you know how to train and test the models you create.

It is recommended you train the agent as early and as soon as possible for training the agent can take a very long time.

1. Recall from lecture that a reward is an incentive for the agent to do something to achieve its goal. Open `PetoiBittleEnv.py` file and examine the reward function return value. Set the reward function to always return `-1`. **Train the agent.** Provide a brief description of what the agent learns to do. Why do you think this is?

2. Now set the reward function to return a value of `+1`. **Train the agent.** Provide a description of what your agent learns to do. Why do you think this is? How is this different from setting the reward function to always return `-1`?

3. How can you change the reward function so that it will give points for moving forward? Make those changes (and only those changes). **Train the agent.** Provide a brief description of what your agent learns to do. Why do you think this is?

4. Does the above question successfully create a walking agent? Is this what you expected? Why or why not?

5. Add in the provided reward function to the code again. What is different about it from the reward function you added in Step 3? Identify at least two distinct parts of the function and how they impact the result. For each part, answer the following: What does this part do? What is it composed of? Where do these inputs come from? How do you anticipate they affect the output?