Pacbot V1: A Retrospective

Holt Spalding (with contributions from Saurav Gyawali & Eric Wu Yufeng)

Abstract

This paper discusses Pacbot, an autonomous mobile robot designed to play a physically-rendered variation of the classic Atari arcade game, Pacman. The idea was conceived by the Harvard Robotics Club in 2016, and since then they've hosted an annual intercollegiate competition pitting students' Pacbots against one another. This year (2019), my two collaborators and I managed to place third at the competition with our very first version of Pacbot. This paper does not explicitly describe any sort of experimental research we conducted, but it serves more as a record of our design process and our plans for the second version, Pacbot V2. This project was intentionally designed with very lofty goals in mind, originally focused primarily on implementing an RL-based path planning algorithm for our Pacbot which could be tested in simulation. As it turns out, designing a mobile robot from scratch that's under budget with the proper computational power, precision, and size we require is more difficult than expected, and so many of these higher level concerns have been put on hold until the completion of the redesign (I have already begun work on Pacbot V2, which will continue into the summer). Though we haven't yet achieved all of our original goals, this project was in no way a failure. The work presented here represents the culmination of hundreds of hours of learning and engineering which will serve useful during the redesign process.

1 Introduction

The Harvard PacBot competition is an annual intercollegiate robotics competition hosted by the Harvard Undergraduate Robotics Club which began in 2016. The competition attempts to bring Pacman, the classic Atari arcade game, to life by asking each competing team to design an autonomous mobile robot which is capable of navigating a maze whilst avoiding "ghosts" and collecting as many points as possible within three given lives. The team with the most points at the end of the game wins the competition. The maze is a 94.5 x 105.5 inch plywood arena with 7 inch wide corridors, upon which an implementation of the Pacman game, designed by alumni of the Harvard Robots Club, is projected. The rules of the game are exactly the same as the rules of the arcade game, which I will assume you, the reader, are at least somewhat familiar with.

The game engine is run off of a private wireless network which all competing robots must connect to before beginning their respective rounds. This server

publishes game state information at a rate of 10 Hz which Pacbot can then draw upon to make decisions. The game state information includes Pacbot's current score, its current number of lives, the position of the four ghosts in the maze in discretized coordinates (the maze is split up into 7 x 7 inch units), whether or not the ghosts are in a frightened state (which means they are safe for Pacbot to "eat"), and Pacbot's position in the maze in discretized coordinates. Pacbot's position is calculated with the help of two camera's connected to the game server running OpenCV and a large yellow acrylic sheet placed on top of the Pacbot. Pacbots can be no more than 4 inches tall and they must be small enough to fit through the 7 x 7 inch corridors.

There are few, if any, competitions quite like the Pacbot competition, but I think the "Pacbot problem" ultimately boils down to a precision real-time path planning problem in a known dynamic environment, which there is quite a bit of precedent for. The Pacbot problem is interesting and potentially has research value because (as far as I'm aware) no one has yet put research into efficient dynamic path planning in the context of a game, where speed *and* strategy play a huge role in the success of the robot. The goal of Pacbot is not simply to find the optimal obstacle-avoiding path from point A to B, but rather to find the optimal path from point A to the optimal point B. Because Pacbot is an embodied system and is inherently error-prone, there is a constant struggle between the higher level game-playing agent, the lower level path planner, and the physical limitations of the robot which all must be reconciled in order to achieve optimal play. *This* is the difficulty of the Pacbot problem, and it's my belief that its research value could be far reaching should anyone ever solve it.

2 Background Related Work

Navigation and efficient path planning have been a major focus of robotics research since the first autonomous robots were ever built and today there remain countless path-planning algorithms at the disposal of roboticists everywhere.[2, 4, 9] The Pacbot problem, while the only of its kind, definitely has its roots in some of these more fundamental path-planning problems and understanding how these problems are traditionally solved could definitely serve in improving future versions of Pacbot. Duchon *et.al* compare and contrast some of the most well-known path-planning algorithms in terms of computational efficiency and planned-path length in the context of real-time path planning based on a known representation of the environment.[1] Among the algorithms described, including A*, Focused D*, Incremental Phi*, Basic Theta*, and Jump Point Search, they cite Jump Point Search as being the most computationally efficient algorithm when navigating a largely static environment with a few moving obstacles. Although Jump Point Search is very computationally efficient, it does not always find the shortest path to its goals, which is not necessarily an issue in the context of the Pacbot problem. At a high level, Jump Point Search is an optimization of the more well-known A* search, which reduces symmetries in the search procedure by means of graph pruning.[3] JPS is incredibly efficient be-

cause of its capacity to significantly reduce the path-planner's search space, and its perfect for dynamic environments because of its capacity to quickly re-plan whenever an obstacle gets in the way.

