Learning Biomimetic Locomotion through Online Machine Learning

Final Design Report

BENG 187C Winter 2017
Jared Buchanan –Bioengineering
Huan Tran – Bioengineering
Donald Dean – Biotechnology

Table of Contents

Abstract

The implementation of online Q-Learning with Incremental Feature Dependency Discovery algorithm on a simulated snake robot provides a software basis for developing robotic locomotion policies. The design goals of this software include low dependency and user input, a high quality locomotion policy output, a low convergence and runtime to find the policy, and a low software development cost. By developing this software within the existing physics framework on Unity, an open-source video game development software, robotic locomotion be developed and tested with a minimal dependency on user input for free, disregarding the cost of development man hours.

Online Q-Learning, a type of reinforcement learning, ensures that a near-optimal locomotion policy is acquired within the environment itself, while Incremental Feature Dependency Discovery ensures a low convergence time with reduced computational complexity and power. For Q-Learning with Incremental Feature Dependency Discovery, the snake robot traveled 0.01 meters per learning iteration over 500,000 total iterations, compared to 0.0050 meters for Q-Learning Tabular. The time complexity of Q-Learning with Incremental Feature Dependency Discovery is O(k*2^k) with a clocking of 2-3 ms per iteration.

Executive Summary

- *Low Dependency and User Input:* Q-Learning with Incremental Feature Dependency Discovery is implemented on simulated three-jointed snake robot in Unity, which is a free, open-source video game development software and scripting environment. The software for development robotic locomotion policies is easily integrated and tested due to the fully integrated physics engine.
- *High Quality Locomotion Policy Output:* The snake robot travels 0.01 meters per learning iteration over 500,000 total iterations, compared to 0.0053 meters and 0.0050 meters for random and Q-Learning Tabular respectively.

- *Low Convergence and Runtime:* In terms of convergence, Q-Learning with Incremental Feature Dependency Discovery has per-time-step-complexity of O(k*2^k), for which k decreases overtime as new features are discovered. Q-learning Tabular has per-time-step-complexity of O(k), for which k stays constant overtime. Therefore, over the duration of the learning process, the former method is less computationally taxing and thus converges faster than the latter method. In terms of runtime, Q-Learning with Incremental Feature Dependency Discovery, which requires more operations than Q-learning Tabular, is clocked at about 2-3 ms per iteration. The Q-learning Tabular is clocked at less than 1 ms per iteration. Both methods are under the 200 ms iteration threshold are thus low in terms of runtime.
- *Low Software Development Cost:* The free, open-source Unity video game development platform means that software development is essentially free.

Introduction and Relevant Background Information

Reinforcement learning (RL), a subset of machine learning, is focused on goal-directed learning by mapping situations, or *states*, to *actions*, with the end goal being the maximization of a *reward* signal (Sutton). *Online* machine learning refers to the data about the *environment* being sequentially available to the *agent*, which in this case is the snake robot. This means that no prior training with outside data occurs before the agent is placed in the environment, and all the learning occurs while the agent interacts with its environment in this sequential fashion, also called a *Markov Decision Process* (MDP).

In general, an MDP with respect to the environment refers to decision making in discrete time intervals such that for some state *s*, an action *a* is chosen to transition to a new state *s'*, for which a unique reward *R* is given for the transition tuple (*s, a, s'*). This process satisfies the *Markov Property* (MP), which means that the decision to take action *a* does not depend on previous states and actions. In a small finite environment such as a Tic-Tac-Toe game or GridWorld, an agent is fully aware of possible states and actions, and through learning, continually satisfies MP. However, with a game such as chess, there is a finite but very large amount of existing states; in a realistic amount of time, an agent cannot possibly explore every single state and action to find the best *trajectory*, or the sequence of actions from state to state. In other words, the general model of the environment (e.g. the 8 x 8 chessboard) is known, but the agent is required to explore the environment in order to gain information and learn. This problem of exploration is called the *curse of dimensionality*. Because infinite exploration of the environment is impossible, the agent must be programmed with a predetermined balance between *exploration* of new state-actions and *exploitation* of known state-actions, known colloquially as *exploration vs. exploitation*.

RL addresses the problems of curse of dimensionality and its derivative, exploration vs. exploitation. Furthermore, with RL and opposed to supervised learning, there is no explicit requirement to improve suboptimal trajectories or thoroughly explore every single state action pair.

Locomotion robotics currently perform a wide range of tasks such as surgery or bomb disposal and the number of applications will surely increase as the field advances. The software embedded on these systems that control how the hardware operates to create locomotion is just as important to successful robotic operation as the hardware itself. Therefore, there is a need to create efficient locomotion policies with a low cost/effort development process.

Hard coded control policies, which are the standard for locomotion robots, are inefficient and expensive since policy development requires direct attention from an engineer who's time and effort is very costly. Additionally, the quality of the hard coded policy is dependent on the quality of the engineer. Lastly, the hard coded approach is dependent on the environment and there is no guarantee that it will work outside of the context it is specifically designed for. In other words, unexpected encounters not foreseen by the designer may cause problems for hard coded locomotion policy because that static policy cannot adapt to new situations. Successfully applying RL techniques to this task would alleviate these issues since RL builds a policy without direct interaction with a human designer, since RL is mathematically guaranteed to converge to an optimal policy (i.e. quality is independent of an individual engineer's skill), and since RL can adapt to any environment the robot is placed in (Csaji, 2008).

Ultimately, the development cost and quality of hard coded locomotion policies is dependent on the complexity of the underlying robot. Therefore, we will limit our focus to the highly complex robotic locomotion systems, such as those that seek to mimic the structure of animals, for which the development cost becomes quite expensive.

For example, a similar research project to ours involved finding a locomotion policy for an autonomous robotic cockroach. Designing a successful locomotion policy that controlled the legs of the robot involved a tremendous amount of effort from a team of engineers and researchers. They were able to create a successful hard coded control system only after intensively studying the cockroach's nervous system control and the associated leg mechanics and then mimicking that control system (Sanchez, 2015). This is certainly a viable approach that could be taken for creating a locomotion policy for our snake-like robot project, but we seek to implement a simple RL learning algorithm that allows the snake-like robot to self-learn an optimal locomotion policy. This will eliminate the need for an intensive study into the mechanics of snake locomotion and will allow for a significantly reduced development time and cost. The direct end user (researchers and engineers) could benefit from a streamlined and significantly less complex robotic software design process. This in turn will increase productivity and the performance of the robot. Individuals who actually use the robots, whether it be researchers, consumers, or industry professionals, could benefit from an improved quality of an existing robotic system (e.g. better Roomba) and from novel products that improve quality of life and/or productivity (e.g. snake-like surgery robot).

Alternative Solutions and Design Choice

There are only a few, albeit critical, functional requirements for the design solution of the desired control algorithm. First, the solution must incorporate a learning aspect so that it can it improve from interactions with the environment, thus creating an efficient locomotion policy. Second, the algorithm must be computationally viable with the limited processing power of a robot. Since the physical robot specifics are not available at this time we will use the hardware characteristics of an Arduino, a common robotic microcontroller, for our processing power benchmark. An Arduino has a 16MHz clock speed and 2 kB of runtime RAM (Margolis, 2011). Third, the algorithm must converge to a control policy in an efficient and timely manner. A poorly designed algorithm could take years to converge to a good control policy. Convergence must occur on the hour to day timescale and not the year timescale.

