Two Approaches for Learning Humanoid Walking

Andrew Hulsizer ANDREW.HULSIZER@OU.EDU
Kim Houck KIM.HOUCK@OU.EDU
Peter Reid PETERREID@OU.EDU

Abstract

The task of controlling a robot to walk like a humanoid is a long-standing topic of research. We describe two approaches to this problem that we have explored. The first uses a tree to classify a skeletons position as a member of a state set then runs Q-learning on those states. The second uses a neural net to control each of the joints. We compare the efficacy of these approaches to a fixed, hand-made policy and a random actor. These comparisons are done in simulation, using an open source physics simulation engine.

1. Learning Task

1.1 Goal

Our goal is to train a simulated humanoid robot to move forward efficiently, given an environment that simulates common forces (e.g. gravity). Ideally, the method of moving that it settles on will mimic how humans walk in real life.

1.2 Criteria

Our project attempts to move a simulated robot throughout a simulated world. At the highest level our project aims to move the humanoid robot as far forward with the fastest velocity. On a deeper level, we have aimed to use good machine learning techniques in an efficient manner to decrease the vast number of states we have as well as have the robot learn the best action given a particular state.

The success or failure of our implementation can be judged by charting the robots speed as it tries to make its way down our simulated field. If, over time, that speed increases, it can safely be said that learning is taking place. We can also compare the robots speed with the speed of other implementations, such as a fixed, hand-made policy or a random actor. All simulation and measurements will take place in a world constructed using the Bullet physics engine [1], which is widely used in game and movie production.

1.3 Problem Motiviation

The problem of making machines walk like a human has been an area of research for decades. This is an inherently useful problem to solve, since a machine that can move similarly to a human can better operate in human-made environments and rough terrains than a typical wheeled robot. A variety of approaches have been tried, but the problem remains fundamentally difficult.

2. Related Work

One similar and thoroughly studied problem domain is that of the acrobot (Yoshimoto, 1999). Like the skeleton in our problem, the acrobot controls continuous joints in order to execute some pattern of movement. Also similar to our problem is the fact that the acrobot is underactuated; it has more degrees of freedom than it has direct control over. In our case, those extra degrees of freedom are in the form of the whole skeleton tipping back and forth and moving around.

There is also a large body of research that directly tackles humanoid problem. Discussions of design elements required (such as function approximators) and the issues faced (such as balance) in these papers has informed our own plans (Peters, 2003). Some existing approaches, like ours, use neural nets as part of their system (Palmer, 2009).

3. Partition Tree and Q-Learning Approach

3.1 Motivation

Q-learning provides a simple way to learn an optimal policy for any domain that has the Markov property.

[1] The Bullet Physics Engine, widely used in the production of games and movies, is available at www.bulletphysics.com.

In our domain, a state representation that includes the positions and velocities of all the joints of the skeleton is perfectly Markov. Just like all future movement of a ball in an idealized world can be determined given its current position and velocity, a skeleton could predict how it is going to move given its current configuration; this would simply be a matter of running a separate physics simulation on the side.

Unfortunately, this perfectly Markov state representation is not amenable to Q-learning. There are far too many states. If the agent were to perfectly predict what an actions outcome was, the state would need, at a minimum, one real value for each degree of freedom in the skeleton. For our particular skeleton, which has 17 degrees of freedom, that would be 17 32-bit numbers, leads to about which is about $10^{1}63$ states. To give a tired comparison, this is more atoms than there are in the observable universe. Clearly this is impractical.

This is where our tree comes into play. It is designed to sacrifice some of the Markov property in order to bring the number of states down to a reasonable number. Ideally, this would give us a world representation in which we can compute a close approximation of future states given the current state and in which we can apply Q-learning. The basis of this is that most of the time, small changes in joints have little effect on the outcome of actions. For example a rotation difference in the elbow has little impact on the outcome of taking a step, but where the foot was has a large effect. If our tree can distinguish between important and unimportant aspects of the skeletons configuration, then we can discard the unimportant ones and be left with a smaller set of states.

3.2 Design

The most novel aspect of this agents design is the tree. At the highest level, its function is to map the skeletons configuration (descriptions of the position, rotation, and velocity of various joints) onto a set of states. In the rest of this description, it is important to distinguish between two terms:

Configuraton The raw position, rotation, and velocity of joints in the skeleton. There are about 10163 possible configurations for our skeleton.

State The abstract summary of a configuration. There are much fewer of these and they are intended to be as Markov as possible.

Using this terminology, our Q(s,a) values use states, not configurations, as the first argument.

The structure of the tree is as follows. Each leaf node in the tree corresponds to a single state. Each interior node has two sub-trees and a binary test that can be applied to a configuration. Depending on the outcome of that test, it will categorize the configuration as either belonging to a state in the left or right sub-tree.