The Pacbot problem is not simply concerned with efficient path planning however, there is also an element of game-playing strategy involved. This subproblem has arguably already been solved. Thanks to modern advances in parallel computing and machine learning methods grounded in frequentist statistics, programmers can now use machine learning models to learn to play just about any arcade game for them through millions of trial-and-error simulations. A reinforcement learning agent designed by researchers at Deepmind famously learned to play Pacman optimally in 2013 by utilizing a method known as deep Q learning.[6] Since then, researchers have only improved upon this algorithm, finding ways to reduce training times and increase the predictive power of the game-playing agent.[7, 12] Q-learning agents learn the optimal action to execute given any state of their environment by essentially placing a value on each state-action pair. This value, known as a "Q-value", is based upon the expected cumulative rewards the agent believes could be achieved as a result of performing each action in the given state. As a Q-learning agent is exposed to more game scenarios, it's able to observe the consequences of its actions and slowly refine its expectations for rewards achieved. *Deep* Q-learning is necessary when it becomes intractable to save these Q-values for each state-action pair because of high dimensionality in the game's state and/or the agent's action space. Deep Q-learning solves the curse of dimensionality for many medium-sized problems by using non-linear function approximators like neural networks in order to assign Q-values to state-action pairs indirectly based on the parametized weights of a model.

Although there's been a lot of research put into efficient real-time path planning and high-level game-playing strategy, there remains the issue of how these two components can be adequately reconciled to solve the Pacbot problem. Rather than designing ways for these two components to work together, many researchers have tried finding ways to blur the distinction between them and essentially formulate them in terms of a singular problem. Policy gradient RL algorithms are a growingly popular method for solving continuous control problems in which agents must constantly plan motion trajectories in continuous action space in dynamic and unpredictable environments.[8, 10] Much of the research surrounding policy-gradient RL has primarily concerned itself with bipedal and quadrupedal locomotion [5, 11], in which the agent must constantly plan the continous actuation of its joints in order to maintain balance and move in a desired direction. Recently, more and more research has been put into policy gradient algorithms for mobile robots because of the rising importance of autonomous vehicles. Yi describes the virtues of using policy gradients for autonomous driving over deep Q-learning, which suffers the curse of dimensionality in high-fidelity state space.[13]

3 Technical Approach

The following provides a detailed description of the design process for Pacbot V1 as well as its technical limitations and a few proposed design changes for Pacbot V2. All developed source code, images, and videos for this project are available **here**. All embedded hyperlinks have been bolded.

3.1 Pacbot V1 Hardware Design

Pacbot V1 stands at about 4 inches tall and is 5 pounds in weight with a circular acrylic chassis that's about 6.5 inches in diameter.[1] The robot has two 6 cm rubber-lined plastic wheels connected to two high powered, brushed DC motors placed on the underside of the chassis (two caster wheels are placed on the front and back of the chassis as well to give the robot balance). These motors are connected to a **Polulu Dual VNH5019 Motor Shield** which connects to an **Arduino Mega 2560 Rev. 3**. The motor shield allows for fairly speedy but inaccurate differential drive control, while also providing encoder feedback from the motors for rudimentary odometry. The robot is also outfitted with four cheap **ultrasonic distance sensors** placed uniformly around the outside of the chassis facing outwards. These send signals to the Arduino, preventing Pacbot from running into any walls should it get too close. The "brain" of the machine is a **Raspbery Pi 3B** which interfaces with the Arduino using ROS. In this design, the Pi provides the robot all of its decision making faculties, while the Arduino serializes information sent from the Pi into electric signals in order to control all the on-board hardware. It is also the job the Raspberry Pi to interact with the game engine server so Pacbot effectively knows the state of the game at all times.[2]

3.2 Pacbot V1 Software Design

Below is a rudimentary diagram of the software pipeline that was used in the design of Pacbot V1. ROS Kinetic sits at the bottom of the stack, running on a stripped down version of Ubuntu-Xenial for Raspberry Pi designed by a company called **Ubiquity Robotics**. This OS sets up a wifi access point whenever the Pi turns on, making it easy to ssh into. At the start of the control loop, a ROS subscriber node receives game state information in a serialized form from the game server which it then parses and publishes for other nodes to utilize. The game server information is received using a heavily modified version of Harvard Robotics Club's **open source code** which relies upon an asynchronous networking library known as asyncio. This library is exclusively written for Python 3, while ROS Kinetic exclusively runs on Python 2, so it required quite a bit of engineering to make these softwares compatible.

