April 2016

Deep Q-Learning for Humanoid Walking

Alec Jeffrey Thompson
Worcester Polytechnic Institute

Nathan Drew George
Worcester Polytechnic Institute

Deep Q-Learning for Humanoid Walking

Submitted by:
Alec Thompson, Nathan George

Project Advisors:
Professor Michael Gennert, Professor Joseph Beck

April 28, 2016

Abstract

Existing methods to allow humanoid robots to walk suffer from a lack of adaptability to new and unexpected environments, due to their reliance on using only higher-level motion control with relatively fixed sub-motions, such as taking an individual step. These conventional methods require significant knowledge of controls and assumptions about the expected surroundings. Humans, however, manage to walk very efficiently and adapt to new environments well due to the learned behaviors. Our approach is to create a reinforcement learning framework that continuously chooses an action to perform, by utilizing a neural network to rate a set of joint values based on the current state of the robot. We successfully train the Boston Dynamics Atlas robot to learn how to walk with this framework.

Acknowledgements

We would like to thank Professor Gennert and Professor Beck for guiding us in understanding our project, giving feedback when necessary, and helping us reach realistic goals. We would like to thank Batyrlan Nurbekov for working with us in shaping our original Q-learning and Neural Network framework. We would also like to thank the member of the WPI Humanoid Research Lab for helping us learn to interface with Atlas, start the real robot, and guide us in understanding parts of our project.

Authorship

Section	Writer	Editor
Abstract	All	All
Acknowledgements	All	All
Chapter 1: Introduction	Nathan George	All
Chapter 2: Background	All	All
2.1: Boston Dynamics Atlas	Nathan George	All
2.2: Walking with Humanoid Robots	Nathan George	All
2.3: Robot Operating System	Alec Thompson	All
2.4: Robotic Learning	All	All
2.4.1: Q-Learning	All	All
2.4.2: Neural Networks	Alec Thompson	All
2.4.3: Software	Alec Thompson	All
Chapter 3: Procedure	All	All
3.1: Our Approach	Alec Thompson	All
3.2: Design Phases	Alec Thompson	All
3.3: Framework	All	All
3.3.1: Atlas I/O Layer	Nathan George	All
3.3.2: Q-Learning Layer	Nathan George	All
3.3.3: Neural Network Layer	Alec Thompson	All
3.4: High-Level Design	All	All
3.4.1: Action Representation	Nathan George	All
3.4.2: State Representation	Nathan George	All
3.4.3: Exploration	Alec Thompson	All
3.4.4: Issuing Commands	Nathan George	All
3.4.5: Reward Calculation	Nathan George	All
3.5: Mixed Level Design	All	All
3.5.1: State Representation	Alec Thompson	All
3.6: Low Level Design	All	All

3.6.1: Action Representation	Nathan George	All
3.6.2: State Representation	Nathan George	All
3.6.3: Action Selection	Alec Thompson	All
3.6.4: Exploration	Alec Thompson	All
3.6.5: Issuing Commands	Nathan George	All
3.6.6: Reward Calculation	Nathan George	All
3.6.7: Parallelization of Neural Network Updates	Alec Thompson	All
Chapter 4: Results	Alec Thompson	All
4.1: Q-Learning Framework	Alec Thompson	All
4.2: High Level Implementation	Alec Thompson	All
4.3: Mixed Implementation	Alec Thompson	All
4.4: Low Level Implementation	Alec Thompson	All
Chapter 5: Future Work	All	All
Chapter 6: Conclusion	Nathan George	All

Table of Contents

Table of Figures

Table of Equations

Chapter 1: Introduction

Many control systems for humanoid walking need to model the kinematics for multiple joints, use path planning algorithms for obstacle avoidance, trajectory optimization to avoid singularities, and possibly model a dynamic system for force-torque control. While for simpler robots, these methods are very effective, when the number of joints increase, the complexity makes the modelling become less accurate. It is also hard to adapt the systems to new environments because the motion must be pre-planned. Even with Simultaneous Localization and Mapping (SLAM), the motion the robot takes to advance its position is pre-planned. For most of these systems, they need exteroceptive sensor information, information about the environment, to adjust the robot path plans.

Instead of thinking as the robot as an entity to model and control, we propose to view the robot as an agent with multiple possible actions it could take, with the goal of maximizing a reward it is given. This reward is to move a certain distance, assumedly, by walking. Reward maximization is a common approach in the area of reinforcement learning. Previously, even attempting such a method as reinforcement learning for robots would be impossible as the number of possible actions—the action space—of the robot is too large, even if discretised; but with the advancement in processing with Graphical Processing Units (GPUs), such methods are plausible.

With this approach for humanoid walking, the input into the system is some sensor data, and the output is a set of joint angles that, at a particular moment in time, would advance the goal of walking. Now, simply choosing the best action replaces complicated control systems. We also do not need to model or analyze exteroceptive sensor data; we only need to pass it and proprioceptive data, information about the robot, into the system.

Chapter 2: Background

2.1: Boston Dynamics Atlas

Atlas is a humanoid robot given to WPI by the DARPA Robotics Challenge (DRC). WPI worked with Carnegie Mellon University (CMU) to develop a software system capable of doing challenging disaster recovery-based tasks, including walking over rubble, drilling, turning a door, turning a valve, and climbing up stairs [1] [2].

Wall power and a generator through a power conversion module power Atlas. Atlas also has wireless emergency and wired hard stops in case the robot performs undesired motions. For perception hardware, Atlas has a MultiSense-SL head, equipped with illuminators, stereo-vision cameras, and LIDAR; an IMU; and attachable hands. Internally, the robot has a 1 degree of freedom (DoF) neck, 6 DoF arms, 6 DoF legs, and a 3 DoF back [3], as depicted in Figure 1.

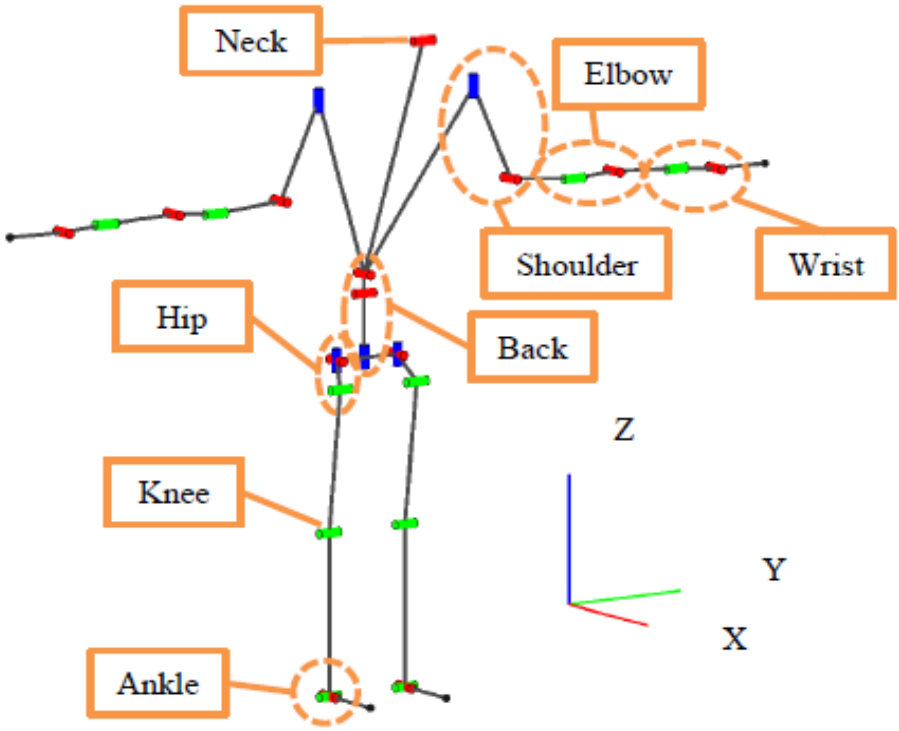

Figure 1: Joint System of Atlas

The legs, back, shoulders, and elbows are hydraulically activated, and the wrists and neck utilize electric motors [3]. Of these degrees of freedom, 15 are necessary for walking – both legs and the back.

2.2: Walking with Humanoid Robots

The task of walking for robots has always been an interesting and challenging problem. The majority of robotic frameworks, up to this point, have been wheel-based, as motors spin circularly. Walking, on the other hand, involves moving many muscles at various speeds and angles synchronously. Such a task is hard to mimic in a robot because humans do not fully understand the details of our gait. There have been successful attempts at humanoid walking; but the majority of attempts are isolated to flat terrain with awkward stepping patterns to minimize the chance of falling over.