Each of these ”tests” is actually just a comparison of one value in the configuration with some threshold. Originally, our tests has a more general form: testing whether the values in the configuration represented a point above or below some hyperplane. However, this level of generality seemed to do more harm than good, since it made the tests harder to learn and allowed irrelevant details to impact the test result. Our current setup, of testing just one value in the configuration, is equivalent to allowing only hyperplanes that are orthogonal to some axis.

The behavior of this tree can be more clearly illustrated with an example. Suppose that the tree has determined that the rotation of the left hip is the most important aspect of the configuration to consider. This is not an unreasonable supposition, since whether the left hip is forward or backward will make an action have a dramatically difference consequence. When the tree is asked to give the state corresponding to a configuration, it will test whether the left hips rotation exceeds some threshold. If so, it will use the left sub-tree to further categorize the configuration. Otherwise, it will use the right sub-tree to further categorize it. By the time a leaf node is reached, for example, the tree might end up as categorizing the configuration as having the right leg forward with its knee straight, the left leg back with its knee somewhat bent, and the arms out to the side. All this information would be encoded, essentially, by the state that the tree ultimately returns.

Next comes the question of how to learn the tree. That is, how does it determine that the left hip is the most important to test first? Our approach is as follows. First, the agent gathers some examples of what happens, in terms of reward and expected return, when it takes an particular action in a particular class of configurations. Sometimes the action will lead to high reward and sometimes it will lead to low reward. (If this state were perfectly Markov, it would always lead to the same reward.) Once it has gathered enough examples, it can try to determine which aspect of the configuration has the most impact on that reward. This is done by considering several potential hyperplanes to branch based on. The hyperplane that best separates the configurations from which the action worked well from those from which it did not is selected as the new

test to use at that node. If all goes well, when in the future the agent has a choice to execute this action, it will be better able to predict the outcome based on this new test.

The precise definition of forming a new nodes test is this:

$$(d,t) = \arg\min_{d \in D} \arg\min_{t \in \mathbb{R}} \; \sigma(\{X.reward | X.d < t\}) + \sigma(\{X.reward | X.d \geq t\})$$

Each example X has a reward attribute and an attribute recording what the reading for each input when the reward occurred. The input dimension (d) and threshold values (t) are chosen such that the standard deviation in rewards of the partitions are minimized.

Aside from using this to approximate Q-values, we can use it to approximate an arbitrary, fixed function. Some examples of this are given in the results section.

4. Neural networks with control outputs modified by Q-learning

4.1 Motivation

Neural nets were chosen to work in the continuous state domain because, unlike trees or straight RL, they can handle continuous input and output if necessary. This makes them useful for imitating another walking algorithm, whether a hard coded walking cycle or even eventually a motion capture video of a real human walking. Motion capture has been used before on a simpler but similar problem: a robot playing air hockey where the positions of the paddles and pucks on the table were tracked visually (Bentivegna and Atkeson, 2002). The air hockey robot used nearest neighbor look-up to represent it's state space, but neural networks were choosen for the humanoid robot due to the much more complicated state space.

Q-learning reinforcement learning was chosen to improve upon the controller generated by pure imitation because the choice of action for a given state needs to depend on the long term value of that taking that action, in this case moving a joint or joints. A given movement may do little in the short term to move the robot forward, but if it results in the robot remaining upright and eventually making forward progress it may be a better choice than leaping forward quickly but then falling and not going as far in the long term. Q-learning was chosen because it is simpler than Q-lambda, however it was modified to make it somewhat more like Q-lambda in that instead of taking the reward value at the next immediate time step it uses the reward value from a single time step some time in the future. This was done due to the fact that the controllers update at over 60 Hz and therefore a single update does not give time for a given joint action to generate much of an effect on the model as a whole. Below is the update function for Q-learning:

$$\begin{aligned} Q(s_t,a_t) = \; & Q(s_t,a_t) + \\ & \alpha[r_{t+1} + \gamma \arg\max_{a \in A(s)}(Q(s_{t+1},a) - Q(s_t,a_t)] \end{aligned}$$

(Sutton and Barto, 2005)

Due to this delay the update function for Q-learning our function is modified to be this:

$$\begin{aligned} Q(s_t,a_t) = \; & Q(s_t,a_t) + \\ & \alpha[r_{t+delay} + \\ & \gamma \arg\max_{a \in A(s)}(Q(s_{t+delay},a) - Q(s_t,a_t)] \end{aligned}$$