Once the game state information is successfully published to a ROS topic

[1] photos and video of the robot available **here**

[2] all robot firmware can be found **here**

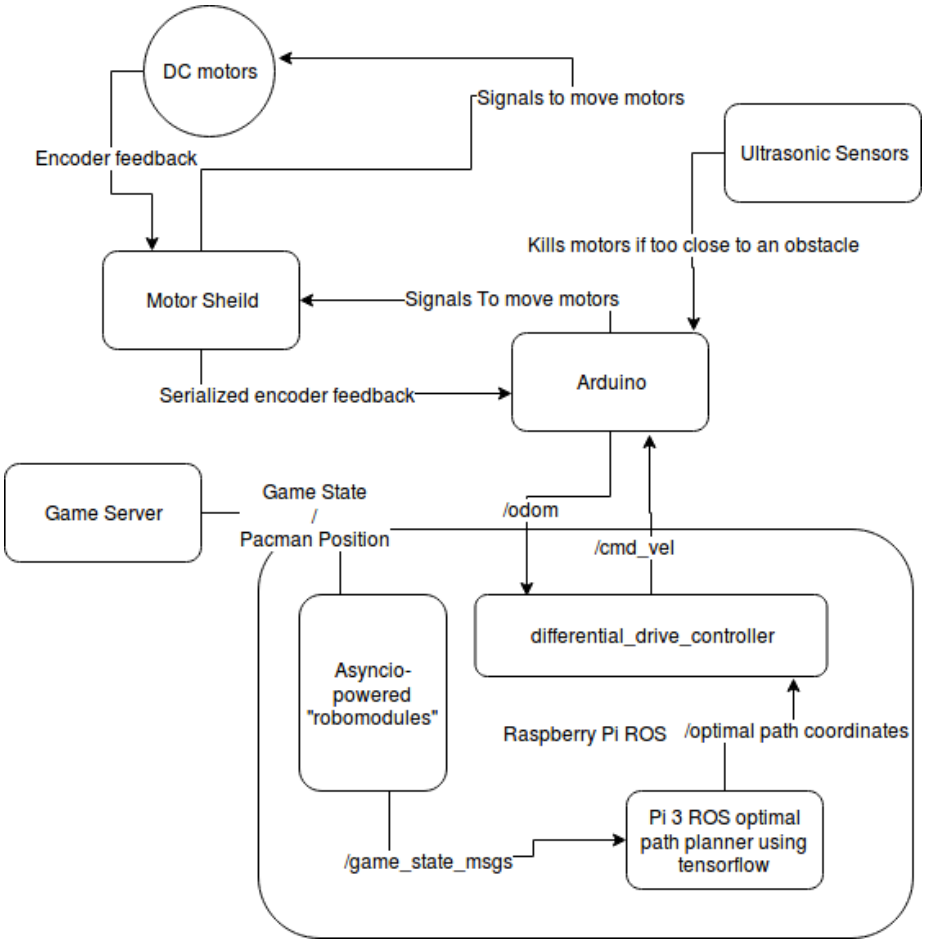

Figure 1: Software control loop blue print

in a human readable form, a high level path planner finds optimal maze coordinates to travel to, given the state of the game. We originally wanted the high level planner to be a deep Q learning agent written in Tensorflow, however we never implemented this on the actual robot due to a lack of computational resources. Instead, we used a rudimentary breadth first search algorithm in competition that could plan a few maze spaces ahead and essentially favored paths with the most number of pills and the fewest number of ghosts.

Even though the DQN agent was never implemented on the actual robot, a lot of work has been put into designing both a game-playing RL agent as well as a simulated learning environment for Pacbot V2. [3] In the provided reposi-

[3] all path-planning and software designed to teach the robot in simulation can be found **here**

tory is a Tensorflow implementation of a deep Q-learning RL agent which can be trained on **Open AI Gym's Mspacman-ram-v0** . This agent learns the optimal action to take at any moment during a game of Pacman based soley on a 128 byte RAM buffer representing the state of the game. This 128 byte RAM buffer is similar to the 200 or so byte buffer used by Harvard's game engine to represent its game state. So, in principle at least, the Tensorflow file provided could be just as easily be used to train an agent on the Harvard game engine, and I've heavily edited their game engine code to allow for that. This work however, was eventually abounded to make room for a lot more fundamental engineering kinks that still need to be worked out, and I probably wont come back to this till Pacbot V2 is finished sometime in the next month or so. In addition, work has also been done on creating a simulated learning environment in Gazebo for Pacbot V2 to train in (Rviz is also used to visualize the position of the pills and ghosts). I played around with a few different mobile robots in simulation, however, I soon realized training in Gazebo isn't useful unless you have a physically-accurate model of the robot you intend to train. For that reason, I've put that project on hold as well until the completion of Pacbot V2.

Once the planner determines the optimal coordinates to travel to, a ROS package known as **diff_drive_controller** handles all the differential drive control, publishing command velocities to either motor. The robot performs simple tank-steering maneuvers, always turning in place when it wants to switch directions. The differential drive control package also handles odometry. Given our knowledge the wheels' radius, how far apart they sit from one another, and information from the rotary encoders, we are able to make fairly accurate predictions of the robots location, especially when that information is combined with game server information about Pacbot's location (we plan to implement a Kalman filter to reconcile these two sources of information in future versions). At this point, the ultrasonic sensors do almost nothing, only killing the motors when Pacbot gets to close to a wall.