Design Alternatives and Analysis

General design alternatives exist for programming robotic snakes. The quality of the locomotion policy produced by the algorithms is what makes the applications of the general design alternatives unique to the project. The quality of the policy as one of the design goals later in this paper. The following design alternatives is proposed for the machine learning algorithm:

1. **Q-learning (model-free)**: This method uses individual interactions with the environment to iteratively improve the locomotion policy. According to **Figure 1**, an arbitrary initial locomotion policy is input into the system by a user. The robot then implements this policy and interacts with the environment as a result. This interaction generates feedback from a reward function that assigns value to the outcome of certain actions performed within the environment. The Q-learning algorithm then takes this feedback and improves the current locomotion policy with low computational costs. For example, if the feedback was negative then the robot in the future will not perform that same action again and vice versa. This improved policy is then implemented and the process is repeated until the policy improvements begin to converge to the optimal policy (Bertsekas, 2012).
2. **Environment Model + Policy Iteration Method (model-based)**: This method contains 3 steps according to **Figure 2**. The first step is to implement an exploration policy that allows the robot to explore its surroundings and build an estimate environmental model that describes how the robot will interact when certain actions are chosen. More individual interactions yield a more accurate environment model. Once a suitably accurate environment model is known the optimal policy is computed using the policy iteration model-based method. An initial policy is implemented and the estimate model is used to evaluate how good that policy performs in said environment according to a reward function. After policy evaluation the policy is then improved slightly in an iterative fashion based on the environment model and reward function (Bertsekas, 2012).
3. **Standard "hard-coded" robotic controller**: **Figure 3** shows the hardcoded robotic controller. This method relies exclusively on the skill and expertise of the engineer specifying how they want the robot to locomote and then using visual or data feedback to improve the hard coded policy. However, there is no guarantee that the engineer will be able to improve the policy whereas in the self-learning methods (**Fig. 1 and 2**) there is such guarantee unless optimality is already reached ("Robot Control", 2016).

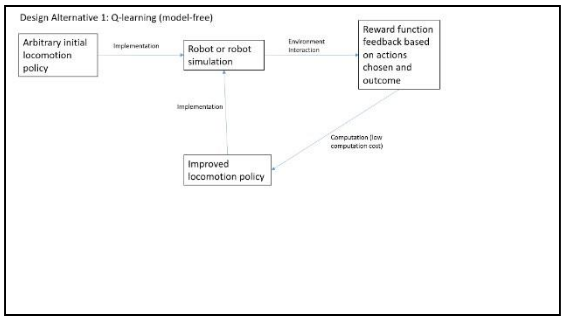

Design Alternative 1: Q-learning (model-free)
Arbitrary initial locomotion policy
Implementation
Robot or robot simulation
Environment Interaction
Reward function feedback based on actions chosen and outcome
Implementation
Computation (low computation cost)
Improved locomotion policy

Figure 1 *Q-Learning*

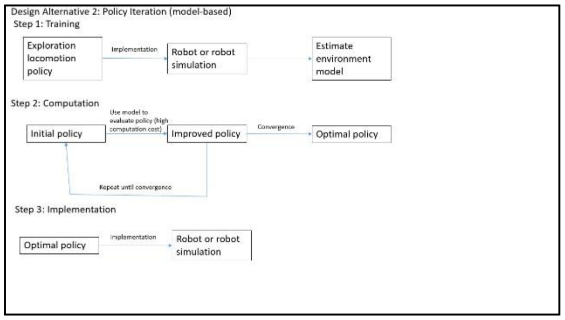

Design Alternative 2: Policy Iteration (model-based)
Step 1: Training
Exploration locomotion policy
Implementation
Robot or robot simulation
Estimate environment model
Step 2: Computation
Initial policy
Use model to evaluate policy (high computation cost)
Improved policy
Convergence
Optimal policy
Repeat until convergence
Step 3: Implementation
Optimal policy
Implementation
Robot or robot simulation

Figure 2 *Environment Model + Policy Iteration*

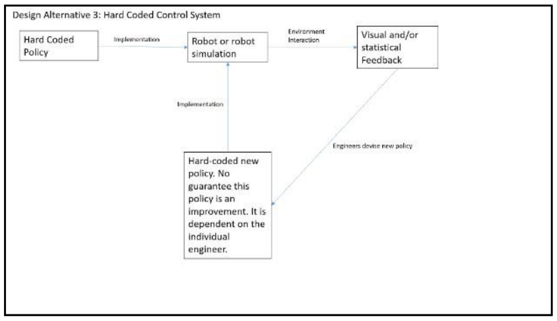

Design Alternative 3: Hard Coded Control System
Hard Coded Policy
Implementation
Robot or robot simulation
Environment Interaction
Visual and/or statistical Feedback
Implementation
Engineers devise new policy
Hard-coded new policy. No guarantee this policy is an improvement. It is dependent on the individual engineer.

Figure 3 *Hard-Coded Control System*

Patent Issues

The scope of this project is to create a locomotion control algorithm, which cannot be patented until it is implement onto specific hardware according to the World Intellectual Property Organization and United States Patent and Trademark Office ("Patenting Software", 2016 & "An Examination Of Software Patents | USPTO", 2016). However, if we were to merge our project with a hardware research group and implement this algorithm onto a physical robot then we run into issues with patents. We would have to look at existing RL learning implementations and the associated hardware implementations in order to determine if we are infringing on any existing patents. To be clear, the issue of patent infringement lies mostly outside of the scope of this project since the project's main goal is the design of a reinforcement learning algorithm.

Design Goals

The design goals were chosen and weighted as follows:

- **Dependency on user input** (does not need additional correction after reaching an optimal solution) (0.2)

- **Quality of the locomotion policy** (0.4)
- **Time to converge to near optimal policy** (0.15)
- **Computation power needed to run in real time** (i.e. hardware has enough processing power to run algorithm in real time) (0.15)
- **Development cost** (not related to hardware cost) (0.1)

The **quality of locomotion policy** is the most important design goal since the goal of the project is to move the snake in a desired direction. All other design goals are meant to support this central design goal. **Dependency on user input** is the second most important because we want to limit the errors that can occur due to dependency on users and we want to increase the ease of use of the system. **Time to converge** and **computational power** are both weighted the same since both categories are necessary for algorithm feasibility on the final robot. We define *convergence time* as the total time for the algorithm to reach a solution. *Computational power* is the required number of operations per second. If it takes too long for the algorithm to converge or if the processor on the snake robot cannot run the calculations in real-time, then a near optimal policy will never be reached in a reasonable amount of time for the design to be practical. **Development cost** is the least important issue because this is a research project and not a product for the general public. Furthermore, designing algorithms is relatively inexpensive given the necessary hardware and software such as computers and an interpreter (which are already provided for us).

Decision Matrix

	Dependency on user input	Quality of the locomotion policy	Time to converge to near optimal policy	Computation power needed to run in real time	Development Cost	Total
Weights	0.2	0.4	0.15	0.15	0.1	1
Policy Iteration (Model-based)	5	7	7	10	6	6.95
Q-learning (Model-free)	10	7	9	8	10	8.35
Hard coded Algorithm	0	2	3	10	2	2.95

Decision Matrix Rationale

User Input: Using the **Q-learning method** unequivocally results in the least amount of necessary user input since it only requires an initial policy input, which is arbitrary and thus does not have an effect on the overall success of this method. The **policy iteration method** requires more user input since a successful exploration policy is needed for step 1 in the process. In other words, to build a model the entire environment must be explored. Thus, the user's choice in exploration policy directly affects the success of the method. The traditional **hard coded method** is completely reliant on user input since users are directly responsible for the locomotion policies being implemented.

Quality of the locomotion policy: The **policy iteration** and **Q-learning methods** both rely on the same statistical principles when calculating and improving their policies, thus resulting in locomotion policies that are generally equal. In the context of robotics these methods will not yield optimal policies (rating of 10) because in theory an infinite number of environmental interactions is needed to converge to the optimal policy. Instead, we settle for a near optimal policy achieved with a reasonable amount of environmental interactions performed within a reasonable time frame (i.e. on the scale of hour/days and not centuries). Therefore, a near optimal locomotion policy warrants a rating of 7 for both these methods. The standard **hard coding method** on average produces lower quality locomotion policies in complex robotic systems due to the difficulty of dealing with these systems, thus resulting in a low rating in comparison with the self-learning methods.