4.2 Design

The basic design for the controller uses a separate neural net for each joint, which takes as input a normalized time step value, boolean for each foot designating whether it is in contact with something, presumably the ground, and the joint positions of the knees hips and ankles. The first layer of the network uses 9 sigmoid nodes, one for each input, the second hidden layer has 3 sigmoid nodes and the output layer has a single tanh node that outputs the change in radians for that time step. Initially the neural networks are trained off a hard-coded walking cycle algorithm. This cycle was created by trial and error however with proper image processing it could be taken from a video of an actual human walking. In this case, however, the data to train the networks comes from sending the joint commands to the robot in the simulation, ensuring accurate physics. Since the hard-coded commands do not take dynamic balance into account the model is respawned each time it falls, although each time step of valid simulation is treated as a separate instance for purposes of supervised training of the networks. Once the networks are trained to mimic the observed policy below an error threshold they are used to control the agent and Q-learning is used to decide whether to scale the output of the neural nets up or down by 20

The state representation for the Q-learning command scaling takes into account the time step in the walking cycle(grouped together into intervals of 10), whether the left, right, both or neither foot is touching the ground, and whether the head is to the left/right

and/or forward/behind the centroid of the 2 feet. The motivation behind this state representation was to attempt to introduce the agents center of gravity into the control process and therefore give it at least some ability to balance.

5. Results

5.1 Function Approximators

5.1.1 Tree as a Function Approximator

To test our tree's correctness independent of the walking application, we applied it to various functions in a two-dimensional space. The three functions tested are listed below:

- A piecewise constant function in which the function is defined as flat and constant in each of several regions. This tests its ability to handle discontinuities.

- A function that is quadratic in both x and y. This tests its ability to learn nonlinear functions.

- A trigonometric function, specifically atan2. This is both discontinuous and nonlinear.

Each of the following curves represents averages after 10 runs. The sudden drop after 20 samples is an artifact of our trees threshold for determining whether a number of points is large enough to justify forming a new branch.

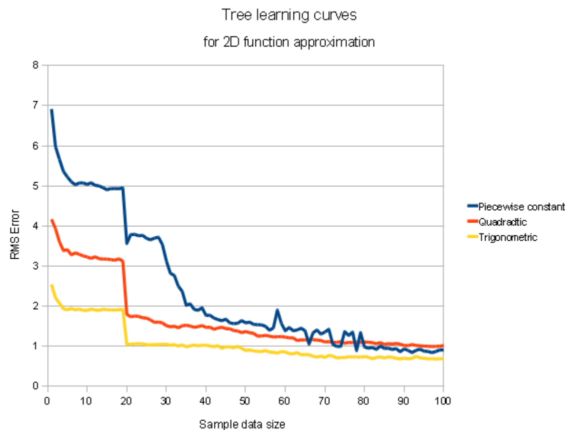

Figure 1. Learning curves for a successful run of the tree as a 2D function approximator

5.1.2 Neural Net Mimicking a Fixed Policy

Over time, the neural networks learn to mimic the commands given by the hard coded, albeit not perfectly. The learning rates are similar between the low and high friction environments and all 6 controllable joints in the model should learning, although none seem able to perfectly mimic the heuristic's commands.

The error of the neural network compared to random is not shown, although the random commands would vary over the range of motion of each joint, being approximately pi/2 for the knees and ankles and pi radians for the forward/back swing on the hips. The "error" of the walking heuristic is 0 at all times, since the heuristic's commands are the comparison from which the error is obtained.

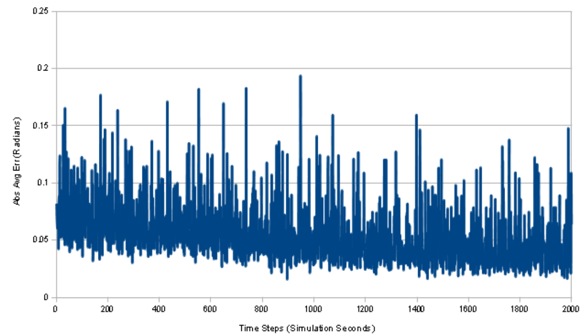

Figure 2. Average error over time for a knee joint's neural net learning to mimic the fixed policy. This is in a low-friction experiment.

5.2 Walking Results

In our first experiment, we applied various skeleton controllers to walking over a high-friction plane. The most interesting of these is our learning agent, which used a learned, tree-partitioned state set and Q-learning. Joining it were a random agent and one using hand-coded walking cycle. The neural net-powered agent is not yet ready to join these three. We recorded results by graphing forward velocity with respect to time. Time was segmented into short intervals and forward progress during each was recorded. Each of these time intervals corresponds to a point in the graphs below.

The random and fixed policies worked as expected, which helps validate the our methods of measuring agent performance. The random agents performance is low and relatively constant. The fixed-policy agents performance is marginally higher and still constant.

