Learning Humanoid Jumping using Neural Evolution

Toni Ivanov
Computer Science
UCLA
destinator84@ucla.edu

Kevin Lu
Computer Science
UCLA
kevlu94@gmail.com

Yichen Chen
Mathematics
UCLA
ycchen39@ucla.edu

***Abstract*—We built an artificial bipedal character from scratch using OpenGL and ODE. The character's motion is learned through an artificial neural network whose structure and weights are evolved using NEAT. The character was able to learn how to jump up in the air from a standing position and land back on the ground using a piece-wise fitness function to guide the evolutionary process. We implemented this in C++ and ran it in real time.**

1. Introduction

In 3D character animation, designers and developers have been able to achieve greater visual realism through the use of realistic physics engines. One popular area of relevant study in the literature is human locomotion [8], which is essential for animating virtual characters in motion picture. Even interactive gaming has begun to adopt realistic locomotion into its arsenal. However, an aspect of human motion that can be more complicated is human jumping, as it involves multiple unintuitive steps. At the same time, it must still deal with many of the difficulties inherent in all physically realistic virtual bipedal experiments, such as the task of keeping the creature balanced on its feet. In this paper, we explore the area of bipedal jumping and landing, and how to learn its motion using an evolving artificial neural network.

The components of our system and their interplay are depicted in Figure 1. A simulation environment was set up with a virtual character that was animated using a physics engine. A neural network would drive the character's motion by passing the network output, which is a set of joint torques to impose on the character, into the simulator. In turn, the character's joint angles were frequently measured and fed into the network as inputs. Evolutionary methods were used to modify the network's structure and weights. Various other measurements were taken from the simulation environment to build the fitness function that guided the evolution.

2. Related Work

Prior methods for humanoid robot control largely fall into two categories – reinforcement learning [6] and

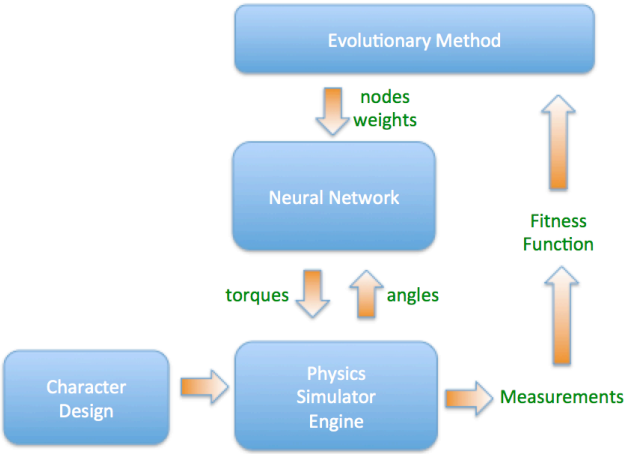

Figure 1: System diagram that shows the components of the work presented here.

artificial neural networks [5]. While the former method is reverse engineering inspired, the latter method is biologically inspired. Our group explored the second method and its application towards jumping. M. Lewis first started using the genetic programming approach to evolve neural network controllers [1]. Since then, many works have been devoted to improving the effectiveness of the genetic algorithm applied to neural network evolution. The main challenge in application is that the problems have enormous search spaces. In response to this problem, Kenneth O. Stanley proposed using NeuroEvolution of Augmenting Topologies (NEAT) [9]. How NEAT works is that it starts with a small, simple network and then convolutes the network topology. Features like historical markings and speciation make NEAT especially suitable for evolving networks that control innovative behaviors. Robot control problems including bipedal walking [8] and aquatic station keeping [4] have been effectively solved by NEAT. In [8], NEAT was adapted in the genomic distance calculations for the large networks needed for bipedal locomotion. In [4] NEAT is optimized by using a cumulative fitness function to control behaviors that indirectly contribute to search objectives. Stanley also tailored the fitness function construction for evolving innovative neural network topologies by abandoning

