On the Automated Design of Bipedal Robot Control

Trevor Standley
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
tstand@cs.ucla.edu

Abstract

The open source physics engine Box2d was coupled with several machine learning techniques and applied to the task of developing a system for controlling a two dimensional robot simulation to engage in lifelike bipedal locomotion. A two dimensional, rigid body, humanoid model was designed with body part proportions and masses as close to those of a human subject as possible. The model had 13 degrees of freedom and 14 bodies. A function mapping a vector of features of the state of the robot to a vector of appropriate joint torque controls was defined and implemented using a neural network. The parameters of the neural network were learned using an evolution algorithm which proceeded in stages from standing to walking. This resulted in unnatural looking forward locomotion. Furthermore, the parameters of the evolution algorithm had to be hand tuned for a specific set of robot parameters and configuration. In order to overcome these limitations, other machine learning techniques for implementing this function and learning the function parameters were implemented including q-learning with neural networks and q-learning with regression trees. Neither method was more successful than the evolution algorithm.

Introduction

Designing motion controllers is a well studied problem in both robotics and digital entertainment. Typical bipedal motion controllers, such as the one that controls Asimo are designed by hand, and result in motion that does not look lifelike (Chestnutt et al. 2005). Reinforcement learning techniques such as those used in Boston Dynamics' Big-Dog, and evolution algorithms such as those used by Natural Motion and in (Reil and Husbands 2002) do create lifelike motion controllers, but have not been applied or are not effective for the task of simulated bipedal locomotion in unknown or stochastic environments[1].

Much work has been done on learning to control bipedal robots both in the real world, and in three dimensional simulated environments, but there are many applications, especially in digital entertainment, for motion controllers for two dimensional robots. Although human minds have evolved in three dimensions and modern computer hardware can easily represent three dimensional environments, many (if not most) modern video games still operate in strictly two dimensional environments, and merely employ the capabilities of advanced graphical hardware for aesthetic purposes. Recent award winning games such as Sony's Little Big Planet and Nintendo's Super Smash Brothers Brawl are a few such examples. It seems that the simplicity of understanding and character control in two dimensions gives such games their appeal.

A good deal of work has also gone into finding a particular sequence of torques to apply to each joint during each time step in a model to result in locomotion. Some of this work focuses on creating signal generators which generate the appropriate waveforms for desired motion, while others aim to create an explicit representation of the waveforms in memory which can be deterministically simulated on the robot to result in locomotion. These approaches are simple, but have several drawbacks.

Since the problem of bipedal motion in two dimensions must consider balance in only one dimension, one might expect attempts to design motion controllers in two dimensions to be more successful than attempts to design motion controllers in three dimensions, and since this problem has many applications, I chose to explore this problem for my final project.

Goals

The ultimate goal for the project is to create a system capable of automatically designing a motion controller for a simulated robot given only a description of the physical characteristics of the robot and the physical laws of the simulation. Once designed, the motion controller would take a description of the state of the robot and the world, and produce the appropriate muscle torques for the robot to apply. We would like the motion controller to be robust to noise in the state description as well as nondeterminism in the state transitions. Furthermore, at any time during the simulation, it should be possible to switch to a different motion controller that was trained using a different objective function. For example, the system could design a motion controller for walking, and another for jumping. Then a user should be able to press a key to give the walking motion controller control of the robot, causing it to walk forward. At any time

[1] Recently Boston Dynamics was able to accomplish this in a real 3d environment with Petman

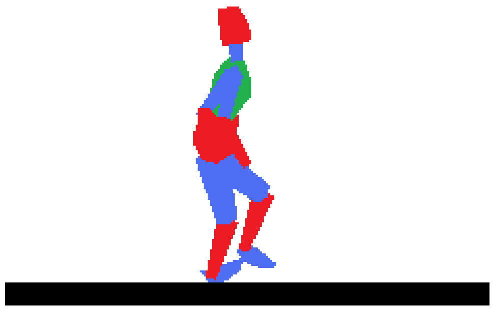

Figure 1: The bipedal robot model.

later, the user should be able to press a different key to give the jumping motion controller control, at which point the robot should seamlessly jump regardless of which state the robot is in when the switch is made.

Testbed

Although the goal is to be able to generate a motion controller for any type of robot, a bipedal robot was designed to test the limits of the methods.