Time to converge to near-optimality: The **Q-learning method** takes the shortest time to converge to a near optimal policy since it has only a single step and a low computational requirement. **Policy Iteration** methods must perform a training step and a high cost/time computation before reaching the near optimal policy. Therefore, the Q-learning method on average will take a shorter amount of time to converge. The **hard-coded method** takes the longest to converge if it converges at all. Individual engineers are responsible for adjusting the control algorithm (i.e. no self-learning component) and convergence to near optimality is completely dependent on the skill of the engineer. Thus, this method scores low on the convergence time criteria.

Computation power needed to run in real time: The **Policy Iteration** method uses more computational resources compared to the **Q-learning method**. However, this heavy computation does not occur during the operation of the robot (see step 2 in figure 2). Instead, all the computations are done beforehand allowing the locomotion policy to run in real time with minimal computational resources. A **hard coded control algorithm** also does not require any complex real time computations since the algorithm simply sends movement signals to different hardware components. Lastly, the Q-learning method does require computations to be performed continuously in real time while in operation. However, the computations themselves are not computationally expensive. Therefore, the Q-learning method only has a slightly worse rating in this area.

Development Cost: We define development cost as the cumulative man hours and computational resources needed to reach a near-optimal locomotion policy. **Q-learning** scores the highest since this method is self-contained, has minimal user dependency, and low computational requirements. **Policy iteration** has 3 steps and needs to interact with engineers in between these steps. It also has a dependency on user input, and has high computational requirements which is why it scored lower than Q-learning. Lastly, the **hard coded algorithm** scored very low because it requires an engineer or a team of engineers to put large amounts of effort into improving the locomotion policy due to the lack of self-learning in this method.

Conclusion: The cumulative valuations given to each design goal and their associated weights give the total score for each design alternative. The best design alternative according the total score is clearly **Q-learning.**

Proposed Design Solution

A summary of the parts and resources for the design solution is shown at the end of this section. We will be implementing a model-free Q-learning RL algorithm for creating a generalized locomotion policy (**Fig. 1**). We will then apply the generalized locomotion policy to a simulated snake robot. The simulated snake robot will then be placed in different environments and multiple trials will be run to build a locomotion policy through trial and error interaction. The design of the Q-learning algorithm is already well-defined in literature and the structure of the algorithm is located in Figure 1. The main focus of our design will be taking the standard Q-learning algorithm, creating an interface that allows implementation to any type of simulated robot, and then applying this to a simulated snake robot in different environments.

The generalized interface will basically consist of the RL algorithms and a tool to communicate and control the hardware of the model. The interface will allow you to choose the parameters to be controlled by the locomotion policy (e.g. joint angle) thus making it model independent. This model independence will increase usability and flexibility of the algorithm and its implementation without having to rework much of the underlying algorithm implementation code. This will ultimately streamline troubleshooting if we alter the model and will allow for further application of the code beyond this project.

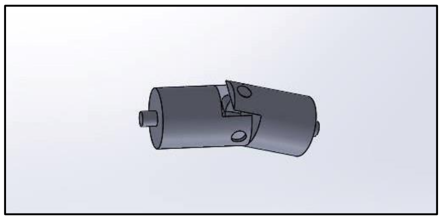


Figure 4 *SolidWorks model of snake joint with two degrees of freedom.*

The snake simulation will be built in a physics simulator, and will have similar proportions and dimensions to a real snake. It will consist of spherical body segments and joints connecting those body segments **(Fig. 4-5)**. Each joint will have two rotational degrees of freedom (i.e. a double pin joint). This will allow the snake to change its overall body orientation in the x and y direction but it won't allow the snake's body segments to rotate about each other. Oftentimes, RL algorithms converge to unexpected locomotion policies and we want to prevent this model snake from locomoting by rolling since the physical robot and snakes for that matter are not able to roll, which is why
the last rotational degree of freedom will be constrained.

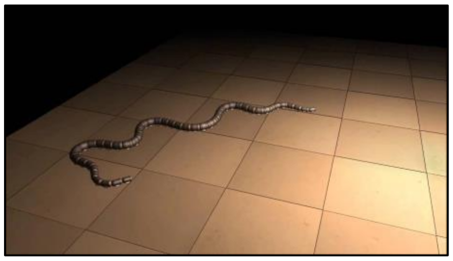


Figure 5 *Simulation of assembled snake. using the above joints (Lu, 2016).*

The main advantage of a RL algorithm is that the robot designer does not need to worry about the complicated dynamics occurring between robot and environment. Our RL policy will control the joint angles, which are easily measured parameters, and will use the observed output motion caused by the change in joint angles to improve that policy. Rewards will be given according to a simple criterion in order to provide feedback on the quality of the locomotion policy. The ultimate goal of the robot is to reach some specified point in the shortest amount of time so the rewards for each action chosen will be proportional to the distance the snake has moved in the desired direction. For example, let's say 3 separate actions (i.e. each joint angle is moved by a specified amount) cause the snake to move 5 cm, 1 cm, and -5 cm in the desired direction respectively. The returned reward after each action will be highest for 5 cm and then 1 cm and -5cm. This reward will allow the RL algorithm to 'learn' that the action that caused the snake to move 5cm is better than the one that caused the snake to move -5cm. The locomotion policy will then be improved accordingly to favor the action that yielded the highest reward. After running many episodes of this task and after convergence to a locomotion policy we can judge the overall quality of said locomotion policy by simply measuring how long it takes the snake to reach a given finish point from a given starting point.

The joints will connect between 5 and 10 body segments. The number of body segments will increase the number of joints and the number of joint angles that need to be controlled. We will start with a lower number of joints (i.e. 5 body segments) so that convergence to an optimal policy can occur more rapidly (i.e. faster troubleshooting) due to the fewer number of controllable parameters. We will then increase the number of body segments once we know that algorithm works so that we can test the algorithm on a more 'snake-like' model.

Several recent papers have optimized methods for robotic snake locomotion to move with and without the use of reinforcement learning. This project seeks to improve upon the existing reinforcement learning algorithm implementation to discover novel and optimal ways to move In 2009, scientists at Carnegie Mellon University developed parametrized sinusoidal algorithms for a robotic snake which freed robot operators from specifying every single degree of freedom for movement ("Robot Control", 2016). A Q-learning algorithm would offer even more independence from the designer than this parametrized approach. Sensors on robotic snakes were also developed to take pictures of the environment to generate a 3D map. This could be considered an iterative and flexible self-learning approach but it doesn't offer the guarantee of convergence to an optimal policy that a true RL approach (i.e. Q-learning in this case) does. RL has been applied to optimize mechanical parameters such as joint angles and screw sizes in the robot itself ("Robot Control", 2016). Regardless, the need exists to utilize reinforcement to learn optimal movement and robot parameters in new environments, while minimizing data consumption and usage.

No physical fabrication needs to be done for the project. However, the snake model robot as described above needs to be fabricated in a physics simulator using body segments and double pin joints. Different environments will also be created in order to test the algorithm in a wide range of situations to test our claim that RL algorithms offer better performance in unknown situations. Some environment parameters that we will be varying include incline/decline, friction coefficient, and surface material.

Constructing the simulated snake robot will be fairly easy as there are only 2 unique components for this model. The Q-learning algorithm is already well understood as it has been a central research topic for two decades in the RL community so there exists numerous resources that can assist us with implementation of the actual algorithm (Schaul, 2010). The main challenge faced will be the potential for a slow convergence to an optimal policy. This convergence must occur within a reasonable number of trials (i.e. reasonable time frame on the order of days to hours in real time) or else the benefits of the RL approach will be severely diminished. Running a large volume of training trials on a physical robot could cause unwanted damage to the robot and could take a significant amount of time. Keep in mind that the physical robot cannot speed up time like a simulation on a computer can. Overall, this general RL interface applied to our specific snake robot will meet all of our design goals assuming that the issue of convergence does not become a major hindrance.