Usually, these attempts try to keep the robot statically stable—at any point in time, if power to a humanoid robot was cut, the robot would not fall over. This is because the robot constantly keeps its center of gravity in the center. Humans, though, walk dynamically; when we walk, we assume that future movements of our body will prevent us from falling over. For example, humans lean forward when walking to maximize efficiency. Gravity propels us forward instead of just muscle movement; but if we suddenly stopped moving our joints, we would fall over.

One of the goals with our project is to learn how to walk dynamically. Currently, implementing statically stable systems for humanoid robots seem challenging enough. To bypass this, we look to how human infants walk—by trial and error. It seems they learn to walk by trying over and over; and eventually, after many attempts, they finally find the muscle memory that appears to work. We hope a similar strategy works with humanoids as well.

2.3: Robot Operating System

Robot Operating System (ROS) is an open source library covering almost all aspects of robot software, from drivers to control algorithms. It provides a framework to handle control of a robot's subsystems at both high and low levels and is the software that runs the Atlas robot [4].

2.4: Robotic Learning

Robot learning is a term used to describe concepts involving both robotics and machine learning. Machine learning is taking data, usually large quantities, discovering patterns in the data, and learning how to best utilize those patterns. Usually, using software involves taking input and producing output; while such tasks speed up processes, it does produces work similar

to what a human could do with a vast amount of time. Learning, on the other hand, aims to produce output that a human could not figure out.

Our project is based on the subfield of reinforcement learning that has an agent, or system, learn using a reward system. A critic, or an algorithm, determines which action is most optimal to perform in a specific environment based on various attributes that the final state of the agent should have. In our case, we want our agent Atlas to be in a final state of having walked a few steps without having fallen over. We will accomplish this using a Q-learning framework with a neural network to pick the specific actions.

2.4.1: Q-Learning

Q-learning is a reinforcement learning technique; and as mentioned previously, reinforcement learning tries to learn the best decisions in an environment based on some reward function. What makes Q-learning unique is that it aims to incorporate the traditional concepts of machine learning with newer ones. Q-learning essentially tries to find a state-action pair that will get an agent closer to its goal by not only incorporating current data but also data from the past to support hypotheses into the future. A state is how the environment currently is situated. An action is what an agent would do to change its orientation in the environment.

One example of Q-learning is illustrated through a door-based problem. It finds its current state-action pair by finding Q values for every possible state-action pair. If the state-action domain is too large, the domain would have to be discretized in some way. A Q value for a specific state-action pair is found by adding its previous Q value (the R value) to a learned constant multiplied by a future state-action pair—a state-action pair that logically follows from the current state-action pair—that has the highest current Q value already, as visible in Equation 1.

Equation 1: The Q-Equation

$$Q(state, action) = R(state, action) + \gamma * Max[Q(next\ state, all\ actions)]$$

For example, suppose we have rooms that can only access other rooms, specified by forward arrows, as visible in Figure 2.

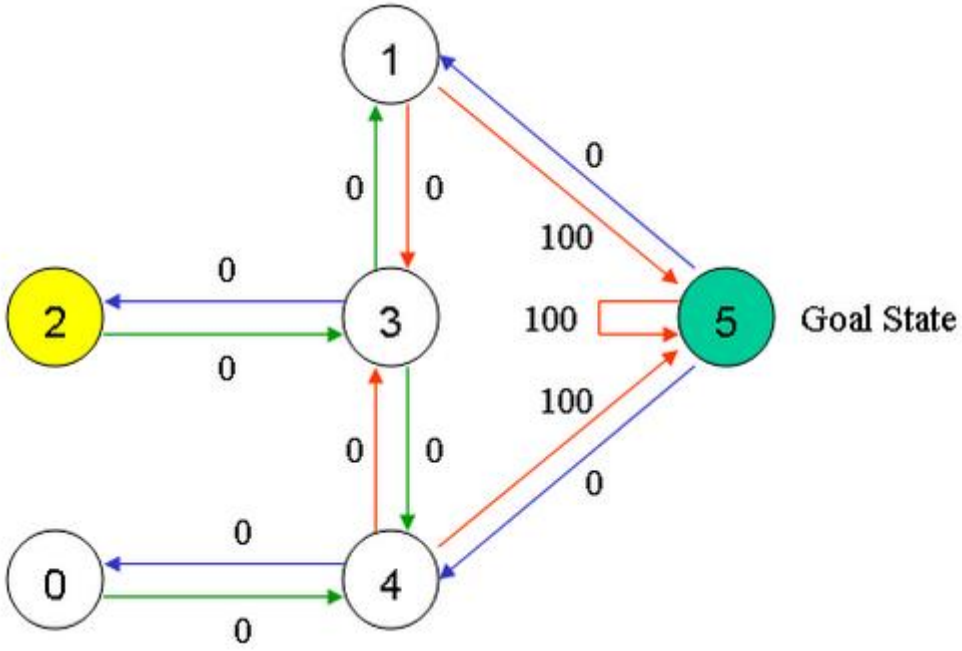

Figure 2: Example map with interconnected rooms, including path costs

Our ultimate goal is to get from state 2 to state 5 in the shortest amount of actions. To do so, we need to find Q values for all states, states 0-5. At the start, all Q values are initialized to zero and the R values are specified by the number next to the rays of the room diagram above (Figure 3, Figure 4). The ray values can also be specified by an R matrix.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 3: Initial Q-matrix

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Figure 4: Initial R-matrix

We can start with state one. Its possible actions are moving to state 3 and state 5. One is chosen randomly, which we can say is state 5 (Equation 2).

Equation 2: Q-Equation evaluating moving to state 5 from state 1

$$Q(1,5) = R(1,5) + 0.8 * Max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 * 0 = 100$$

The R value for (1,5) is 100. The learned constant is .8. The highest Q value among (5,1), (5,4), and (5,5) is 0. One might think (5,5) should give a value of 100 because going from state (5,5) has a reward of 100; but this is for the R matrix only. The Q matrix is still initialized to zero. Therefore, the equation leads to a final value of Q(1,5) = 100.

After multiple iterations for every state-action pair and after normalizing the learned values, the Q matrix converges, as depicted in Figure 5.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Figure 5: Final Q-Matrix

Then, using just this Q matrix, we can see that the best state-action list we can choose is (2,3), (3,1) or (3,4), and (1,5) or (4,5) [5].

This example differs slightly from an implementation that makes use of a neural network, for example when applied to a few Atari games such as Pong, Breakout, and Space Invaders. This Q-learning algorithm is based on a replay memory, or a set of experiences. An experience is the current state, current action, current reward for that action, and next state produced from the current action. For a chosen number of episodes—iterations of the algorithm—either choose an action with probability ϵ for some random small value or choose a_t using Equation 3.

Equation 3

$$a_t = max_a Q^*(\phi(s_t), a; \theta)$$

In this equation Q* is the optimal action-value function that tries to find a future action that will optimize the reward, $\phi(s_t)$ is the pre-processed RGB color data from the simulation, essentially the state of the game, *a* is an action, and theta consists of the weights for a neural network. In this case, the Q-network can be optimized by minimizing a series of loss functions $L_i(\theta_i)$ for each iteration *i*. The action is then executed in an Atari simulation so that the reward can be calculated and next image x_{t+1} can be made. Then the next state s_{t+1} is created by storing the current state, current action, and next image. The next pre-processing of the data is stored from the next state. An experience is now created by storing the current pre-processing, current action, current reward, and next pre-processing. (The pre-processing data are essentially the states.) The reward output y is then calculated and a step of gradient descent is taken to change the weights for the Q-network.

Equation 4: Gradient equation for Atari Q-Learning

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}\left[\left(r + \gamma \underset{a'}{max}\, Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)\right)\nabla_{\theta_i} Q(s, a; \theta_i)\right]$$

In Equation 4 ∇_{θ_i} is the gradient of the current weights, $\mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\varepsilon}$ is the floor function to apply to the estimation of the value of the action-state mapping, r is the reward, γ is a learned

constant, $\max_{a'} Q(s', a'; \theta_{i-1})$ is the optimal future Q value, and $Q(s, a; \theta_i)$ is the current Q value. The pseudocode for the Q-learning algorithm for Atari is found in Appendix A.