A simple syntax for inputting the characteristics of the robot and the physical laws was designed, and a simple file defining a bipedal robot was coded within that syntax. This approach theoretically would allow any robot to be input into the program for the evolution algorithm to work on, but the algorithm was only tested on a single robot design.

The file defined the shapes of 14 rigid bodies: two feet, two lower legs, two upper legs, one lower torso, one upper torso, two upper arms, two lower arms, a neck and a head. The file also specified the positions of the necessary 14 revolute joints to connect the rigid bodies.

The length measurements for the rigid bodies were carefully measured on a human subject. The masses of each rigid body were found using an equation taking total body mass and height. The equation, which was available online from the Department of Anatomy of the University of Brussels[2], was created using linear regression on experimental data from cadavers for each mass. The moments of inertia were calculated as if the mass was uniformly distributed throughout the area. Other parameters, such as muscle strengths, could not be found online, and were hand tuned to give good walking performance, but kept within the realm of human ability.

States, Actions, and Functions

We consider a class of motion controllers that are stateless, and therefore only consider the current configuration of the robot when selecting actions for a particular timestep. The task of designing a motion controller can therefore be seen as a task of defining a function $f(x)$ that maps the state of the robot at any timestep x to the appropriate action (joint torques for each joint) for that timestep a.

[2]`http://riodb.ibase.aist.go.jp/dhbodydb/properties/m/table/e-k-01_m_table.html`

Another function, $p(x_t, a)$, represents the laws of physics. This function maps the state of the robot and an action to a subsequent state x_{t+1}. The trajectory of the robot is a sequence of states, $x_0, x_1, ...$ that the robot passes through until the end of the simulation.

Given the motion controller $f(x)$, a function $p(x, a)$, and an initial state x_0 we can generate the remaining states in the trajectory $x_1, x_2, ...$ by iteratively applying a function $g(x) = p(x, f(x))$ to the start state until the end of the simulation.

A more common approach, such as the one in (Erez 2010), defines a motion controller as a function $h(t)$, which maps a timestep t to the appropriate action. Under this approach, one generates a state x_t in the trajectory from a state x_{t-1} by applying the function $j(x_{t-1}, t) = p(x_t, h(t))$ to the state. In this case, the function j requires both the timestep and the previous state to generate the next state.

We will define, for both approaches, a fitness function $q(x_1, x_2, ...)$ which takes a trajectory and returns a value. Larger values correspond to more desirable trajectories.

Our approach has several advantages over the more common latter approach. Firstly, if we allow for nondeterministic physical laws, then the latter approach must find a sequence of actions that bring the start state through a trajectory with high fitness regardless of which random choices are made by the physical laws. This may not be possible. Our approach, however, can learn in the presence of nondeterminism, and must only choose an action that works for a particular state rather than choose an action that works for all possible states that could occur during a particular timestep.

Secondly, even if a trajectory with high fitness passes through a state multiple times, or passes through a set of similar states, the latter method has to learn an action for each timestep, while our method must only learn the action that corresponds to that state. For example, our method can learn a rule that whenever there is a state in which the left foot is behind the right foot by half a meter, and the center of mass is in front of the right foot, then raise the left foot and bring it forward. The other method must learn many corresponding rules, such as when the timestep is 40, raise the left foot and bring it forward, and when the timestep is 140, raise the left foot and bring it forward, etc.

Thirdly, if the rules of physics change slightly (for example, the simulation is run with a weighted backpack attached to the model, or the model will learn from a computer simulation, but be applied in the real world) then the latter method will quickly start making the wrong moves at the wrong time resulting in falling. To make matters worse, the robot will continue to make walking motions after having fallen. Our method, however, will gracefully adapt without needing to be retrained, because it will notice that the state is slightly different than expected, and choose a slightly different, and more appropriate, action.

Finally, if each run of the simulation has an unpredictable set of parameters, for example the model must be able to walk on unpredictably hilly terrain, then the static sequence of moves learned by the other method will quickly become inappropriate for the new trajectory, and the motion will fail, but our method will select an action that is appropriate for the current state regardless of the order in which those states occur in the trajectory.

State Features

In order to design a function $f(x)$ as described in the previous section, it is useful to select a set of simple features each of which can be quickly determined from the raw state. Together these features form a vector $\vec{v} = v_1, v_2, ..., v_n$ where each feature is an element of the vector.