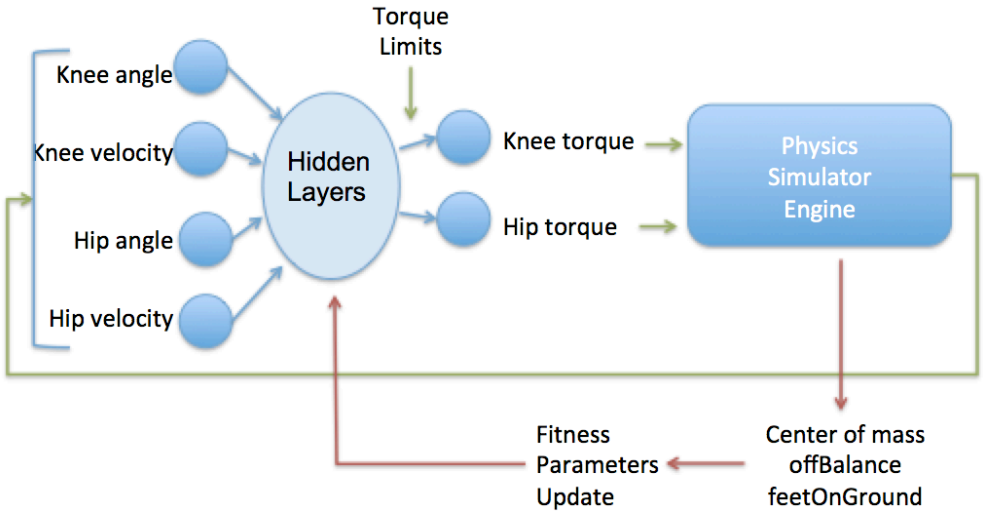

Figure 2: Driving the character's motion using neural network.

search objectives altogether, and instead evolving the network through search for novelty alone. [7] In addition to artificial neural networks, the central pattern generator is also a biologically inspired controlling method for humanoid robots [2].

3. Simulation Environment

Our graphical user interface is rendered using OpenGL and the motion is simulated using Open Dynamics Engine (ODE) [10]. The simulated 3D scene consists of a simple flat ground surface and a virtual humanoid constructed from rectangular blocks. The lengths of the limbs are based on anthropometric measurements of the human body [3]. We used only hinge joints because only motion in the coronal plane is relevant to jumping. The torso and the head are rigid, and the arms swing freely with respect to their hinges. The torque applied to the hip and the knee joints are controlled directly by the neural network. Note that no proportional-integral-derivative (PID) controller or other intermediate functions are used for driving the motion.

During the evolution process, the simulator runs without rendering, causing a significant speedup. The simulator updates the state of its objects every small fixed time step. Every few time steps, the simulator takes several measurements to pass into the neural network, and accepts a set of torques from the neural network to act on the knee and hip hinges.

The measurements that are extracted from the simulator include the joint angles and joint velocities, as well as the position of the character's center of mass. Additionally, we keep track of whether the character is off balance and whether the character's feet are in contact with the ground.

4. Neural Network Controller

Neural networks have the ability to represent arbitrarily complex functions and are thus a good candidate for learning jumping. In fact, all motion of our virtual character is controlled by a single neural network. Currently only the hips and knees are used for our jumping simulation. The interaction between the neural network controller and the simulation environment is shown in Figure 2. There are four inputs to the network, namely the knee angle and velocity and the hip angle and velocity. There are two outputs from the network – the knee and hip joint torques. The outputs of the network are scaled to be within the torque limits of the physical human body. The internal network nodes and their interconnections are determined by the evolutionary process discussed in the next section. The activation function of each neuron is defined as follows:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The virtual character starts in a standing position and is moved through an iterative process. At the beginning of each set time interval, the simulation measurements are passed into the neural network, and the two torques outputted from the neural network are applied to the character's joints to directly drive its motion for the remainder of the time interval. The center of mass of the character is monitored in order to update the fitness function that guides the evolutionary process. If the character is off balance, the iteration is terminated immediately and a final fitness value is assigned to the current neural network, indicating how successful it is in achieving jumping. After evolutionary modifications to the network, the process above is repeated again starting from the initial standing position. Note that this process involves no training data from human motion.

5. Evolving the Neural Network

We chose NEAT as our neuroevolution method. Our

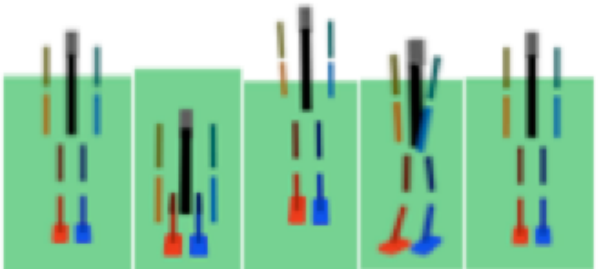

Figure 3: Stages of jumping. First is initial standing, then squatting, than lift off in the air, then landing, and standing back to initial position.