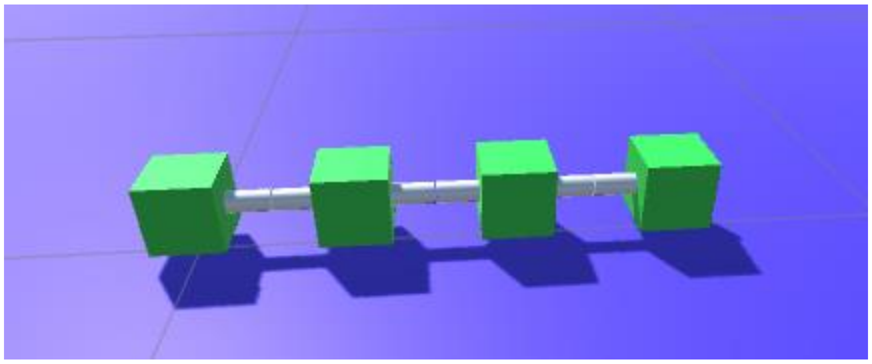
All potential simulation software packages being used are **free** with most of them being open source. An example of a free simulation software is V-Rep, which is free to download and install at www.copelliarobotics.com. A computer capable of running the simulation software (any modern personal laptop or desktop) is the only potential cost. Computers, however, are available to us at no cost through existing ownership or through the university. Therefore, the design requires no monetary cost. If the algorithm were to be implemented on a physical system then the robot would cost a significant amount of money to build, but the hardware aspect of the robotic system is outside the scope of this project.

Time to troubleshoot algorithms could be sped up significantly if we were to use a supercomputer to run simulations thus allowing for convergence to an optimal policy in a significantly shorter time. This wouldn’t reduce the number of trials to reach convergence just the time needed to simulate those trials. This may be a viable backup option if simulation time is a major bottleneck during our troubleshooting process. Whether or not this will be necessary is to be determined but it remains a potential backup resource. Access to a supercomputer costs as little as $100 an hour but free access to the university's supercomputer can be granted (Allan, 2016).

Parts and Resources Summary

Item	**Cost (and relevant information)**
Computer	Free (through existing ownership)
Robot simulation software	Free (open source available)
Physics simulation software	Free (open source available)
Optional: supercomputer access	$100 (per hour)

Implemented Design Solution



Q-learning is a simple algorithm that only functions efficiently when implemented alongside an appropriate state-action space representation. Pairing Q-learning with a function approximation technique such as Incremental Feature Dependency Discovery (IFDD), which will be described in detail later on, can also improve performance. IFDD has only been show to work reliably with a state-action space that has a hundred million state-action pair. Therefore, we will set a 250 million state-action pair upper limit on our snake's state-action representation (Geramifard).

Additionally, the state-action space representation must approximate a Markov Decision Process (MDP) such that the state-action representation captures all relevant state information. This means that the information encoded in the state-action representation must convey enough information to act as a predictor for what will happen to the snake model when an action is taken (e.g. position, velocity, acceleration, etc.). However, including more information in the state-action representation increases the number of state-action combos. Therefore, an efficient algorithm implementation would have a state-action space representation that includes enough information to sufficiently approximate an MDP while minimizing the state-action space size. This places severe limits on creating a tractable snake model and state-action representation.

The goal for the algorithm is to produce movement along a single axis parallel to a uniform plane. This was changed slightly from the initial design proposal where the goal was to have the snake model move to a point located away from its starting location. This change makes testing easier since we can now perform infinite non-episodic tasks. Furthermore, the state-action space representation need not include coordinate information since the environment is uniform. The only information that needs to be captured in the state-action space representation is the snake's local orientation and global orientation, or in other words its current body conformation and how it is oriented with respect to the desired global axis of movement.

To encapsulate local orientation information several quantities need to be measured such as local joint angles, joint angular velocities, and joint angular accelerations. However, we only measured the local joint angles in order to reduce the size of the state-action space representation. In order for this representation to sufficiently approximate a Markov Decision Process we had to reduce the effect of these quantities on the physical model. This was achieved by selecting an appropriate time step between action choices. Movement of the snake is powered by setting target joint angles which then becomes the setpoint for a spring/damper system that applies force to these joints. We picked a large enough time step between action choices (0.1 s) so as to allow the system enough time to reach the angle setpoint, thus reducing the angular joint velocity and angular joint acceleration at the time of the next action choice. This time step is also small enough to allow the snake movement to appear continuous, which is purely an aesthetic design choice.

To encapsulate global orientation information, we created a 3-D orientation vector that tells us how the snake is oriented in global space so that it may learn to move in a desired global direction. We do not include any state information describing how this 3-D orientation vector changes with time (i.e. angular velocity or angular acceleration of this 3-D orientation vector) even though it affects how the snake model will move in the environment. Once again this is purposely done to reduce the state-action space size and we must reduce the effect of these ignored quantities in order to sufficiently approximate a Markov Decision Process. The previously mentioned time step design choice reduces these ignored quantities for the same stated reason. Additionally, we decided to change the initially proposed spherical snake body segments to cubes. This change prevents the snake from rolling and gaining significant angular velocity around the long axis of the snake body which could otherwise dominate the snake's movement.

Adding more joint degrees of freedom combinatorically increases the state-action space size. We want to maximize the number of joints in the model to make the simulated snake more "snake-like". We also decided to reduce the number of joint degrees of freedom per joint to 1, thus allowing us to add more joints while maintaining the same state-action space size.

Next, we will describe specifically how the snake's continuous state and action variables will be discretized to make locomotion learning tractable. First, we limit each local joint angle's 1 degree of freedom from -108 degrees to 108 degrees and discretize this range in increments of 36 degrees (i.e.7 total discretizations per joint degree of freedom). Each component of the 3-D global orientation vector can range from 0 to 360 (non-inclusive) degrees and discretizing this by 36 degrees gives us 10 total discretizations per orientation vector component. Lastly, to limit the size of the action space we only allow 3 actions per joint degree of freedom: +36 degrees, + 0 degrees, and -36 degrees. Below is a table that shows the size of the state-action spaces for different numbers of joints, which is computed by finding all the possible combinations of these state-action dimensions.

Number of Joints	State-Action Space Size Calculation	State-Action Space Size (i.e # state-action combos)
N	(7)^N * (10)^3 * (3)^N	-
1	(7)^1 * (10)^3 * (3)^1	21,000
2	(7)^2 * (10)^3 * (3)^2	441,000
3	(7)^3 * (10)^3 * (3)^3	9,261,000
4	(7)^4 * (10)^3 * (3)^4	194,481,000
5	(7)^5 * (10)^3 * (3)^5	4,084,101,000

As you can see an addition of a joint increases the size of the state-action space combinatorically which is a testament to the curse of dimensionality. This chart shows that we can only realistically explore snakes with up to 4 joints since 5 joints exceed our 250 million state-action combo limit. Most of our testing and evaluation was done using a 3-jointed snake since it has sufficient complexity while having a tractable state-action space size.

Next, the implementation of Q-learning will be discussed. The initial design proposal did not go into specifics about how it would be implemented. First, we performed a non-episodic task so we will use a discount factor of 0.5 to avoid divergence of the Q-function. Second, since we will be performing online learning, the snake will follow the policy that maximizes the current estimate of the Q-function (i.e. it will greedily follow the best estimate of the optimal policy available given all past data samples). Theoretically, the policy should improve over time as more data is observed and collected. Lastly, we will address the issue of exploration vs. exploitation by adding in a condition that with a 0.05 probability the snake will choose a random non-greedy action. This will encourage a small amount of exploration of potentially unexplored or underexplored actions.