2.4.2: Neural Networks

While the Q-learning framework decides which action will be taken next, a neural network helps figure out correlations between input and possible actions. A neural network, conceptually adapted from the connections of neurons in the brain, can be used to approximate almost any multiple-input function. The nodes, similar to neurons, are connected by links, similar to synapses, of differing weights that simulate how important each connection is. These nodes are then arranged into layers, with a minimal neural network containing only an input layer and an output layer. The variant of neural networks that we are using is the deep neural network, which includes one or more hidden layers in between the input and output layers. These layers perform additional transformations on the input, and allow the network to approximate a wider range of increasingly complex functions [6]. A graphical rendition of a simple neural network is in Figure 6.

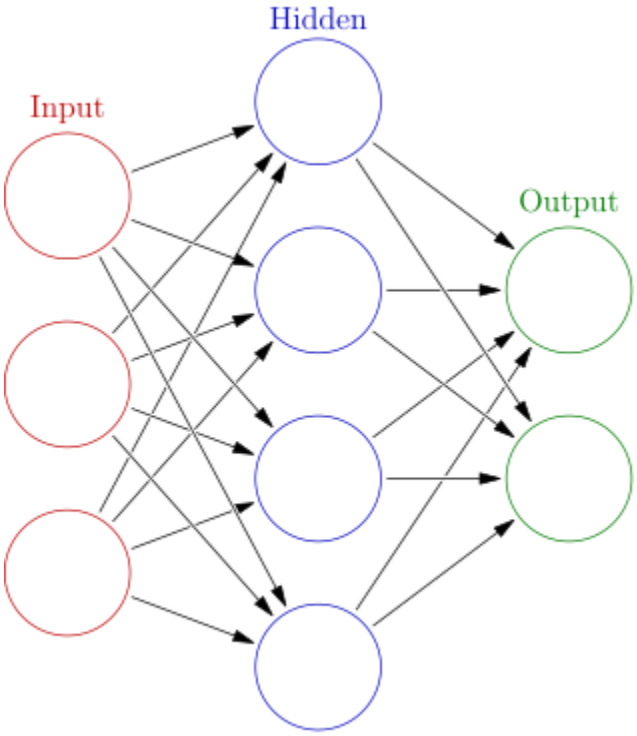

Figure 6: Representation of a neural network – Source: https://commons.wikimedia.org/wiki/User_talk:Glosser.ca

In the above rendition, the input data (in this case, three numeric values) would be passed into the input nodes. Then, with each connection, the value is multiplied by a weight and has a bias added. Each input to the hidden layer is summed. The hidden layer than applies the same process to generate the output values.

These can be used in many areas, from computer vision, such as handwriting recognition, to robot control as proposed in this paper. In the case of handwriting recognition, the input is an image of a number or a letter, and the output is a guess as to which number or letter the neural network thinks it is.

Neural networks are trained by either running them through labelled training data for a supervised neural network, or by running the network through training or actual data and then evaluating the resultant state with a reward function. The result of the action taken is then back-propagated through the network using a gradient descent, which is the derivative of some cost function, generally the difference between the output and the expected value from the labelled data or reward function. The ultimate goal is to minimize the cost function, meaning that the neural network's output for a given input matches the reward or data label for the given input [7].

Whether using a utility function or labelled data to fuel the backpropagation error setting, a neural network is designed to be a structure that can learn the importance of particular features of a dataset [8]. Its strength lies in that it operates on continuous data, meaning that even if the network has not been trained on a particular input, if the input is close to values it was trained on, then the output will be close to the output for the trained values.

Neural networks do have flaws – there is a risk of overtraining, where the network learns to become too specialized, and performs poorly on situations it has not been trained on. Additionally, deep neural networks are more difficult to train – the more layers the network possesses, the more difficult is to back-propagate updates to the earlier layers. It also is computationally expensive when run on a typical general purpose processor, though using graphics processing units (GPUs) alleviates this somewhat.

2.4.3: Software

To handle the neural network, we use the Python library Theano. This library adds features for implementing efficient neural networks by creating symbolic functions that can then be compiled into heavily optimized platform appropriate code. It represents the neural networks as a large multivariable function, which it then can compile at runtime into CUDA, allowing the

neural network computations to be performed on NVidia graphics cards, and provides performance far better than even hand-crafted C code designed specifically for optimal performance [9].

Chapter 3: Procedure

3.1: Our Approach

Our approach to implementing a learning system for the Atlas robot is focused on coupling the Q-Learning algorithm described in Chapter 2.4.1 with a deep neural network to estimate the Q-value of a given action from a particular state of the robot. A reward function is applied to calculate the reward for the selected action, and update the network. This allows the network to be trained directly with a simulation of the robot, and allows it to be run without human interaction, or tedious collection and labelling of data. Implementing these features will be done incrementally, with increasing complexity. With each increase in complexity, the neural network will be trained in simulation, and then tested on the physical robot.

3.2: Design Phases

The first step of implementing the learning system is to handle a high-level representation of the states of the robot and use high-level actions already created from previous teams. This will include identifying the state representations and actions necessary to walk, such as stepping and turning. The states are simplified to representations such as which leg is forward. In this process, the necessary software components will be implemented so that future modifications will be easier. Once the system is created, it will be trained using the Darpa Robotics Challenge (DRC) simulator and then tested on the physical robot. This phase will serve to test our implementation of the framework and ensure that the neural network can learn in a reasonable timeframe.

The second phase will be to maintain the high-level actions, but switch to using a lower level view of the state space – it will consist of values such as joint angles and angular velocities instead of high-level abstractions such as leg position. The framework we develop in the first phase is intended to be flexible enough that this step should be a simple modification of the components used for the high-level state representation, meaning most of the time spent in this phase will be training the new network in the simulator. Again, once simulator training is complete, we will test the system on the physical robot. This phase will determine whether the network is capable of extracting the necessary features from the low-level representation.

The final phase is to change the actions used. Instead of abstract commands such as “step forward with the left leg”, the program will be directly changing the position of joints. This will again, have a strong foundation from the software framework established in the first two phases, but will likely require substantially more training due to the massively increased number of actions.

3.3: Framework

Our implementation has three primary components. The first is our interface with Atlas (I/O Layer), which will construct a state from the robot’s sensors, and request actions to perform from the second component, the Q-learning layer. The Q-learning layer determines the best action by submitting each possible action to the neural network, and selecting the best rated one. The selected action is then handed back to the I/O Layer to be performed. The third and final layer is the neural network, which estimates how beneficial an action is, and learns from the results of performing an action. The components interact as specified in Figure 7.

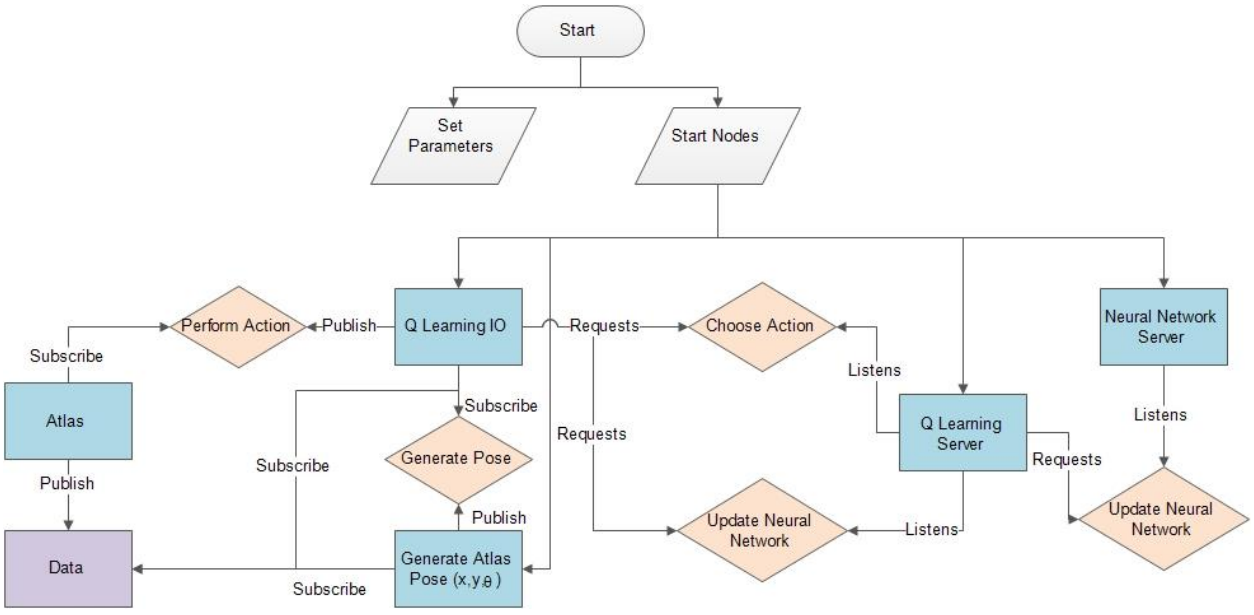

Figure 7: Flowchart of Q-learning framework

3.3.1: Atlas I/O Layer

The I/O Layer serves to act as our interface between the Q-Learning algorithm and the robot. It handles state calculation and sending the selected actions. This layer makes use of 5 classes, and runs an overall command loop for the framework, as depicted in Figure 8. The classes used are described below.