The tree-based approach shows a fairly linear increase in performance followed by a plateau at around 2000 time steps. A line segment approximating the increasing portion of the learning curve has a slope of around .0005 units per second per second. The plateau is at around 1.26 units per second.

For our second experiment, we made the plane slippery and ran the same set of policies.

Results: This experiment tested our agent's ability to learn to move forward on terrain that has a low friction (i.e. slippery). From our results below you can see that the friction did not play much role in all agents ability to move forward. Both the random and hand-built agent remained roughly constant over the period measured.

The tree-based agents performance can again be approximated by a line segment followed by a constant function. The plateau is marginally lower: 1.24 units per second.

The learning from observation approach using neural networks and Q-learning was not as successful, as the forward velocity drops dramatically when control is switched from the heuristic to the neural nets at 1000 time steps, and it never perceivably recovers as an attempt is made to scale the learned commands. While the raw numbers cannot be directly compared to the tree based approach due to the fact that the model is allowed to fall down and respawn, it clearly performs wore due to the fact that it is outperformed by the hard-coded heuristic. Most on the larger negative velocities in the graph are due to these respawns and the models position being reset back to the origin, although sometimes the model does manager to actually take steps backwards.

6. Analysis

Overall, the learning curve from our tree-based approach is encouraging, since it slopes up until reaching a plateau. However, there are several problems with this approach that we believe limit the height of the plateau in the learning curve.

- Knowledge leak - The repartitioning process can destroy useful information, since the old partition is thrown away. The new partition is likely to be somewhat better, but it would be better if it kept the best of the previous partition.
- Crude partition placement - Our agent places its partitions orthogonal to axes and only approximates where they belong. There are more complex methods for doing this kind of task (support vector machines) that could be implemented as future work and would likely have better results.
- Slow Q-value propagation - We used Q-learning to learn Q-values, which is the most basic of this class of methods. Some tweaks to it would accelerate how quickly Q-values converged to something correct. This slowness is particularly harmful if a repartition happens on Q-values that have not yet properly converged.
- Lack of action choice - The agent chooses from among a fixed set of actions, which limits what it can do much more than the skeletons physical limitations. A more powerful agent would be able to come up with new actions to run rather than using the relatively small set that our agent starts with. We chose not to incorporate an ability to change actions into our agent. If we had, there would be three kinds of learning going on at once: Q-values, partitions, and action sets. Three simultaneous kinds of learning seems like it would be slow and problematic.

The neural network controller based learning from observation approach is still having difficulties however, part of this is because unlike the tree approach the model was not constrained to not fall down, therefore adding to problem of balance as a requirement for the agent. Despite this however there are a few things that could possibly improve upon this approach:

- Better representing the discontinuous nature of the steps in the heuristics walking cycle in the inputs to the neural networks. The heuristic uses a 900 time step counter broken down into 100 time step wide sections of continuous trajectory. Since this time step parameter is represented to neural network as a the time step value divided by 900 it leaves little difference in the inputs on either side of the discontinuity. Despite the fact that neural networks can handle some degree of discontinuity, a different representation of this input or a different network topology might help improve the approximation of the heuristic.
- Association of the actions across all joints might help the Q-learning phase better learning to scale the controller commands. As it is the Q values are learned in a vacuum, which prevents coordinated action between the joints. Doing this however would increase the number of actions from 3 to 3^6, which is a significant increase.
- More accurate representation of the relative location of models center of gravity relative to the feet

Figure 3. Learning curves for experiment with a high-friction plane ($\mu_k = 25$)

Figure 4. Learning curves for experiment with a high-friction plane ($\mu_k = 100$)

in the Q-learning input model.

- Additional training time, data collection was done using only 2000 seconds of simulation time, only allowing for about 120,000 update cycles to learn from. More time might yield better results.

7. Conclusion

Both approaches showed learning when applied as general function approximators. Additionally, the tree-based approach showed some success when applied as part of our walking system. There is room for improvement in the placement of partitions and in the development of better actions to use. Although the system is not up to powering an actual walker, it has shown that it can at least support some degree of learning.

References

Sutton, R. S. and Barto, A. G. (2005) *Reinforcement Learning: An Introduction.* Cambridge, MA, MIT Press.

Bentivegna, D. C. and Atkeson, C. G. (2002) *Learning How to Behave from Observing Others*

Yoshimoto, J., Ishii, S., Sato, M. *Application of reinforcement learning to balancing of Acrobot.* Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 516 - 521 vol.5.

Peters, J., Vijayakumar, S., Schaal, S. *Reinforcement Learning for Humanoid Robotcs.* Third IEEE-RAS International Conference on Humanoid Robots, Karlsruhe, Germany, Sept. 29-30 2003

Palmer, E. Michael; Miller, B. Daniel;, *"An evolved neural controller for bipedal walking with dynamic balance,"* Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, 2009