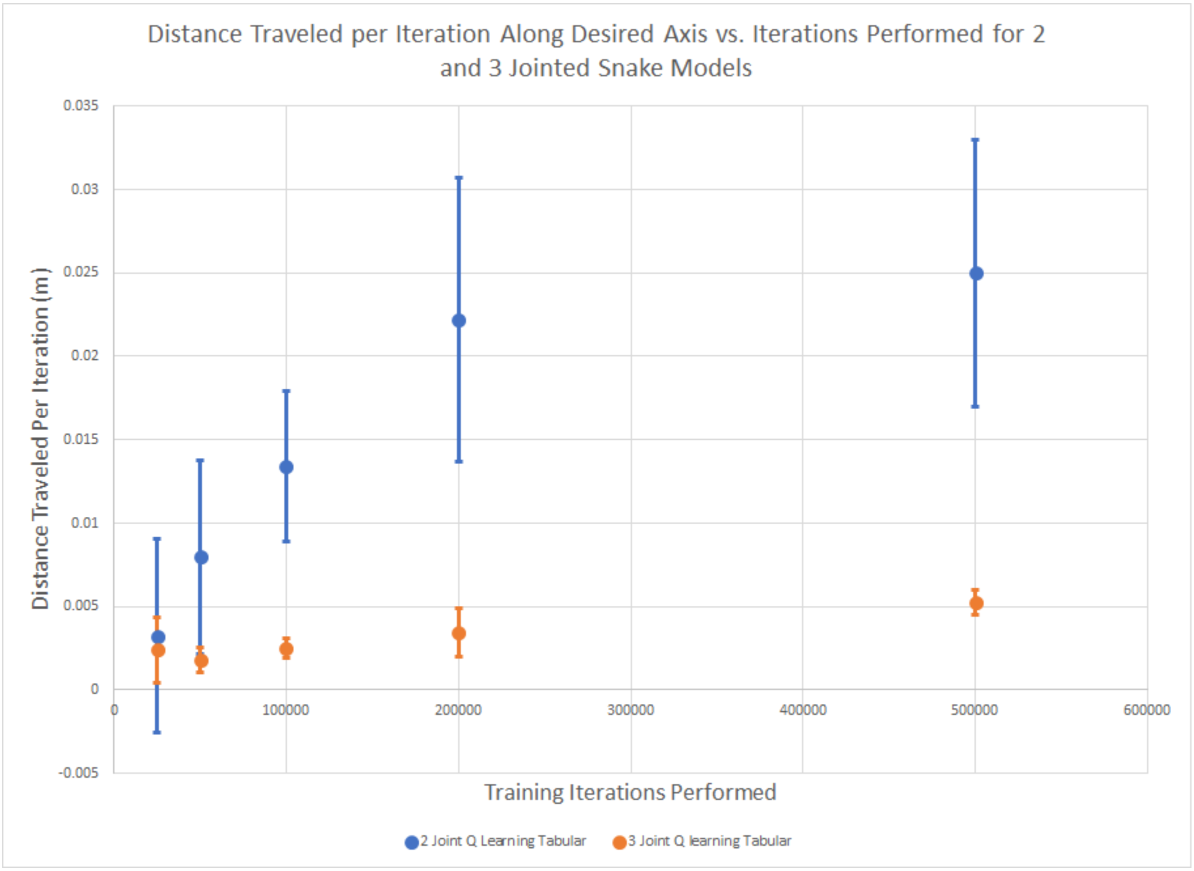
Lastly, we coupled Q-learning with a function approximation technique to improve performance and reduce the amount of information we must store for the Q-function estimate. Tabular Q-learning simply stores its Q-value estimate for a given state-action pair in a unique bin. This means that visiting one state-action pair will not affect the estimate for another state-action pair even if they are very similar. We want to leverage as much information from an observed experience, which we can do by generalizing each experience across many different state-action pairs using function approximation. However, we also must make sure that we do not over-generalize otherwise we will get a poor non-converging Q-function estimate. Q-learning with Incremental Feature Dependency Discovery (IFDD) finds a balance between these extremes while combinatorically reducing the amount of stored information needed to estimate the Q-function (Geramifard). IFDD begins by storing one weight per dimension discretization. The Q-function value is then produced by linearly combining the weights of the dimension discretizations present in the given state-action pair. However, for this initial set of weights to yield an accurate approximation of the Q-function the state dimensions must be linearly independent from one another which often isn't the case in RL problems, especially when dealing with complex systems like a jointed snake model. IFDD deals with this by gradually expanding the number of features, or in other words IFDD adds weights to the linear combination that compensates for nonlinearities in the Q-function estimate. Adding new features to the state-action representation is controlled by keeping track of the cumulative error of all possible feature conjunctions. If the cumulative error exceeds a user defined threshold (i.e. relevance threshold) then a new feature is added. This allows the learning algorithm to generalize across states, correct over generalization through an expansion of more precise conjunctive features, and reduce the amount of necessary stored information to represent the Q-function. While IFDD requires more computational power to run than Tabular Q-learning it is still an easy-to-run algorithm with computationally simple operations. It should also be noted that the exact choice of the relevance threshold value is not super critical to successful performance.

In total, this design implementation can run the following algorithms: random policy (i.e. snake chooses random actions), Q-learning tabular, and Q-learning with IFDD function approximation. All required parameters for these algorithms to function can be edited and changed by the user to tailor the Unity program to their specific task. The previously described implementation of the Q-learning algorithm is specific to our snake model. However, these algorithms can be applied to any jointed 3-D model in the Unity program. Therefore, this generalized interface can be used by researchers to test the specific implementation of these algorithms without having to implement any of their own code, thus making this Unity script widely applicable outside of our project.

Evaluation and Testing

Dependency on user input, optimal solution:

The success of the underlying algorithms in the platform are heavily dependent on the model and its state-action space representation. This is because algorithm performance is directly dependent on the size of the state-action space, which it must explore thoroughly before reaching a near optimal locomotion policy. In that respect, the platform is highly dependent on user input. Figure # clearly shows the performance difference that occurs when the state action space is increased from 441,000 state-action combinations for the 2-jointed snake to 9,261,000 state-action combinations for the 3 jointed snake.



However, if we define user input as only the necessary parameters that the user must enter for the algorithms to function efficiently (i.e. learn within a reasonable time frame on the scale of hours) then this dependence is much lower. In fact, performance should be consistent as long as parameters such as discount rate, learning rate, and relevance threshold are within certain ranges (Geramifard). For example, choosing the discount rate for the non-episodic task between 0 and 1, the learning rate between 0 and 1, and the relevance threshold to be greater than 0 may improve the performance but not to a significant extent. However, further testing must be done in order to quantify the extent to which parameter choice affects the performance.

<u>Computational power needed to run in real time:</u>

Satisfying this criteria is difficult since quantifying computational power is rather vague. However, we can make some useful conclusions based on how the algorithms performed in our physics simulator. We used a stopwatch module to time how long each algorithm took to run per iteration. Keep in mind the time step between iterations is 200 ms, so as long as the algorithm's execution time is considerably less than this value then there should be no issue running this algorithm in real time on a physical robot. Tabular Q-learning was clocked at less than 1 ms per iteration, which is less than the precision of the tool. Q-learning with IFDD, which requires more operations than Tabular Q-learning, was clocked at about 2-3 ms per iteration, which is still considerably smaller than the 200 ms threshold. Therefore, as long as the physical robot does not contain 2 orders of magnitude less computational power than a standard laptop, which is an unlikely scenario, then the algorithm should run with no issue.