NEAT code is adapted from the single pole-balancing problem in the original NEAT implementation [9]. It is also heavily based on the NEAT utilization in bipedal walking [8], and that in aquatic station keeping [4].

In NEAT, each neural network is called an organism. The whole evolving population has 150 organisms, i.e. 150 neural networks. Sensors, neurons, and outputs are all treated as nodes, and the links between the nodes are called genes. Through the evolution process, the neural networks will add/remove neurons, add/remove links, and change weights on the links. A sample initial network is shown in Figure 4.

Like all evolution methods, the networks are selected based on the fitness values they have.

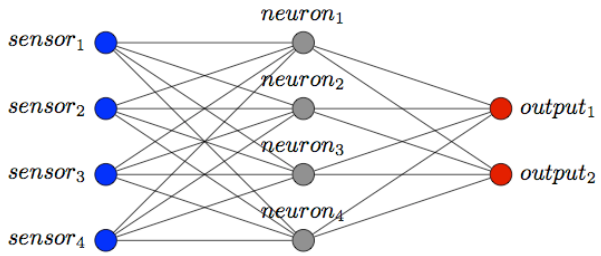

Figure 4: Sample neural network for NEAT

6. Fitness Function Design

We designed our own fitness function to guide the evolution towards the goal of jumping. Jumping is composed of four stages: squatting down, jumping up in the air, landing on the ground, and lastly standing back up (Figure 3). Measuring progress in this multi-stage motion required a piecewise fitness function that optimized each stage separately. In contrast, the fitness function for bipedal walking simply measured forward progress. It was not possible to achieve our objective with a simple function such as optimizing for the height above ground. This is because the evolutionary method would greedily select jumping up directly from a standing position, which would only produce a small hop.

Our fitness function was based on the vertical position of the center of mass of the character. The minimum and maximum height of the character was tracked throughout the simulation of a particular neural network. The different stages of jumping mentioned above were detected and the fitness optimization objective was changed accordingly. It was not possible to optimize the fitness in one stage because the method would get stuck in a local minimum. For example, an objective function such as the difference of maximum and minimum height could get stuck in a squatting position without being able to stand back up. Other attempts using a simple fitness function resulted in the character jumping high up into the air without being able to land on his feet.

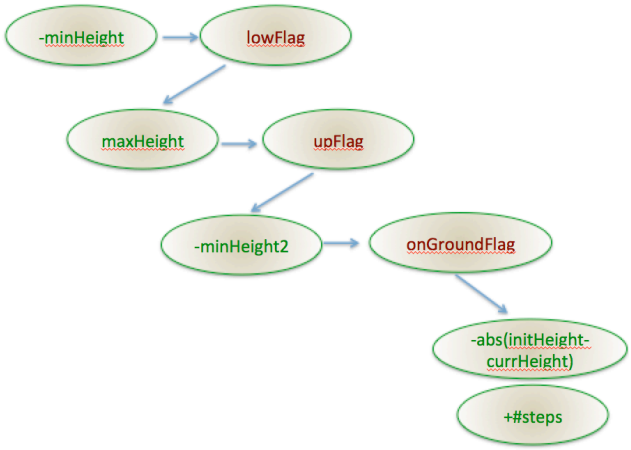

Figure 5: Piece-wise fitness function. Different stages of jumping are detected and the fitness optimization criteria are changed accordingly.

The fitness function is given by the following equation:

(1-lowFlag)(10-minHeight)+*
lowFlag(10+(1-upFlag)*maxHeight+*
upFlag(10+(10-minHeight2)+*
onGroundFlag(10-abs(initHeight-currHeight)+steps)))*

A graphical representation of this function is shown in Figure 5 for better illustration. First we optimize for the minimum height of the character so that it squats down from the initial standing position. Once we detect the event of squatting, a "low" flag is activated and the fitness is further improved by raising the maximum height of the character. Next, we detect when the character is high up in the air and activate the "up "flag. At this point the fitness is improved by targeting again the minimum height to return the character back to the ground. Once the ground is reached, an "on ground" flag is activated to indicate that the character should return to the initial standing position after possibly squatting a bit from the impact with the ground. In this final stage the fitness improves proportionally to the closeness of the current height to the initial height. To achieve balancing, the number of time