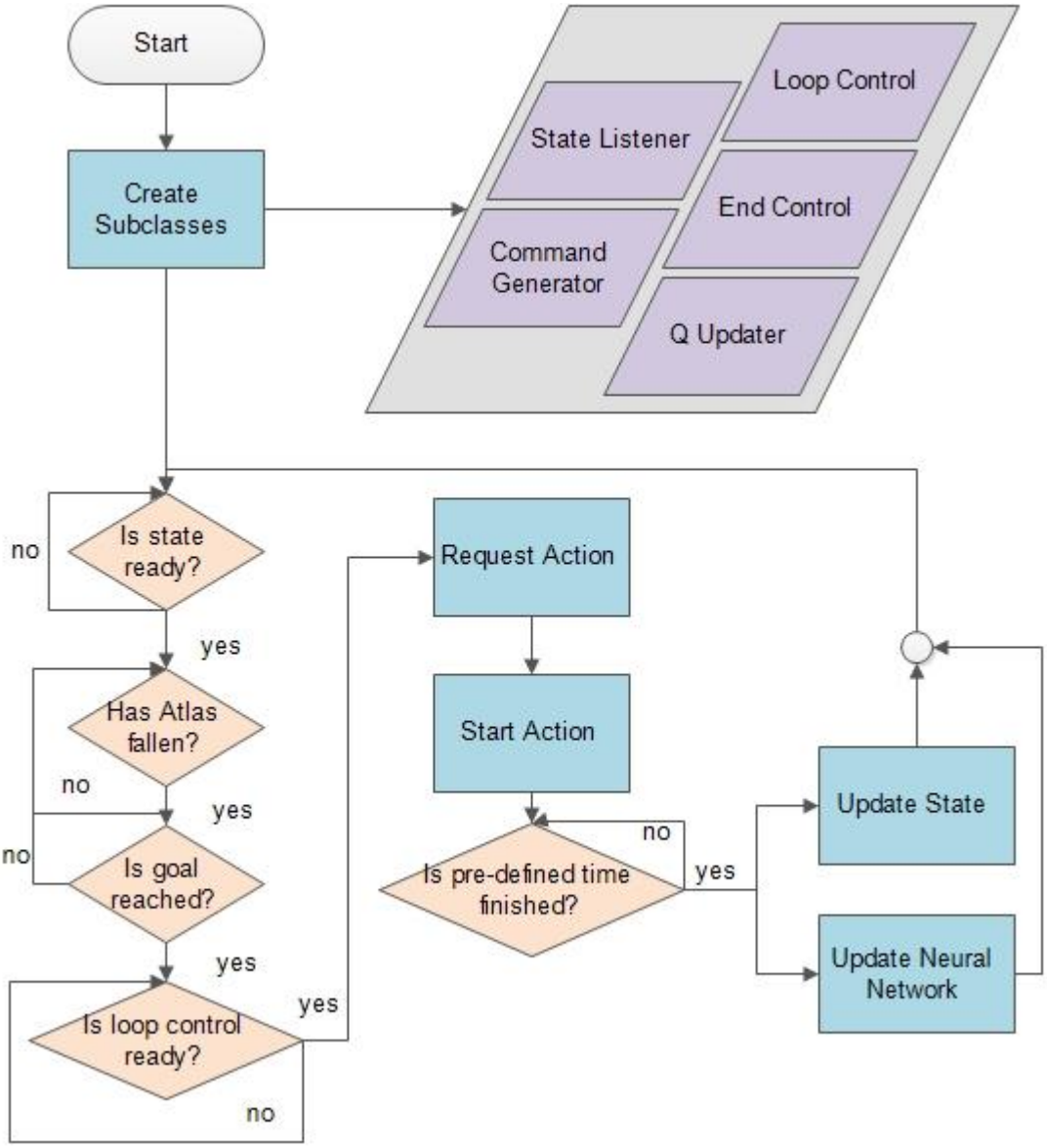

Figure 8: IO Server Flowchart

3.3.1.1: Action Requester

The action requester handles issuing an action request to the Q-learning layer, and receiving the response. The request contains the current state of the robot, and the response contains the action to be performed by the robot.

3.3.1.2: Command Generator

The command generator converts the selected action to a ROS message that can be handled by the robot.

3.3.1.3: End Control

The end control interface handles the programs end conditions, such as the robot reaching the goal, or falling over. It also ensures that a final update to the neural network is made before the program exits.

3.3.1.4: Loop Control

The loop control handles when the main logic loop of the I/O Layer runs. This could either act like a timer interrupt flag, or handle various conditions ranging from internal state information (such as an action completing) to external environmental information.

3.3.1.5: Q-Updater

The Q-Updater interface issues updates to the Q-Learning Layer, including information about the current and previous state, as well as the action performed.

3.3.1.6: State Listener

The state listener handles listening to all necessary ROS topics to gather state information, as well as performing any computations necessary to generate more abstract state information. To enable this, it also requires a function to update the listener based on the previous action performed.

3.3.2: Q-Learning Layer

From a more abstract view, the learning algorithm is composed of two main parts: sending an action to the node interfacing with atlas and updating the neural network. When a state is passed to the learning algorithm, it chooses the most appropriate action for atlas to perform based on the data in the current state. To prevent just picking the most optimal action every time, resulting in a local maximum, a random action is chosen with a small chance. Otherwise, as long as the robot has not fallen over, the node calls the neural network to find the Q values for all actions. It then returns to the interfacing node the action with the highest Q value.

The other job of the Q-learning node is to update the neural network after the action is performed in the simulator. The node passes the current state, recent actions, the current

performed action, and the reward to the neural network to update its weights between every node.

Eight subclasses are created when instantiating the Q-learning server. These classes are depicted in Figure 9, and described below.

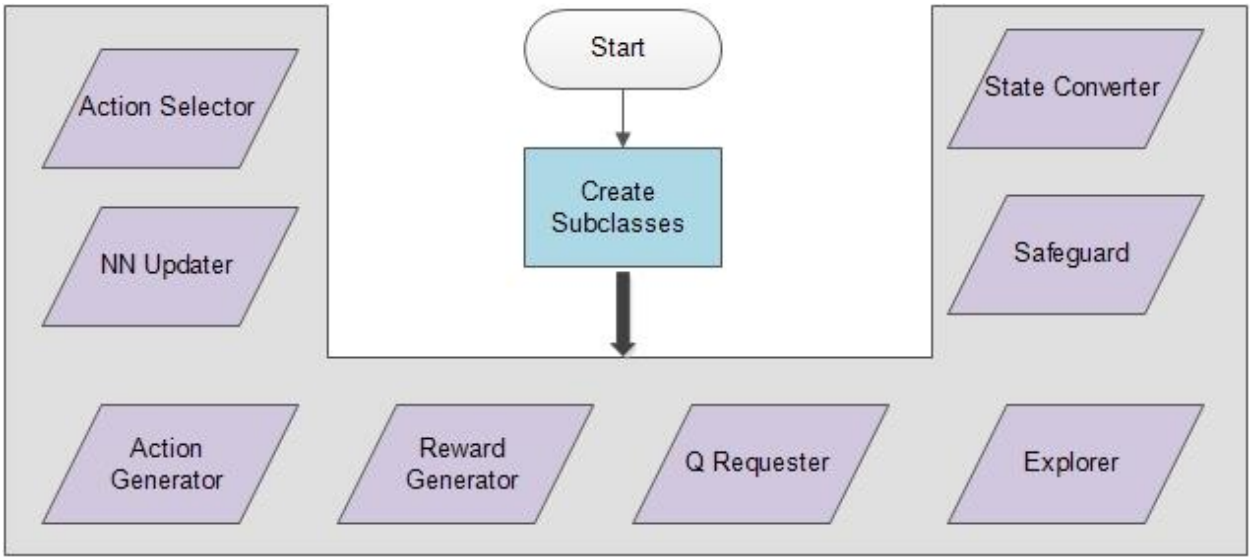

Figure 9: Components of the Q-learning Server

3.3.2.1: Action Generator

The action generator is used as a source of possible actions to select from, and is used by the action selector, explorer, q-requester, safeguard, and neural network updater to retrieve lists of possible actions.