4 Technical Demonstration

This project was never designed experimentally, so there's little in the way of "results" that can be presented here. If it's any indiciation of our success, the robot designed by my collaborators and I managed to place third at this year's (2019) Pacbot competition despite numerous technical issues. Unfortunately there is no video of our robot in competition, and we can't demo our Pacbot without first recreating the maze used in competition (complete soon). Regardless, there are a few videos of the robot running and being teleoperated in the "Media" folder of the provided repository.

5 Conclusion and Future Work

The original goal of this project, which was in designing a high level path-planning RL agent for Pacbot, unfortunately could not be achieved due to numerous technical challenges. So much went wrong in the design of Pacbot V1, the primary issues including its size (it was far too large for the 7 inch corridors of the maze), its general lack of adequate sensory modalities, cheap and imprecise encoders, and very rudimentary software design which was brought on by a general lack of computational resources. Regardless, the value of this project to myself and my collaborators cannot be understated, and I wholeheartedly believe it to have been a success. Before working on Pacbot V1, I had no exposure at all to ROS, robotic design, or the fundamentals of mechanical and electrical engineering, but now I feel as though I could design Pacbot V2 entirely on my own. A lot of necessary progress was also made in breaking down the Pacbot problem, and only now is it obvious what Pacbot V2 will require in order to perform well in next year's competition.

In the design of Pacbot V2, we are going to begin by completely scrapping the hardware used in V1. Instead of a Raspberry Pi and an Arduino, we will instead run most of the firmware on Nvidia's **Jetson Nano**, which is much faster and has been explicitly optimized for ROS. The Nano can also interface directly with a motor shield and all of our sensors, reducing the amount of space we'll require to house all the new hardware. Our Tensorflow RL agent will be run on a **Coral TPU Dev Board** which is highly optimized for Tensorflow (in fact it can't do much else). With all this new hardware, it's also likely we may need to look into better mobile power supplies and potentially a liquid cooling system.

With our computational requirements met, we'll be able to add much improved sensors and control mechanisms to Pacbot. Pacbot V2 will be outfitted with two omni-directional wheels with very high-fidelity encoder information, providing faster and more precise turns around corners and far superior odometry. We will also outfit Pacbot V2 with an IMU so its orientation can be determined, and we will implement much more sophisticated wall-avoidance software for our ultrasonic sensors. Finally, in the interest of testing Pacbot V2, I've begun work on completely reconstructing the maze that is used in competition.

Once Pacbot V2 has been completed, I will turn my attention back to the fundamental issue at hand, which is in efficiently planning paths that generate the most in-game reward. Based on my research, treating the Pacbot problem as a continuous control problem to be solved using a policy-gradient algorithm will likely be the fastest and most accurate means of controlling Pacbot's movements. This method however will likely require a lot of training time and there is also the question of whether behavior learned in a Gazebo simulation would necessarily transfer to the actual robot. Another avenue I plan to investigate is whether or not the knowledge from an RL-agent trained on the Harvard Robotics Club's game engine can effectively be transfered to Pacbot. More on these issues will be published in the provided repository once Pacbot V2 has been completed.

References

[1] František Duchoň, Peter Hubinskỳ, Andrej Babinec, Tomáš Fico, and Dominik Huňady. Real-time path planning for the robot in known environment. In *2014 23rd International Conference on Robotics in Alpe-Adria-Danube Region (RAAD)*, pages 1–8. IEEE, 2014.

[2] Shuzhi Sam Ge and Yan Juan Cui. New potential functions for mobile robot path planning. *IEEE Transactions on robotics and automation*, 16(5):615–620, 2000.

[3] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*, 2011.

[4] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[5] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 3, pages 2619–2624. IEEE, 2004.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in neural information processing systems*, pages 2863–2871, 2015.

[8] Jan Peters and Stefan Schaal. Policy gradient methods for robotics. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2219–2225. IEEE, 2006.

[9] Anthony Stentz et al. The focussed dˆ* algorithm for real-time replanning. In *IJCAI*, volume 95, pages 1652–1659, 1995.

[10] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[11] Russ Tedrake, Teresa Weirui Zhang, and H Sebastian Seung. Stochastic policy gradient reinforcement learning on a simple 3d biped. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2849–2854. IEEE, 2004.

[12] Tycho van der Ouderaa. Deep reinforcement learning in pac-man, 2016.

[13] Hongsuk Yi. Deep deterministic policy gradient for autonomous vehicle driving. In *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, pages 191–194. The Steering Committee of The World Congress in Computer Science, Computer ..., 2018.