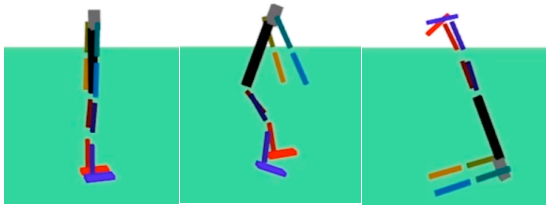

Figure 6: Intermediate results for learning jumping: balancing (left), jumping up in air (center), headstand (right)

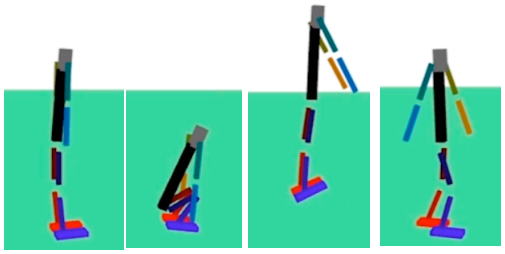

Figure 7: Final jumping result: standing, squatting, jumping, and landing

steps is added to the fitness value so that the character stands as long as possible when the jump is completed.

7. Results

The experimental process for learning jumping involved different stages that incrementally improved the results (Figure 6). Initially our character simply learned to balance. Because joint torques were imposed once every time interval instead of continuously, the character's balancing animation involved a back-and-forth oscillation instead of standing up still. After this, the character learned how to jump slightly in the air but would fall face first onto the ground. One interesting intermediary result was that after squatting and falling on its hands, in order to get back up and raise its center of mass, the character performed a headstand. After several attempts, the character was finally able to jump in the air and land on his feet (Figure 7).

We completed 5000 evolutionary iterations of a population of 150 different networks. This took about 10 hours of computation. Our final network consisted of about 60 hidden nodes and 250 edges between nodes.

8. Future Work

We looked at several aspects of the system that could be improved.

The first aspect is the physical character model. We currently control only the knees and hip joints while the arms are freely swinging. It is desirable to also control the motion of the arms in order to aid jumping for a more realistic animation.

The second aspect is the fitness function. We could use a stochastic optimization to tune the fitness parameters, such as weight coefficients. To create different jumping at a target height, we can create a more involved procedure for setting the flags that are used in calculating fitness. At the same time, novelty search [7] can be used to achieve alternative jumping styles. Novelty search is a method that rewards innovative jumping behaviors over achieving certain heights of jumping.

Last but not least, advanced jumping scenarios can be learned for the purpose of engaging animation. These scenarios include jumping from a running start and single foot jumping. These can both be achieved by using multiple neural networks. The latter scenario would involve two separate neural networks for the left and right sides of the body, and evolving them independently.

References

[1] Lewis, M. A., Fagg, A. H., & Solidum, A. (1992, May). Genetic programming approach to the construction of a neural network for control of a walking robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on* (pp. 2618-2623). IEEE.

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

[3] Department of Defense, United States of America, *Anthropometry of U.S.MilitaryPersonnel(DOD-HDBK-743A)*. DepartmentofDefense, United States of America, 1991.

[4] Moore, J. M., Clark, A. J., & McKinley, P. K. (2013, July). Evolution of station keeping as a response to flows in an aquatic robot. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation* (pp. 239-246). ACM.

[5] Vaughan, E., Di Paolo, E. A., & Harvey, I. (2004). The evolution of control and adaptation in a 3D powered passive dynamic walker. In *Proceedings of the Ninth International Conference on the Simulation and Synthesis of Living Systems, Artificial Life IX* (pp. 139-145).

[6] Morimoto, J., Cheng, G., Atkeson, C. G., & Zeglin, G. (2004, April). A simple reinforcement learning algorithm for biped walking. In *Robotics and Automation, 2004. Proceedings. ICRA'04. 2004 IEEE International Conference on* (Vol. 3, pp. 3030-3035). IEEE.

[7] Lehman, J., & Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, *19*(2), 189-223.

[8] Allen, B., & Faloutsos, P. (2009, October). Complex networks of simple neurons for bipedal locomotion. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on* (pp. 4457-4462). IEEE.

[9] Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, *10*(2), 99-127.

[10] Smith, R. (2005). Open dynamics engine.