3.3.2.2: Action Selector

The action selector receives action requests from an action requester, and handles the requests by selecting the next action to perform. It utilizes a Q-requester, an explorer, and a safeguard to accomplish this. The selected action is returned to the action requester in the I/O Layer.

3.3.2.3: Explorer

The explorer is used to override the neural networks selected actions and prevent the learning algorithm from getting stuck in a local maximum by exploring other, untried actions. The methods of exploration are dependent on the specific implementation.

3.3.2.4: Neural Network Updater

The neural network updater handles everything necessary to issuing updates to the neural network. After receiving an update request from the Q-updater interface, the neural network updater packages together the received states and action, alongside the list of possible future actions generated by an action generator, and a reward computed by a reward generator. It then sends this package to the Neural Network Layer.

3.3.2.5: Q-Requester

The Q-requester is used by the action selector to evaluate the possible actions that could be performed, and issue requests to the neural network to compute each action's Q-value.

3.3.2.6: Reward Generator

The reward generator is used by the neural network updater to compute the actual benefit of the action performed.

3.3.2.7: Safeguard

A safeguard is used to override actions that are known to be dangerous, such as stepping twice in the row with the same leg. It is intended to be used when running the framework on a physical robot, to prevent falls that could lead to damage to the robot, while still notifying the operator that either the neural network or exploration selected a dangerous action. It is not intended to be used in simulation.

3.3.2.8: State Converter

The state converter translates the various state information messages used by ROS to a vector of single-precision floating point numbers usable by a Theano neural network.

3.3.3: Neural Network Layer

The neural network will serve two purposes – estimating the values of an action from a given state, and updating the network based on the action performed and the reward calculated. This makes it integral to the learning process.

3.3.3.1: Overall Service

The main functional component is a ROS service, meaning it will handle service requests from other ROS nodes. The service will respond to two types of service calls – Q requests and update requests. Q requests take the state of the robot and the action that is being estimated, and

returns the Q value, which is the estimate of the utility of that action given the state. The update request will take in the reward calculated for the action performed, it's previous state, the aforementioned action, as well as the current state and all possible actions from the state, and update the network. Both of those are handled in the neural network class.

3.3.3.2: Neural Network

This class is where Theano is used to implement the two actions – getting a Q estimate, and the update process. For the former, the process is relatively simple – a vector consisting of the elements of the state and action is passed into the neural network, and a one-element vector is output, consisting of the estimated utility of the action.

The latter is a bit more complex. It requires calculating a cost based on the action performed and the possible future actions, and propagating the gradient of the cost across the network to update the weights and biases. The first step of this is calculating the cost, using Equation 5.

Equation 5: Cost equation

$$cost = loss + L_1 \sum \theta_w + L_2 \sum {\theta_w}^2$$

In this equation, the cost is the sum of the loss function, along with two additional parameters, L_1 and L_2. They are regulation values, that allow the gradient descent on the network to vary in speed depending on the contents of the network – L_1 regulates the sum of the weights, and L_2 regulates the sum of the squares of the weights, where in this case, θ_w are the weights (the multiplicative values in the network). The loss function itself is shown in Equation 6, and is recognizable from the loss function used in "Playing Atari with Deep Reinforcement Learning."

Equation 6: Loss equation

$$loss = (r + \gamma * maxQ({s'}_i, {a'}_i, \theta) - Q({s'}_{i-1}, {a'}_{i-1}, \theta))^2$$

In this equation, r is the reward for state and action $i-1$, $maxQ$ is the maximum Q value for all possible actions at the next state, and Q is the estimated utility for the previous state and action performed. The value γ is a multiplier to account for uncertainty inherent in

estimating future actions, and is generally set at 0.5. However, for terminal states where the robot has either reached the goal or fallen, γ is set to 0 to account for the fact that there are no possible future actions from those states. This also prevents the

During the gradient descent step, the derivative of the cost is taken in terms of each weight (θ_w) and bias (θ_b) in the neural network, and they are updated by Equation 7.

Equation 7: Update by gradient descent

$$\theta_{w,b_i} = \theta_{w,b_{i-1}} - \delta * \frac{d\ cost}{d\theta_{w,b}}$$

This defines that the updated value of $\theta_{w,b}$, at each step i of the learning process, where $\theta_{w,b}$ is each weight and bias in the neural network. The gradient is multiplied by a learning rate, δ, which limits the speed at which the values can be changed. This reduces the impact of outlier results.

This implementation is flexible enough that it can support creation of any layered neural network of any size, provided that each layer on links to the preceding and following layers, and every node in a layer links to every node in preceding in following layers. This means that the network is not suited for some other deep network designs, such as convolutional neural networks [6].

3.4: High-Level Design

Our high-level implementation was designed as a trial for the q-learning framework, to both ensure that the learning system functioned and test the training automation. The framework fits most closely to that presented above, and all events occur synchronously – the robot must wait for an action to finish to continue on, and it must wait for the neural network update to complete to select a new action.

3.4.1: Action Representation

The high-level actions consist of six actions created by the existing number-of-steps path planner, which generates a static walk path consisting of the requested number of steps, plus one additional step to bring the robot's feet back together. We are able to generate a single step action be requested a 1-step path, and deleting the extra step to a single motion. This enables us

to have negative sequences of actions, such as stepping twice in a row with the same leg, that would not be possible if we didn't isolate a single step. These actions are represented by an integer ID, which is used to identify the action to the command generator and neural network.

The first two actions are variants of stepping forward with the left and right leg, creating by deleting the second step of the two-step sequence generated when a single step is requested. This leaves the initial step out of the path.

The next two actions are the other half of the first two actions – stepping with the back left forward to bring the feet together, created by removing the first step of the plan. These steps have slightly different behavior in how the weight is shifted, but otherwise function similarly to the first two actions.

The final two actions are an entire two-step sequence, but configured to turn left and right respectively, by approximately ten degrees.

3.4.2: State Representation

The low-level state is similarly abstracted. The state representation consists of the distance to the goal, the angle the robot is facing away from the goal, whether the robot has fallen, and a simple representation of the foot position. This is made up of two values, one for each foot, where the value is zero if the foot is behind the other foot, and 1 if it is in front. When both feet are in line with each other, both values are zero. The state also includes the last three actions performed, as the robot falling can occur due to sequences of up to three actions. Including the past three actions allows the network to identify the correlation of those sequences of actions with falling.

3.4.3: Exploration

Exploration in the high-level implementation utilizes two methods – epsilon-greedy and simulated annealing.

Epsilon-greedy (or ε-greedy) is used for near-random exploration. A random value between zero and one is generated, and if it is less than some value ε, then a random action is selected instead of the action selected by the neural network.

Simulated annealing is an optimization process based on annealing in metals [10]. The process selects an option, and then predicts a future state based on the option. A random new option is then selected, and another future state is predicted. An energy level is computed for

each of these states, and compared. In this implementation, the energy is computed by Equation 8 if the new action selected does not cause the robot to use a dangerous action.

Equation 8: Simulated annealing energy

$$E = \begin{cases} \frac{distance\ to\ goal}{initial\ distance} + \frac{|angle\ to\ goal|}{\pi} * 2 & if\ not\ dangerous \\ 3 & otherwise \end{cases}$$

If the new action is dangerous, for example, if the last action performed was to step out with the left leg, and the new action is to do the same, then the energy is the maximum possible value of the energy equation – three.

Then, with some probability, the new option is selected. At the start of the process, the 'temperature' is high, and selecting a new option is very likely. As the process cools, a larger difference in energy between the two possible future states is required, unless the new state has a lower energy than the predicted state for the previously selected action. This process can be run for whatever number of iterations is desired, and allows the robot to vary between taking a 'perfect' path, directly toward the goal when using a high number of iterations, and exploring other paths by using a low number of iterations.

Equation 9: Probability of selecting new action

$$P = \begin{cases} 1.0 & if\ E(new\ state) < E(old\ state) \\ e^{\frac{-(E(new\ state) - E(old\ state))}{temperature}} & otherwise \end{cases}$$

3.4.4: Issuing Commands

Commands are generated using the number-of-steps generator, using the ID of the selected action to determine the parameters to pass to the step generator, and which step to delete. A new action command is generated and sent once the previous action is completed.