The function f can now simply map a feature vector to an action vector. This mapping can be defined via a neural network, as we will see in the next section.

The efficacy of this approach depends highly on the choice of state features for a specific domain. We carefully chose twenty-six features.

The first thirteen features are the angles of the thirteen joints in the model. Together with the angle and position of a single rigid body, this fully specifies the state of the robot except body part velocities.

The fourteenth and fifteenth features are the x and y velocities of the center of mass of the robot respectively.

The sixteenth feature is the y component of the center of mass of the robot. The x component is not used as the robots motions should not depend on how far it has already walked.

The seventeenth, eighteenth and nineteenth features have to do with the right foot. The seventeenth feature is the x position of the foot relative to the x component of the center of mass of the machine. In general, all x components must be relative, otherwise they would not be consistent between different iterations of the walking cycle. The eighteenth feature is the y position of the right foot, and the nineteenth feature is the absolute angle of the right foot.

The twentieth, twenty-first, and twenty-second features are just as above, but for the left foot.

The final four features are the x (relative) and y positions of the head, and the x and y velocities of the head.

Many important features can be derived by these. For example, a useful feature for walking is balance. One way to get a feel for whether or not a 3d robot is balanced is to determine whether the center of mass of the robot is within the convex hull of the points on the robot's feet that touch the ground. In 2d, this degenerates to determining whether the x components of the feet have opposite signs (because they are relative to the center of mass anyway).

Neural Networks

As previously stated, the function $f(x)$ can be implemented using a neural network. We use a feed-forward neural network with twenty-six inputs and coincidentally twenty-six outputs, two outputs for each of our model's 13 joint.

A feed-forward neural network is a directed acyclic graph. In our case, the graph consists of two or more layers of nodes. Between each layer of nodes, there are edges from each node in the previous layer to each node in the next layer. The first layer is known as the input layer, and the last layer is the output layer. Each node has a real valued activation level. Each edge has a real valued strength. The activation level of each node in the input layer is simply the input vector of the neural network. The activation level of a subsequent node is calculated from the activation levels of the previous layer by taking the weighted sum of the previous layer's activation levels and applying the sigmoid function $s(x) = 1/(1 - e^x)$ to the result.

It has been shown in (Cybenko 1989) that a neural network with only three layers can approximate any continuous vector valued function to an arbitrary degree of accuracy if there are enough nodes. Adding another layer allows discontinuous functions to be approximated as well.

Since we did not know the nature of the function we were trying to approximate, we tried many combinations of the number of layers and hidden nodes. Many combinations did not seem to be able to create locomotion. Of the combinations that produced locomotion, no combination seemed significantly better than the rest. We settled on a neural network with three layers. The input layer contained 26 input nodes. The output layer contained 26 output nodes, and the middle layer contained 74 nodes.

In order to translate the values of the 26 output nodes into joint torques for the robot, we implemented a Hooke's law model. For each joint, one output was mapped to the spring constant of the model, k, while the other was mapped to the resting joint angle r. The torque was calculated as $t = k \cdot (r - a)$ where a is the current angle of the joint. A more common approach would be to simply output an angle, and simulate the model as if the angle were achieved. Not only is this unrealistic, but it leads to robotic looking motions that fail to react to unforeseen circumstances. One benefit of our method is that it is possible to limit the amount of torque the model is allowed to produce.

In our case, the weights of the edges are selected using an evolution algorithm. The mutation step of the evolution algorithm adds a normally distributed pseudorandom number with a mean of 0 and a standard deviation that was chosen to maximize performance to each weight.

We later experimented with sexually combined genomes by choosing each weight from the corresponding weight of either parent at random.

Evolution Algorithms

The evolution algorithm with which we obtained our first results is simple. We will refer to this as the simple evolution algorithm[3]. There is a pool of 60 genomes. Each genome is the list of weights for the neural network that controls the robot.

A physics simulation is run for each genome where at every frame the appropriate neural network inputs are calculated and given to the neural network defined by the genome. The function $f(x)$ is then calculated as defined for the neural network. From the outputs, the appropriate joint torques are calculated for that frame, and the simulation is advanced one timestep. The entire simulation is run for 40 simulated seconds (about 50ms real time) for each genome. A fitness function evaluates each trajectory, and the highest rated 20

[3]The details of this algorithm are being recalled from memory as the source code was subsequently modified, and may not be entirely accurate.

genomes replace the lowest rated 20 genomes. The algorithm then mutates the lowest 50 rated genomes (including the clones).