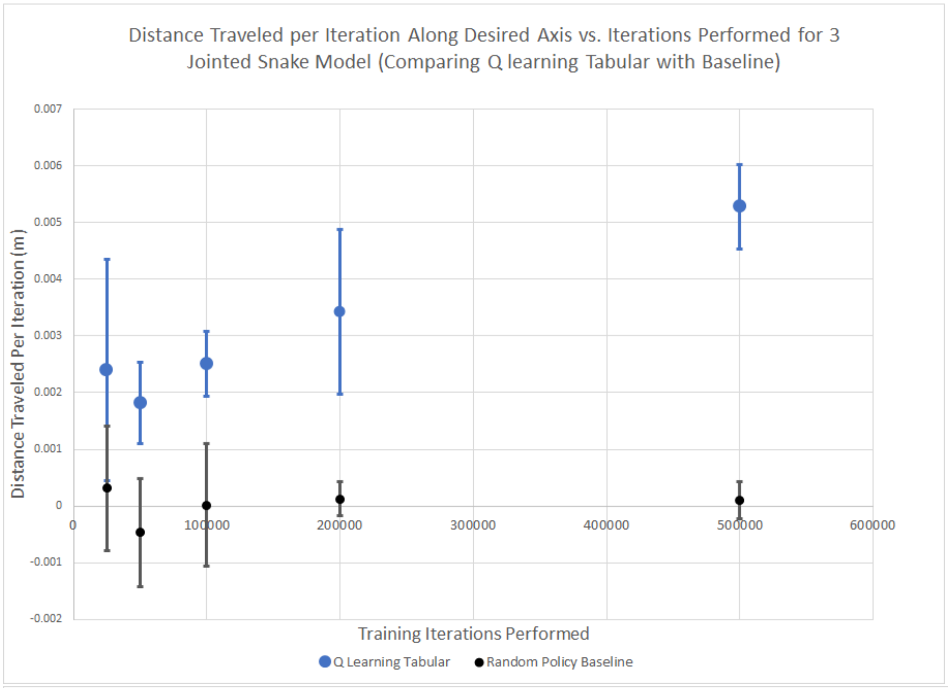
Development cost:

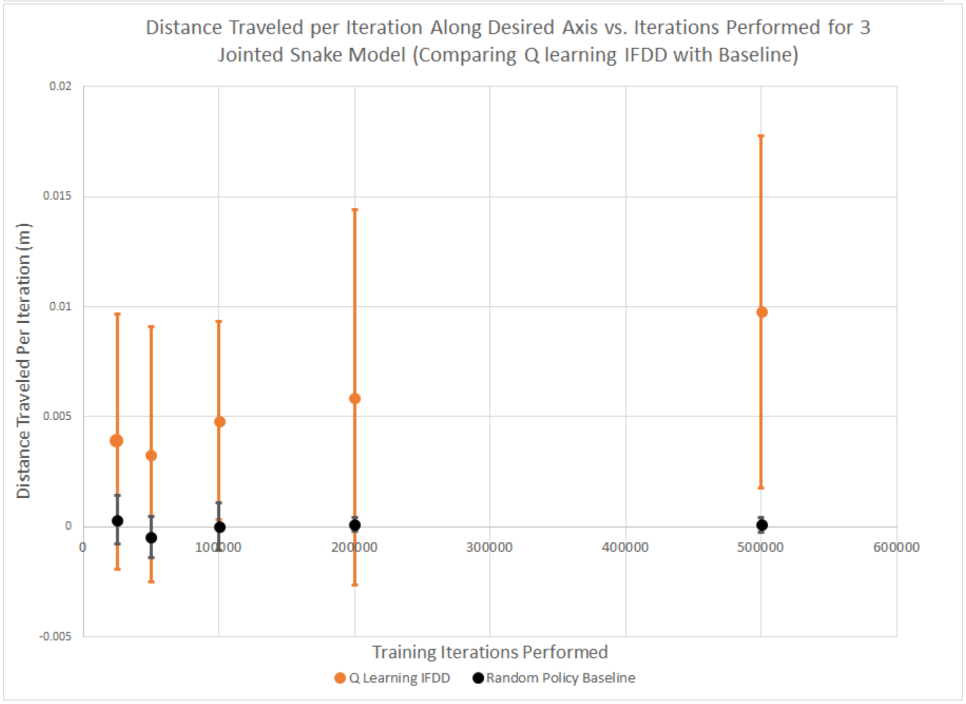
The development cost required no purchasing of physical materials or software. The only cost was the man hours put into researching, coding, troubleshooting, and testing, which are estimated to be at 200 hours. Since this Unity script can be easily applied to any jointed 3-D model, the development cost for a potential researcher investigating these algorithms applied to their model would be very low in terms of man hours since everything is already coded and implemented.

Quality of the locomotion policy:

The quality of the locomotion policy is the most important factor to consider for this implemented solution. Two separate tests were performed on the application of our learning algorithms to our snake-model. First, standard Q-learning and Q-learning with IFDD were both compared to a baseline random policy using a one-tailed t-test in order to confirm that these algorithms do in fact yield policies that learn from experience and perform the desired task. Second, Tabular Q-learning was compared to Q-learning with IFDD to justify the addition of the function approximation technique. State-space discretizations and input parameters such as learning rate, discount factor, and relevance threshold were kept constant so as to control for their effects. Investigation into the impact of these parameters will be performed at a later date.

The first test to validate the occurrence of learning for each algorithm by using a one tailed t-test with a random policy as a baseline. Each policy ran for 6 trials for a desired number of iterations, where one iteration is defined as a single action choice. Or in other words, one iteration represents a single training sample observed at each discrete time step. At the beginning of each trial the snake started with no stored knowledge and at the end of each trial the distance traveled along the desired axis defined by the reward function was measured (i.e. positive distance means the snake travels in the correct direction and negative distance means the snake travels in the wrong direction). We then normalized the distance by the number of iterations in that trial since more iterations yield more time for the snake to locomote. We are concerned with the normalized distance, or in other words the average distance traveled per iteration, so we can compare across iteration numbers. In theory, if learning occurs the average distance traveled along the desired axis per iteration should increase beyond that of the random baseline policy since the learning algorithm is built to maximize this quantity. Therefore, the null hypothesis for the one-tailed t-test will be that the learning algorithm has the same average normalized distance quantity and the alternative hypothesis is the learning algorithm has a higher normalized distance quantity. A standard p-value cutoff of 0.05 will be used. Lastly, in order for the one-tailed t-test to be valid we must assume that the normalized distance quantities for a given iteration number are normally distributed, which is a fair assumption since the observations occur in a continuous domain (Lifshits). Figure # and # show the average normalized distance (i.e. distance per iteration) and standard deviation vs. iteration number for the two Q-learning algorithms and the baseline random policy.