3.4.5: Reward Calculation

To update the neural network, a reward must be calculated, so that the neural network's weights can be adjusted so the output is closer to the reward. In our high-level approach, we use a tiered system. If the robot is not facing toward the goal, the change in the angle toward the

goal is used as the reward, with a decrease in the magnitude of the angle being better. If the angle toward the goal is small enough, the change in the distance toward the goal is used. This prioritizes the robot turning toward the goal, allowing the robot to take actions to turn toward the goal, even if they require the distance to the goal to increase. This is depicted in Figure 10**Error! Reference source not found.**.

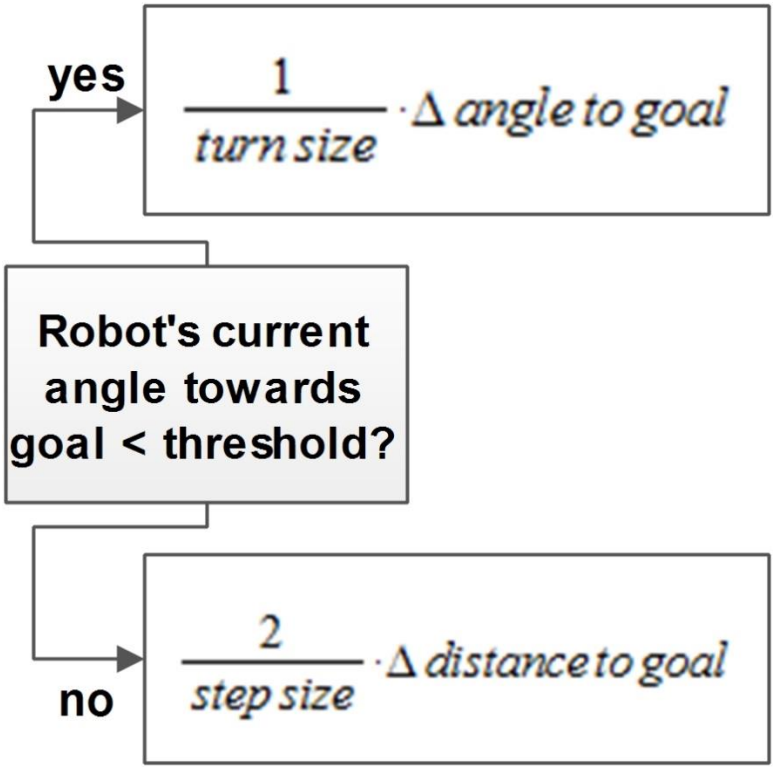

Figure 10: High-level Reward

3.5: Mixed-Level Design

The mixed-level phase did not change much from the high-level implementation. The only change that occurred was how the state was represented.

3.5.1: State Representation

The mixed-level implementation utilizes a low-level representation of the state. The distance to the goal, the angle away from the goal the robot is facing, whether or not the robot has fallen, and the previous three actions performed are carried over from the high level state. The position of the feet is replaced with less abstract information – the position, velocity and acceleration and motor torque of each Atlas's 30 joints. Additionally, the center of mass and

center of pressure are used, alongside the size coordinates making up the robot's support polygon. The absolute position and velocity of the both feet are also used, along with values representing the contact state – whether or not each foot is on the ground.

3.6: Low-Level Design

The low-level phase brought the most radical changes. Action representations are changed to handle direct joint commands, and the exploration methods have been modified, as the more complex state predictions brought about by this make simulated annealing less desirable. Additionally, our method of issuing commands and reward calculation has been updated accordingly.

3.6.1: Action Representation

The largest challenge for choosing a method of action representation was the size of the action space. Our initial designs called for using discrete joint angles for each action, but this would have led to 10^{15} actions if we only used 10 discrete angles for each of the 15 necessary joints. This was far too large. Instead, we opted to treat actions as specifying a change in joint angle by a fixed value. This allowed us to collapse the action space down to 3^{15} actions – with each joint having the option of either increasing, decreasing, or maintaining the angle.

This also allowed us to encode the action into a 32-bit unsigned integer. Each of the 15 joints is allocated two bits, and two bits are unallocated. A value of A value of 0b00 for a bit pair signals to keep the joint angle the same, a value of 0b01 indicates the angle should be increased by 5% of the joint's range of motion, and a value of 0b10 corresponds to a decrease in angle of 5% of the joint's range of motion. When passed to the neural network, these values are converted to 0, 1, and -1 respectively.

3.6.2: State Representation

The different action structure necessitated a change to the number of past actions stored in the state. The amount is changed from the three used by the high-level and mixed-level implementation to 10. Based on the 2Hz command rate, this gives the neural network information about the past five seconds of runtime.

3.6.3: Action Selection

Another difficulty brought about by the size of the action space was action selection. We cannot feasibly evaluate the benefit of performing all 3^{15} actions. That would take slightly less

than two hours using the computer we have available. Instead, we iteratively construct an action one joint at a time. For each joint, we evaluate the three possible actions – increase, decrease, or maintain. We then select the highest rated action, and continue on with the next joint, adding onto the final action from the first joint. This allows us to select a decent action with only 45 values being passed through the neural network, instead of over 14 million. To fully represent all the paths possible to reach each of the possible actions, we use a randomly generated joint order. An example of the process is depicted below in Figure 11.

$$Joint\ 0\text{: }00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00 \rightarrow \begin{matrix} 01 & \rightarrow & q = 0.2 \\ 00 & \rightarrow & q = -0.1 \\ 10 & \rightarrow & q = 0.3 \end{matrix}$$

$$Joint\ 1\text{: }00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 00\ 10 \rightarrow \begin{matrix} 01\ 10 & \rightarrow & q = 0.25 \\ 00\ 10 & \rightarrow & q = -0.05 \\ 10\ 10 & \rightarrow & q = -0.15 \end{matrix}$$

$$\vdots$$

$$Final\ action\text{: }10\ 00\ 01\ 01\ 00\ 00\ 10\ 10\ 01\ 10\ 00\ 00\ 10\ 01\ 10$$

Figure 11: Example of iterative construction of low level action

3.6.4: Exploration

Like the high-level and mixed-level implementations, we have two methods of exploration available. The first is epsilon-greedy, which is described in Chapter 3.4.3. The second is shaping.

While epsilon-greedy works, it faces a major issue – the size of the action space. There are over 14 million possible actions that can be performed, and only a handful that will be successful in a given state. While the probability of discovery of this successful actions approaches 1 given sufficient time, time is not something we had an abundance of. Using an epsilon of 0.75, this method is capable of exploring approximately 2% of the possible actions per week. That is not sufficient to learn sufficiently fast.

Shaping is a method where we feed the simulation a set of pre-generated actions. This can allow us to selectively explore known successful actions to train and direct the neural network toward a path that works. Additionally, small mutations of the shaping actions can be

performed to explore while still performing actions that are close to those that work. This presents the opportunity for optimization of the shaping actions.

The shaping actions are recorded by running the number-of-steps planner on the physical robot, and recording the joint states the robot passes through. These are then converted to a series of actions.

3.6.5: Issuing Commands

Commands are issued to set specific joint states, computed by applying the changes specified by the action to the current joint state. The timing of the commands is what differs from the high and mixed-level implementations. For this implementation, we do not want to allow the robot to pause between actions, as this delay could prove dangerous if the robot finishes an action in an unstable state that the next action could recover from. To prevent this, actions are published at a constant 2Hz rate, with each new action overriding the last.

3.6.6: Reward Calculation

The reward is calculated by incorporating several factors. The first value checked is whether the robot has fallen. If so, the reward is set to a minimum value, the negative of the maximum possible reward. This ensures a large difference between actions that lead to a fall and those that don't.

The highest weighted portion of the reward is time-based, with a constantly decreasing value as time passes, to encourage the robot to choose a sequence of actions that brings it to the goal faster. This is determined by subtracting an amount from an initial reward, with values set to be consistent with the fixed 10-minute length of training, as shown in Equation 10.

Equation 10: Time reward calculation

$$time\ reward = 60.0 + (-0.1 * time\ in\ seconds)$$

The next two components are carried over from the low-level and mixed-level reward calculation – the distance and angle to the goal. These are determined by Equation 11 and Equation 12.

Equation 11: Distance reward calculation

$$distance\ reward = \frac{initial\ distance - distance\ to\ goal}{initial\ distance} * 15$$

Equation 12: Angle reward calculation

$$angle\ reward = \frac{2\pi - angle\ to\ goal}{2\pi} * 15$$

Following is a component to ensure gradual and smooth movement, by rewarding the robot for not moving joints in alternating directions without a command to maintain the current joint angle in between. This is described in Equation 13, where N is the number of joints that switch from an increase command to a decrease command, or vice versa.