The fitness function is calculated in stages. During the first 60 generations, the fitness function simply measured how long the robot's head remained above 90% of its initial height (longer is better). For the rest of the generations, the fitness function measured the final minimum x position of any body part (higher is better), as well as adding a large penalty for each timestep that the head was below 90% of its initial height.

This algorithm gets stuck in one of several local minima. Most often, the algorithm correctly learns to stand, but fixates on a stable stance without ever progressing to forward motion. About 20% of the time, the algorithm results in a genome that makes indefinite forward progress, although it is usually more of a skipping motion than a walk. The problem is that the algorithm never switches the front and back foot.

In order to encourage the model to switch the front and back foot, we add a bonus to the fitness score whenever the legs switch relative positions.

Results

In about 15% of the cases, the model that encourages leg switching results in a legitimate foot-over-foot walk. The video can be watched here: `http://www.youtube.com/watch?v=GoRwwo1LkBI`. Unfortunately, the motion does not seem natural, the model is expending more energy than would be natural, and the head is being flopped up and down.

Without leg swapping, we get the skipping motion seen here: `http://www.youtube.com/watch?v=bLoTAf8vY3I`. Note that not every model is able to maintain forward motion. This is a local maximum that the evolution algorithm never breaks out of without the leg switching encouragement technique described above.

Sometimes this stable local minimum occures: `http://www.youtube.com/watch?v=HJwTLMHxdj8`. The algorithm found a way to bash its arm against the joint limit (set in Box2D), causing the whole system to lurch forward.

Lost Enhancements

As mentioned above, I experienced a computer crash during the summer after the original work. My solid state drive was wiped out and the most recent backup was several months behind. I lost several high quality enhancements which I will explain below.

I added parallelization to the algorithm which better took advantage of my computer's 4 cores (8 virtual). This leaded to a nearly 6x speedup which made manual iterative improvements to the framework much faster.

I also improved the simulation output by showing only the highest rated genome rather than each genome in turn.

One common failure mode was the algorithm choosing a leg for support and a leg for forward efforts. At this local minimum, the genome never produced leg after leg motion. However, since each leg in a true bipedal walker performs approximately the same function, the number of parameters of the model could be reduced while mitigating this local minimum by using the same set of weights for the right and left foot. This was most easily accomplished by running the network twice, once with reversed roles of the right and left sides. Not only did this make successful walkers more frequent in the population, but it reduced the number of iterations required to find a good set of weights substantially.

The physics simulator never quite seemed accurate no matter the parameters. The biggest obvious flaw is that the foot collisions with the ground were typically not solidly connected. The feet would bounce up and down slightly which allowed horizontal motion of the lower foot despite an infinite coefficient of friction between the foot and the ground. In order to alleviate some of the problems this caused, I modified my bipedal model to have shock absorbers in the legs. This reduced the amount of bounciness, but was only moderately effective.

In order to produce a robust model that could predict terrain and recover from unexpected events, noise was added to the simulation. Furthermore, random terrain was generated for each trial of the algorithm. The terrain was generated using a two dimensional fractal landscape algorithm where the starting and ending point were set to have the same elevation. The magnitude of the bumpyness was set to a constant that started at zero and increased to an asymptotic maximum with the number of generations of the genetic algorithm. The neural network architecture was modified to give a read of the height of the terrain at various points below the walker. Ultimately the walker was able to handle slight hills in the landscape, but it was never fully robust.

Reinforcement Learning

After losing my code I decided to start again from a previous backup. Although the lost enhancements really helped the simulation, the motion never looked perfect and natural. I thought I might have more success with reinforcement learning.

There are two major challenges to adapting common model-free reinforcement learning schemes such as Q-learning to this problem. The first is that both the input state information and the output action are continuous vectors, so representing the value of a state-action pair cannot be done with a lookup table (the typical method). The second is that even if the value of a state-action pair were to be represented somehow, it would be difficult to find the action that maximizes the value for a given state. I tried two methods for representing the value for each state-action pair.