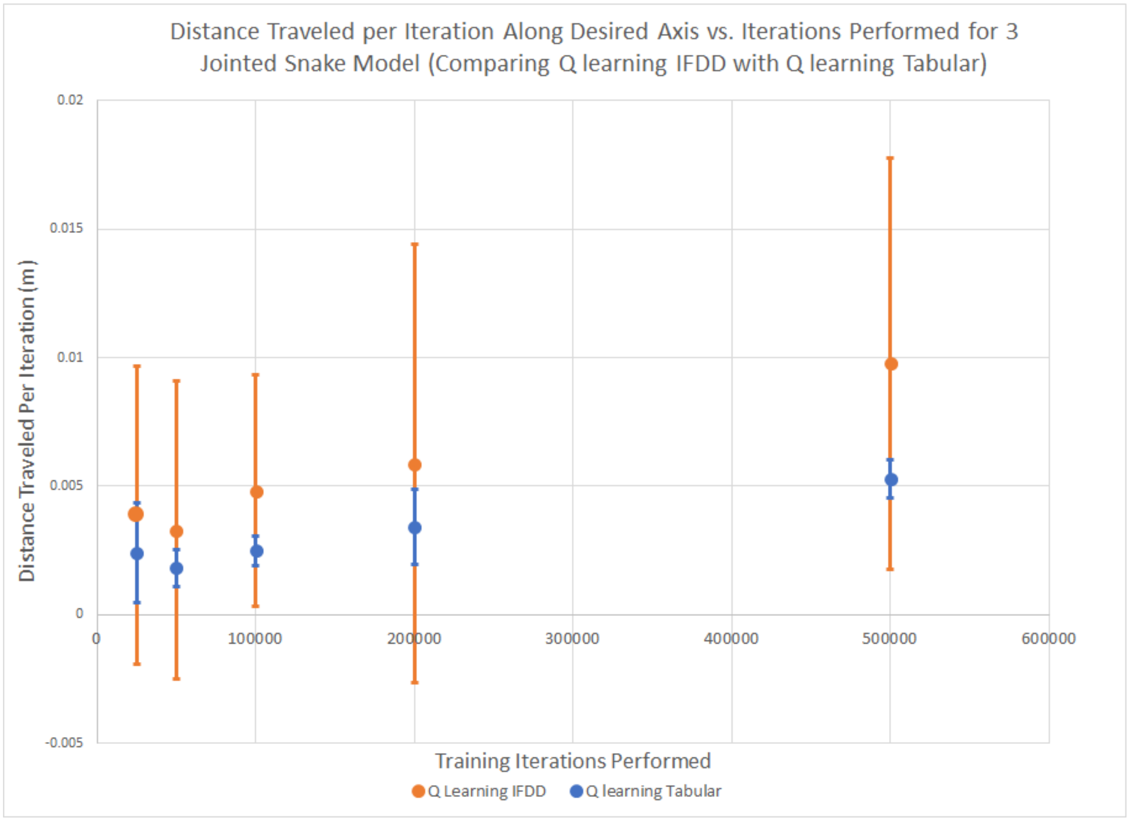


It is clear that the two Q-learning algorithms perform significantly better than the baseline random policy and their performance improve with more iterations (i.e. more training data). The following one tailed t-test will confirm that both learning algorithms perform better than the baseline, thus showing that these algorithms do in fact learn and improve through trial and error interactions with the environment.

One-tailed t-test comparing average normalized distances between 2 learning algorithms and a baseline policy		
Iterations Performed #	Q learning Tabular p-value with respect to random policy	Q Learning IFDD p-value with respect to random policy
25000	0.018924	0.078497
50000	0.000431	0.011067
100000	0.000291	0.009753
200000	0.000075	0.000462
500000	< .00001	0.005532

Each p-value is below the threshold value with the exception of Q-learning with IFDD at 25,000 iterations. Even though we cannot reject the null hypothesis for this example, overall it is quite apparent that learning does occur for both Q-learning with IFDD and Tabular Q-learning since for the majority of examples we do reject the null hypothesis.

Next, we will compare the performance of the two Q-learning algorithms. According the figure #, Q learning with IFDD appears to perform better the standard Q learning, which will be tested with a one tailed t-test below. The null hypothesis will be that both algorithms have the same average normalized distance, and the alternative hypothesis will be that Q learning with IFDD has a higher average normalized distance (i.e. it performs better). Once again, a standard p-value of 0.05 will be used.



One-tailed t-test comparing average normalized distances between Q learning with IFDD and Tabular Q learning	
Iterations performed #	Q Learning IFDD p-value with respect to Q learning tabular
25,000	0.277926
50,000	0.159342
100,000	0.081913
200,000	0.061195
500,000	0.029495

We can only reject the null hypothesis for the 500,000 iteration example and conclude that Q learning with IFDD does perform better than Tabular Q Learning at this level. Q Learning with IFDD probably perform better at lower iteration numbers too but more data must be collected before we can make this statistical conclusion. Taking into account that we can reject the null for one case, that the p values are low but not necessarily past our set threshold for each iteration level, and the observation that the Q learning with IFDD average data points are larger than their Tabular counterpart we will state that Q learning with IFDD can be considered the overall better performing algorithm with a fair amount of confidence.

In conclusion, we can conclude with the following succinct statements from the one-tailed t-tests: Q learning does in fact learn from experience, Q learning improves its policy as more training data is observed, and Q learning with IFDD function approximation generally performs better than standard tabular Q learning.

Environmental, Social, Ethical, Health & Safety Issues

Safety is not much of a concern for our project since it mostly involves using a computer to develop and simulate an algorithm. However, when the algorithm is implemented on the physical robot, we will need to prevent the robot from harming people and from damaging its immediate environment. While the burden of safety lies mostly on the designers and operators of the physical robot, some safety features should be included in the algorithm controlling the robot. For example, a kill switch can be activated if the robot senses that it is in a state where there is a high chance of it causing damage or harm.

Environmental protection is not much of a concern since there are no biologically active components to our project. However, this might become a slight concern during the robotic testing phase since the robot could move in erratic ways and cause damage to nearby plants and animal habitats. Concerns over **biocompatibility, FDA regulations, and sustainability** do not apply to this project since there are no biologically active components or connections to public health and ecology.

Public acceptance of self-learning robots is currently mixed. Products such as Sony's AIBO and Amazon's Cortana seem more like a new kind of toy to the general public. On the other hand, products such as smart homes raise questions about the appropriate level of integration of these "smart" machines into our lives and the tradeoff between convenience and privacy. Developments in industrial robotics and self-driving cars are having an impact on a larger societal level, changing the very way we live our lives. People, such as Steven Hawking and Elon Musk, are weighing in on this conversation, calling artificial intelligence the next great threat to humanity (Hawking, 2014 & Markoff, 2013). There is no clear consensus on how the public will react to the goal of our project, and they may not even be aware of it until a later time when it has been integrated into consumer products. However, similar to previous technological breakthroughs, the public's reaction will be based on how the technology affects their immediate lives and its philosophical implications.

Legislation around self-driving cars is opening up the debate regarding the **ethical standards** governing these self-learning machines and tackles the question of who is liable in the case of an accident (Young, 1997). Although specific answers may seem unclear, principles such as preserving the well-being of individuals and using this technology in ways that aren't destructive should be guiding the applications of these machines.

Discussion and Conclusions

The original purpose of this project was twofold. First, it was to create a general platform for which the Q learning algorithm could be tested on 3-D models of robots with the purpose of creating optimal locomotion policies. This testing could ultimately serve as a feasibility test for the algorithm and the associated state-action space representation before putting encoding it on a physical robot. Alternatively, it could serve as a method to pretrain the snake's locomotion policy before putting it on the robot. Second, the original purpose was to apply this platform to a snake-like robot model in order to try and build a near-optimal locomotion policy that fit within the design constraints put forth in our design goals.

The general platform built in Unity is extremely easy to use in terms of installation and hardware/software requirements. Unity can run on a wide range of operating systems and can run on systems with limited hardware such as smart phones. As previously, discsused the Q learning algorithm requires minimal computational power to run in real time. In this sense, the platform meets the design goals that require the platform to be easily accessible and usable to researchers (i.e. low development cost and low computational power requirements). However, it was quickly discovered through our investigation into the snake-like model that the successful implementation of the algorithm requires the user to develop an appropriate state-action space representation of the model, thus creating a dependency on user input. Considering that the intended user of this platform is a researcher who has some understanding of reinforcement learning this task of creating an state-action space representation should not be a burden.