Equation 13: Gradual reward computation

$$gradual\ reward = 15 - N$$

Next is a component based on whether the robot is statically stable. Unlike the earlier phases, we do not want an extremely low reward when the robot is not statically stable, as that would preclude the possibility of the robot learning how to dynamically walk. Instead, we only want to slightly discourage instability, so while it will try to remain statically stable, a faster route that utilizes dynamic walking is possible. Stability is calculated using the support polygon, the location of the center of mass, and W. Randolph Franklin's PNPoly algorithm, which allows for fast computation of whether a point is inside of a polygon [11]. The reward is determined by Equation 14.

Equation 14: Stability reward computation

$$stability\ reward = \begin{cases} 15 & if\ stable \\ 0 & otherwise \end{cases}$$

The final two components are to discourage actions that could damage the robot, either by a limb trying to move beyond its bounds, or a joint exerting more force than it is capable of,

potentially damaging the hydraulics, motor, or structure. These are computed using Equation 15 and Equation 16.

Equation 15: Joint angle reward calculation

$$joint\ angle\ reward = 15 - \sum_{i=0}^{14} \begin{cases} \frac{lower\ threshold_i - angle_i}{threshold\ size_i} & if\ angle_i < lower\ threshold_i \\ \frac{angle_i - upper\ threshold_i}{threshold\ size_i} & if\ angle_i > upper\ threshold_i \\ 0 & otherwise \end{cases}$$

Equation 16: Joint torque reward calculation

$$joint\ torque\ reward = 15 - \sum_{i=0}^{14} \begin{cases} \frac{lower\ threshold_i - torque_i}{threshold\ size_i} & if\ torque_i < lower\ threshold_i \\ \frac{torque_i - upper\ threshold_i}{threshold\ size_i} & if\ torque_i > upper\ threshold_i \\ 0 & otherwise \end{cases}$$

Each of these rewards is then summed, which produces the weighting specified in Figure 10. The final value is then scaled so that the maximum and minimum possible rewards are 0.25 and -0.25 respectively. This prevents the cost equation from increasing to a sufficiently high value where it would exceed the bounds of a single-precision floating point number. If that occurs, all values in the neural network end up being converted to the python NaN value, causing the network become useless.

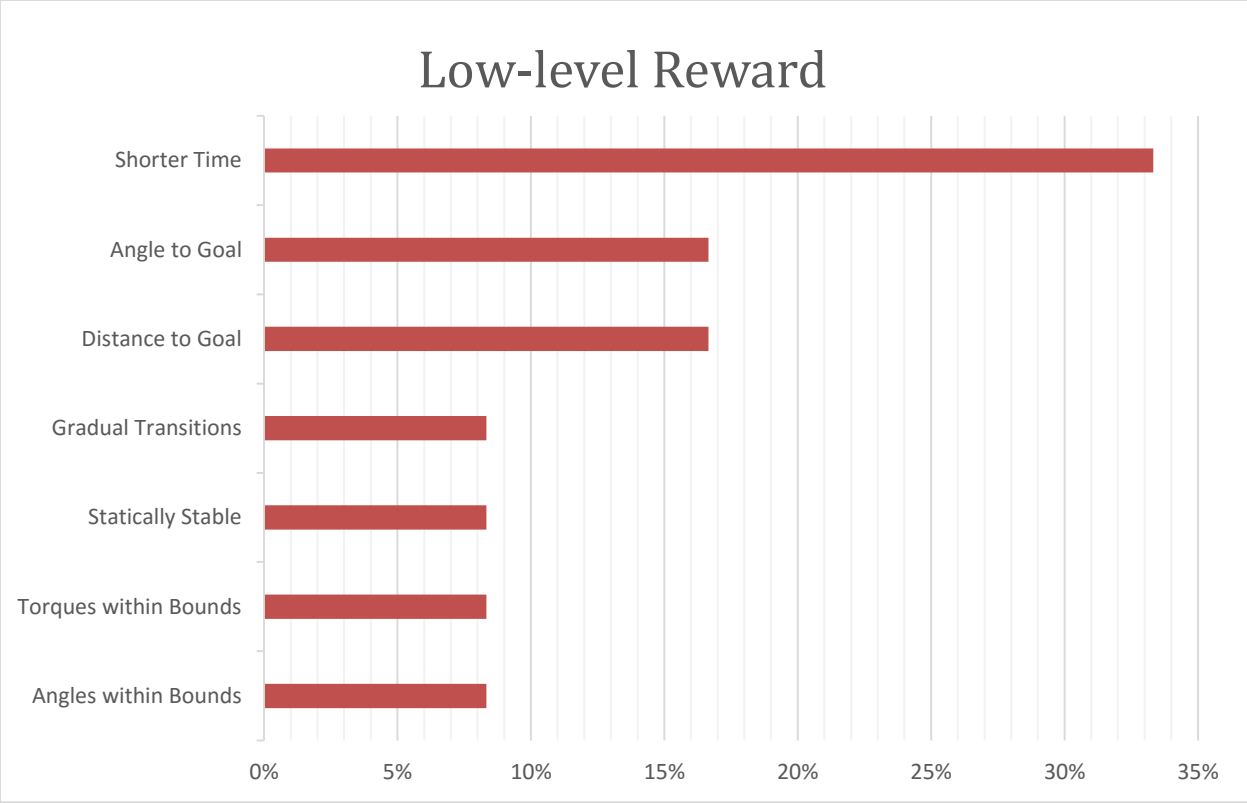

Figure 12: Composition of reward for low-level training

3.6.7: Parallelization of Neural Network Updates

The final change to allow repeated action publications was the separation of neural network updates into a separate process. This allows reward calculation and updates to occur simultaneously with selection and execution of actions. However, there is a downside to this. Because the neural network is updated in a separate process, we cannot have the updated network made available to the simulation iteration that updated it. Instead, it continues using the original, non-updated network for the duration of simulation. When the next cycle of simulation happens, the updated network is loaded from file. However, this is a non-issue, as badly performing simulation cycles will end quickly due to the robot falling over, so the updated network is available quickly. When the simulation performs well, the updated network is not needed immediately, so it is not an issue that it is not immediately available.

Chapter 4: Results

This project was successful, though not all of our goals were accomplished due to time restraints.

4.1: Framework

We successfully implemented and tested our framework with both our low-level and high-level implementations. Both of those phases proved to have different enough requirements to demonstrate the flexibility of the framework's application to different q-learning problems. While we did not test the framework with the mixed-level implementation due to time restraints, we are confident that it would be able to meet the challenges posed by that phase of the project.

4.2: High-Level Implementation

Our high-level implementation proved to be a complete success. We trained the neural network in simulation for two weeks by assigning the robot random goals on a 20-meter square grid centered around the robot's start position. After the two weeks of training, the neural network has been updated 50,000 times, so we proceeded to test the network on the physical robot. For the physical tests, we enabled the 'safe mode' flag, which causes the robot to select the next best action if a dangerous action that is known to cause the robot to fall is selected. This served to protect the robot, but also alert us of any negative choices made by the artificial intelligence. The results of our 5 trials on the physical robot are listed below in Figure 13.

Trial	Reached Goal	Number of Dangerous Actions Attempted
1	Yes	1
2	Yes	0
3	Yes	0
4	Yes	0
5	Yes	0

Figure 13: High-level physical test results

The only failure that occurred during the testing was on the second action performed on the first trial run. The neural network update following that action prevented this from occurring in the future, and all subsequents tests proceeded successfully and without incident.

4.3: Mixed-Implementation

The mixed-level implementation was less successful. While the robot quickly learned to turn toward the goal during simulation, it faced difficulties picking up on which foot was currently forward, and would therefore run into issues when it reached the point where it should walk toward the goal. We halted simulation after approximately 175,000 updates to the neural network, as the low-level implementation was ready to train, and we wished to maximize the amount of time the low-level implementation had to train.

4.4: Low-Level Implementation

This phase again faced difficulties with time constraints. Our initial efforts to use only epsilon-greedy exploration was unsuccessful because of this. One week of training was only sufficient to cover 2% of the action space, and did not lead to any of the small handful of actions that would not lead to the robot immediately falling over.