The first method was artificial neural networks where the input was the the concatenation of the state and action vectors. For this I used a network with 39 inputs (the state vector and the action vector) and a single output for the value of the state action pair. Various parameters were chosen for the number of hidden units and the number of layers. Under this scheme, a single neural network is developed rather than a set of 60. At every timestep, the action vector that maximizes the q-value is chosen using gradient descent with random restarts (the gradient of each input parameter was measured using a technique similar to backpropagation). The agent takes the maximum action which results in a new state and a reward (which was simply the change in the minimum x-value attained by any portion of the robot). The maximum is once again calculated for this new state, and the value plus the reward is backpropagated to the previous state with some decay. In theory, this process will converge to a neural network-defined policy that maximizes the long term reward. Unfortunately, I could only get the system to work with the simplest of robots (a single panel with a wheel).

The more complex the robot, the more difficulty the neural network had in representing a value function. The result was chaos, and I decided to try another regression technique, regression trees. Like neural networks, regression trees map a real valued vector to a real valued output. They start with a simple linear regression on the inputs. Whenever the error for the regression exceeds a threshold, presumably because of nonlinearities in the function, the system branches, creating two linear regressions. One for one half of the input space, and another for the other half. This process is repeated fractally until the desired error is reached. There were a few major obstacles with this approach. The first is that the functions represented are not differentiable so gradient descent was not possible for finding the action that maximizes the value. The second was that regression trees were unable to reach an acceptable error level without using up all available memory quickly. Finally, networks had to be thrown away as old values didn't correctly take into account new information. I tried to solve the first by sampling the space of actions repeatedly. I was unable to solve the second or the third.

I abandoned the reinforcement learning approach, although I believe it could be made to work abstractly, the code became overwhelmingly complex with my implementation.

Future Work

A more effective solution to the problem of accurate foot friction with the ground would almost surely help tremendously. Box2d allows joints to be created and destroyed at any time during the simulation. Furthermore, the newest version of Box2d supports a number of new joint types, including what is called a "wheel joint" the wheel joint will bind a rigid body to another in a way that allows linear motion along a specified axis, and rotation of one body about the anchor that connects it to the joint. Such a joint is ideal for a wheel with a shock absorber, but it could be made to support an infinite coefficient of friction at a distance. After every frame of the simulation, I can create a wheel joint between the lowest point on the foot and the ground if the foot is less than a certain distance from the ground. The joint would allow rotation of the foot, and vertical translation with the ground, but would block all horizontal motion until the foot's distance with the ground exceeded some small threshold. This would effectively create the desired infinite friction even if the simulated foot does break contact with the ground from time to time.

Another idea I think would be promising is heuristic search. By discretizing the action space, the state space of the machine could be searched in an efficient way using a heuristic search algorithm like IDA*(Korf 1985). A goal node could be one in which the machine has completed a step cycle and has made a certain amount of horizontal progress. The heuristic evaluation function could calculate the minimum number of simulation steps that would have to be run for each rigid body to move directly to a goal position assuming a maximum speed or a maximum acceleration/deceleration. On a modern machine, I suspect this search could be completed, yielding an optimal set of steps for any given machine position. Finally, a neural network could learn the correct output for any given input directly, leading to a controller with the desired speed and accuracy (or at least a great place for the evolution algorithm to start).

Finally, I think evolution algorithms can be improved in general. Evolution algorithms typically report the highest ranking genome after a number of iterations. However, instead of reporting the highest ranking genome, I believe it might be better to report the median of top genomes. For each real number in the genome, calculate the median across say the top 15% of genomes and return the result. This would overcome randomness that makes the top genome better overall, but worse in some ways.

Conclusion

Evolution algorithms are hard. While it is possible to get good results for many problems, it requires painstaking hand tuning to get right. Nevertheless, they continue to be more promising than reinforcement learning in this domain.

Also, keep a daily backup of all work in the cloud.

References

[Chestnutt et al. 2005] Chestnutt, J.; Lau, M.; Cheung, G.; Kuffner, J.; Hodgins, J.; and Kanade, T. 2005. Footstep planning for the honda asimo humanoid. In *in Proceedings of the IEEE International Conference on Robotics and Automation*.

[Cybenko 1989] Cybenko, G. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)* 2:303–314. 10.1007/BF02551274.

[Erez 2010] Erez, T. 2010. Local optimization for simulation of natural motion. In *AAAI*.

[Korf 1985] Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

[Reil and Husbands 2002] Reil, T., and Husbands, P. 2002. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Trans. Evolutionary Computation* 6(2):159–168.