Additionally, we also found that the time to convergence and quality of locomotion policy are also heavily dependent on the 3-D model being employed and its underlying state-space representation. For example, as shown in figure # a 2-jointed snake performs significantly better than a 3 jointed snake and even appears to have a convergence in performance near 500,000 iterations (27 hours of real-time operation) of training samples. There is not strict criteria to define when convergence of the policy and the performance occurs so we will simply say that if the performance curve appears to approaching an asymptote, as shown in figure #, then convergence is occurring. Our design criteria stated that convergence should occur on the scale of hours, thus we showed that this platform has the ability to converge to a near optimal within a timeframe specified by our design goals. However, this convergence was not observed in the 3-jointed snake with 500,000 iterations presumably because its state-action space is 21 times larger. Therefore, we can make the conclusion that the quality of the locomotion policy and time to convergence are heavily but not totally dependent on the model and underlying state-action space representation. We were able to satisfy these two design criteria for our 2-jointed snake-model but this might not be the case for researchers who wish to use this platform for their more-complicated models. It is worth noting that we were able to improve performance described by these two criteria through purely algorithmic means using IFDD function approximation techniques, thus showing an ability for this platform to efficiently leverage information through generalization and an over-generalization fixing mechanism. Overall, if we control for the state-action space size of the model (i.e. if the state-action space size is smaller 500,000 state-action combos), the we can ensure that the design goals requiring an appropriate time to convergence and locomotion policy quality will be satisfied. Lastly, as previously described the computational complexity of the Q learning algorithm and Q learning with IFDD should not be a problem as the runtimes of the algorithms on a standard laptop (<1ms and 2-3 ms respectively) are significantly less than the time step (200 ms), which should satisfy for any implementation the criteria that the algorithm must be able to run in real time.

Now the question becomes how useful will this platform be to a researcher. This program does not enable users to test and/or train all-encompassing algorithms for robotic locomotion since the underlying algorithms being used are limited in their scope. The curse of dimensionality severely constrains what these learning algorithms can and cannot do, and thus it constrains what this program can do. We were only able to test the snake model in uniform environments and we were only able to learn simple tasks such as moving along a single axis. These algorithms are certainly not robust enough to create full locomotion policies that will succeed in varying complex environment present in the real world. However, if the limitations and limited scope of applicability are acknowledged then this platform can serve as a highly useful and easy-to-use tool that can test and develop preliminary locomotion policies or Q learning algorithms which can then be altered and tailored for the final robot. For example, let's say we test the model and get a resulting locomotion policy. We could implement this locomotion policy onto the actual robot as an initial policy that the robot can then improve upon using more complex algorithms according to its changing conditions. In this example. the program will essentially serve as an initial training mechanism that can simulate the environment much faster than real time. While this is not its original intended use it can certainly be repurposed for this. Alternatively, we could use the resulting policy to find highly complex and efficient movement sequences that perform specific tasks such as turning or moving quickly in a straight line. Coding these complex movement sequences onto the robot so that they are only activated when the robot observes a specific situation or environment could prove to be much more efficient than hard-coding complex movement patterns by hand. Overall, while the results produced by this program are limited, researchers can still use these results in ways originally intended and not originally intended when constructing their physical robot's method of locomotion. At its core, this program can provide information to researchers which can be leveraged in specific ways to assist in the development of locomotion software.

Now let's focus on how well this platform functions with respect to our snake-like application. Keep in mind we choose to use the 3-jointed snake as our central test case due to the sufficient complexity of the model and reasonably size state-action space. The final locomotion policies produced using Q learning paired with IFDD function approximation allowed the snake to move with surprising efficiency in the desired direction. There is no absolute standard to compare the locomotion policy to in order to judge the quality of the locomotion policy. We can only conclude that the locomotion policy did in fact improve from trial and error interactions with the environment (i.e. learning did occur) within a reasonable amount of time. Convergence of the policy to near optimal within a reasonable time was not observed for the 3-jointed snake model. Improvement of the function approximation method could accelerate the learning process but nonetheless the state-action space size is a determining force in the convergence time. In short, a learning algorithm regardless of its specifics theoretically takes longer to explore and learn in a larger state-action space.

Lastly, the successful implementation of the IFDD function approximation technique with Q learning should be highlighted as one of the major accomplishments of this design. The curse of dimensionality has already established the fact that for reinforcement learning to be successful for high dimensional agents observed data must be leveraged to extract the most useful information as possible. IIFD successfully generalized information from observed data while also reducing the memory needed to store the Q function. Further development into the function approximation aspect of this project should yield the most impactful results since other aspects of this project such as state-action space representation and the state-action dimensionality of the 3-D model are out of our control and in the hands of prospective researchers wishing to use this platform.

Recommendations

There are certain algorithmic tweaks and additions we would like explore as well as alterations to the snake model we would like to explore. AS previously mentioned we would like to explore how changing some of the algorithm parameters affects performance in order to determine how dependent the algorithm is on these quantities. We would like to do testing with more joints (4+) and higher orientation vector discretization resolution. We would also like to implement algorithms for IFDD that can better handle the inherent stochasticity in the physics environment. Lastly, we would like to try and implement a crude obstacle detection system that would allow the snake to learn how to deal with obstructions in their path.

Work Breakdown

Jared Buchanan: implemented, tested, and coded snake model in Unity.
Steven Tran: implemented, tested, and coded snake model in Unity.
Donald Dean: conducted background research.

Bibliography

Allan, Darren. "Renting a Supercomputer Costs You as Little as £90 an Hour." TechRadar. TechRadar Pro IT Insights for Business, 05 Oct. 2016. Web. 04 Dec. 2016.

"An Examination Of Software Patents | USPTO". Uspto.gov. N.p., 2016. Web. 10 Nov. 2016. 8. Schaul, Tom, et al. "PyBrain." *Journal of Machine Learning Research* 11.Feb (2010): 743746.

Bertsekas, Dimitri P., and Huizhen Yu. "Q-learning and enhanced policy iteration in discounted dynamic programming." *Mathematics of Operations Research* 37.1 (2012): 6694.

Csáji, Balázs Csanád, and László Monostori. "Value function based reinforcement learning in changing Markovian environments." *Journal of Machine Learning Research* 9.Aug (2008): 1679-1709.

Geramifard, Alborz, et al. "Online Discovery of Feature Dependencies." *Massachusetts Institute of Technology*,, 2011. Web.

Geramifard, Alborz, Christoph Dann, and Jonathan P. How. "Off-policy learning combined with automatic feature expansion for solving large MDPs." *Proc. 1st Multidisciplinary Conf. on Reinforcement Learning and Decision Making*. 2013.

Lifshits, Mikhail. "Lectures on Gaussian processes." *Lectures on Gaussian Processes*. Springer Berlin Heidelberg, 2012. 1-117.

Lu, Zhenli, et al. "Study on the motion control of snake-like robots on land and in water." *Perspectives in Science* 7 (2016): 101-108.

Owen, Tony. "Biologically Inspired Robots: Snake-Like Locomotors and Manipulators by Shigeo Hirose Oxford University Press, Oxford, 1993, 220 pages, incl. index (£ 40)." *Robotica* 12.03 (1994): 282-282.

"Patenting Software". Wipo.int. N.p., 2016. Web. 10 Nov. 2016.

Ponte, Hugo, et al. "Visual sensing for developing autonomous behavior in snake robots." *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014.

"Robot Control". *Clear.rice.edu*. N.p., 2016. Web. 10 Nov. 2016.

Saito, M., M. Fukaya, and T. Iwasaki. "Modeling, analysis, and synthesis of serpentine locomotion with a multilink robotic snake." *IEEE Control Systems Magazine* 22.1 (2002): 64-81.

Sanchez, Carlos J., et al. "Locomotion control of hybrid cockroach robots." *Journal of The Royal Society Interface* 12.105 (2015): 20141363.

Stamper, Sarah A., Shahin Sefati, and Noah J. Cowan. "Snake robot uncovers secrets to sidewinders' maneuverability." *Proceedings of the National Academy of Sciences* 112.19 (2015): 5870-5871.

Sutton, Richard S., and Andrew G. Barto. *Introduction to reinforcement learning*. Vol. 135. Cambridge: MIT Press, 1998.

Technologies, Unity. "Physics." Unity - Manual: Physics. N.p., n.d. Web. 22 Mar. 2017. <https://docs.unity3d.com/Manual/PhysicsSection.html>.

Tully, Stephen, et al. "Shape estimation for image-guided surgery with a highly articulated snake robot." *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*. IEEE, 2011.

Wright, Cornell, et al. "Design of a modular snake robot." *Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on*. IEEE, 2007.