Shaped actions were implemented to assist with this. Our initial set of shaped actions were recorded from the number-of-steps step generator in the simulator. This faced several issues, as the simulator introduces exaggerated hip oscillations in multi-step sequences. These were unfortunately captured by the recording method. Despite the oscillations, shaping quickly allowed the robot to progress beyond immediately falling over, as the network was introduced to that small handful of actions. This can be seen in the small increase in the number of actions performed before falling in Figure 14.

To further improve the shaping, and avoid the hip oscillation issue present in the simulator, we recorded actions from the number-of-steps generator on the physical robot. This brought a further improvement to the learning process, allowing the robot to fully shift the weight to its left leg in simulation. This can be seen in the large increase in the number of actions performed before falling in Figure 14.

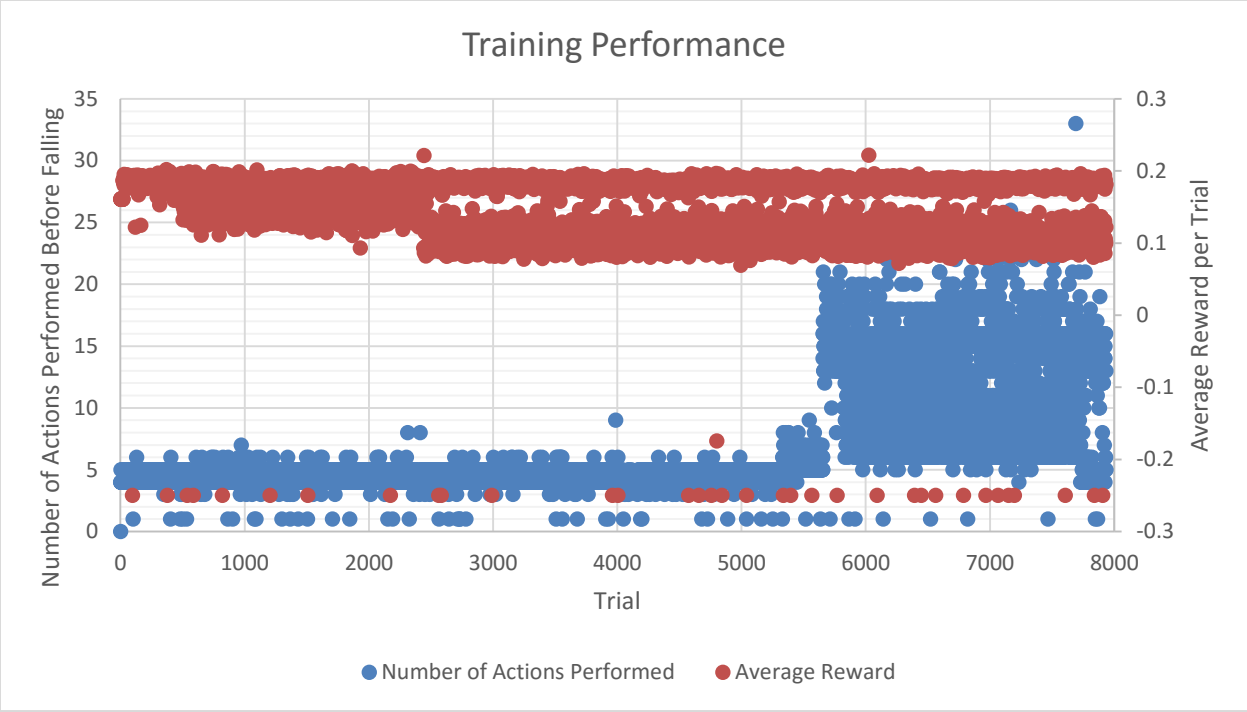

Figure 14: Low-level training performance

In addition to the changes in the number of actions before falling in Figure 14, two significant changes can be seen in the average reward per trial, at approximately trial 500 and trial 2500. These were changes to the reward calculation, that better differentiated between different benefits for actions.

However, despite the large increase brought on by shaping, it was insufficient to allow Atlas to learn how to walk completely using direct joint control. After switching to the physical recordings, the robot was able to fully shift its weight to the left leg, but then fell once it attempted to precede further. This is likely due to the action representation's fixed angle change percentage. The constant change of 5% is likely insufficient to represent the state changes the step generator brings the robot through with complete accuracy. These inaccuracies would then compound, leading the robot to fall.

Despite this failure, the framework proved itself, as did the learning capabilities of the neural network. Shaping took effect extremely quickly, and the learned actions were repeated when exploration was disabled. Even though it failed to move on beyond shifting its weight, it is likely that sufficient time spent training would allow the robot to learn how to walk completely.

Chapter 5: Future Work

The framework established in this project presents several areas for possible future work, both in the realm of walking, and in other areas amenable to the Q-learning algorithm.

Within the realm of walking, one potential project is the modification of the existing low-level framework to better take advantage of shaping. Using shaping as the primary exploration method greatly alleviates the issues prevented by an extremely large action space, as only a relatively small proportion have to be explored. Changing the action representation to handle distinct joint angles, or alternatively, continuous angles, would be rewarding path of research, as they could better represent the joint angles the existing step generators bring the robot through.

Another potential path of research would be to expand the sensors used by the Q-learning algorithm to incorporate other data, such as feedback from the robot's visual system and LIDAR, or the knowledge of ground height. These could enable the learning algorithm to be applied to path planning, or enable walking over more complex terrain than flat ground.

Other methods of interacting with the world than walking also present opportunities. Our framework is design to be used with most, if not all, tasks involving Atlas for which Q-learning is applicable. It could be used for learning manipulation and object recognition, especially if sensor data from the robot's stereoscopic camera and LIDAR are incorporated.

A final path of research, that is applicable to improving training time for any Q-learning task, would be to explore the idea of using distributed computing for training, by running many simultaneous simulations using either multiple physical machines, or by utilizing cloud-based virtual machines such as Amazon Web Services.

Chapter 6: Conclusion

Our goal was to have a humanoid robot learn how to walk, and with our high-level implementation, Atlas successfully learned the proper sequence of actions to walk. Our low-level implementation is working in simulation but needs significantly more training time to learn the appropriate action sequences. It also needs additional modification to run on the physical robot, as the topic used to publish commands from the field compute to the robot. However, both of these tasks proved the strength of the Q-learning framework we developed, opening many paths for future research. These can range from expanding the flexibility of the low-level walking method, to incorporating other sensory information to learn path planning, or even applying the framework to other tasks such as manipulation.

References

[1] "WPI-CMU DARPA Robotics Challenge Team," Worcester Polytechnic Institute, 2015. [Online]. Available: http://drc.wpi.edu. [Accessed 16 December 2015].

[2] "DARPA Robotics Challenge," Defense Advanced Research Projects Agency, 2015. [Online]. Available: http://www.theroboticschallenge.org/. [Accessed 16 December 2015].

[3] Boston Dynamics, "Boston Dynamics Atlas Robot Software and Control Manual," 2013.

[4] "About ROS," Open Source Robotics Foundation, 2015. [Online]. Available: http://www.ros.org/about-ros/. [Accessed 15 December 2015].

[5] J. McCullock, "A Painless Q-Learning Tutorial," Mnemosyne_Studio, 2012. [Online]. Available: http://mnemstudio.org/path-finding-q-learning-tutorial.htm. [Accessed 29 October 2015].

[6] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends in Machine Learning,* vol. 2, no. 1, pp. 1-127, 2009.

[7] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhouke, P. Nguyen, T. N. Sainath and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Sginal Processing Magazine,* pp. 82-97, November 2012.

[8] M. Forouzanfar, H. R. Dajani, V. Z. Groza, M. Bolic and S. Rajan, "Comparison of Feed-Forward Neural Network training algorithms for oscillometric blood pressure estimation," in *4th International Workshop on Soft Computing Applications*, 2010.

[9] "Theano at a Glance," Deep Learning, 15 December 2015. [Online]. Available: http://deeplearning.net/software/theano/introduction.html. [Accessed 15 December 2015].

[10] S. Kirkpatrick, C. D. Gelatt Jr. and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, no. 4598, pp. 671-680, 1983.

[11] W. R. Franklin, "PNPoly - Point Inclusion in Polygon Test," 21 January 2014. [Online]. Available: https://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html. [Accessed March 2016].

[12] "The Human Balance System | Vestibular Disorders Association," [Online]. Available: http://vestibular.org/understanding-vestibular-disorder/human-balance-system. [Accessed 11 October 2015].

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing Atari With Deep Reinforcement Learning," in *NIPS Deep Learning Workshop*, 2013.

Appendices

Appendix A

Pseudocode for Deep Q-Learning as implemented in Playing Atari with Deep Reinforcement Learning

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for