

# QWOP Bot Final Report

Written by Matthew Oros, Sonny Smith, and Michael Terekhov

Baldwin Wallace University  
275 Eastland Road  
Berea, Ohio 44017

## Abstract

QWOP is a challenging 2-dimensional game where the goal is to make the main character walk/run to the right as far as possible without falling. The game's difficulty lies in its physics simulation, which makes movements hard to predict and requires a quick reaction time as well as an understanding of how the combinations of key presses translate to a desired movement on the screen. The character is controlled using four keyboard keys: Q, W, O, and P. Q and W control the thighs, and O and P control the knees by either rotating the joints one way or the other. As soon as an upper-body limb contacts the flat ground, the game is over and restarted. The score of the game is based on the distance traveled by the character.

In this project, we propose a simulation that allows an agent to learn to control the character and play the game of QWOP. Two main implementations are considered and attempted, both based on neural networks. The first method is a genetic algorithm approach to learning, while the second approach is a deep Q-learning method.

The genetic approach is inspired by natural selection where the best performers get to reproduce and spread their genetic information to their offspring. The new population then replaces the old one and plays the game again. This process is repeated until the agents play the game to a satisfactory level. The deep Q-learning approach is a powerful reinforcement learning method that approximates the Q-function, which measures the expected reward of taking a certain action in a given state. By combining the benefits of deep learning and reinforcement learning, the deep Q-learning approach can learn to make optimal decisions in a wide range of environments, even in situations where the optimal policy is not known in advance or labeled data is not available.

## Proposed Implementation

The proposal of the simulation is an agent that is able to learn on its own to control the character to play the game of QWOP. The game is a basic simulation of physics bodies where the joints that connect them were actuated by a key press. This simulation involves knees, thighs, hips, and upper body, which includes head, neck, arms, and forearms, where calves and legs can be controlled.

Two main implementations that will be considered and attempted will both be based on neural networks. The first

Figure 1: The original browser-game QWOP



method is a genetic algorithm approach to learning while the second approach will be a deep Q method of learning. Based on the information presented in the journal, Deep-Mimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills, a neural network approach is chosen for both methods as deep neural networks. Based on the results presented in the journal it was possible to successfully reproduce highly-detailed human movements including walking, running, cartwheel, and flipping, which is what we are interested in our experiment.

In order to create a suitable learning environment for the agent, the original game was reimplemented. The original game is browser-based and would require interfacing code to grab relevant data and to simulate key presses. The logistics of this are not relevant to the desired outcome or exploratory process of the project and thus the approach of reimplementation of the game was chosen.

The programming language that was chosen was Python. Python was chosen due to its simplicity, flexibility, and plentiful collection of libraries, especially related to machine learning. For the simulation itself the graphics library chosen is Raylib. Raylib is a native C library but has various bindings to other languages such as Python. It was chosen due to its simplicity where it is not a graphics engine but rather abstracts OpenGL drawing calls to functions. This helps to decouple the simulation from the graphics. The physics library chosen is PyMunk. It is a simple 2D physics library where physics bodies are created with certain shapes and properties

and added to a simulation space. PyMunk was also chosen for its simplicity and flexibility.

## Neural Networks

Once the progress on the physically-simulated character was acceptable, work began on creating neural networks. The neural network for the genetic approach was originally created without the use of any libraries. A neural network consists of input neurons that connect to hidden neurons which then connect to output neurons. The strength of the connections between neurons is called weight. Each neuron accepts a single or multiple floating point values which then get summed and normalized using an activation function. The activation function chosen was the sigmoid function as it is a very common choice. Which is defined as:

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

In order to utilize a neural network, the inputs that are fed to the network must be determined. These inputs would denote which aspects of the environment the agent would be informed about. If too little information is forwarded to the network, then it may not be able to converge on an acceptable solution. If too much data is sent, the network may also have difficulty converging with having to weigh the numerous inputs. In the implementation, the x and y relative positions to the body of each limb are fed to the network. We hypothesize that this would give the network enough information to determine how it should move its limbs when in a certain physical configuration.

Initial work was made on an early termination condition where if certain upper-body limbs contact the ground then the simulation ends for that particular character and its reward function is adjusted accordingly. An advantage of early termination is that it can function as a curating mechanism that biases the data distribution in favor of samples that may be more relevant for a given task (Peng et al. 2018).

It is crucial to note that neural networks exhibit variations across different approaches.

## Genetic Approach NN

In the genetic approach the simpler version of the neural network was utilized since it was constructed from scratch. The neural networks consisted of 24 input nodes, one hidden layer with 12 hidden nodes, 4 output layers, and bias for each connection between each layer.

## DQL NN

In the DQL approach more neurons for input and hidden layers were added due to the fact of using neural networks provided by torch library. The number of neurons in the input layer increased to 31 neurons for all the limb positions and additional information such as velocities and a loop-back neuron. The number of neurons in the hidden layer increased to 256 for additional complexity since DQL algorithm requires a more complex and flexible neural network.

## Consideration of Backpropagation

A traditional method of training a neural network is to use a method called backpropagation. Backpropagation is done by providing the network with input data and labeled output data where the term labeled refers to the desired output of the given input. It is to be noted that the term backpropagation used in this context is referring to the entire learning approach while it can also be used to refer to a step in a different overall learning approach such as deep-q learning. In this context, it is referring to an overall learning approach where backpropagation is the main overall training step. The input is fed forward through the network in its current state and its output is compared to the labeled output. The weights are then adjusted from back to front based on how incorrect the network's output is.

The difficulty with this approach for this domain of the problem is that there is no simple method of obtaining such labeled data in this problem. The game of QWOP is difficult for humans and as such the goal of the simulation is for the learning process to generate a solution from its fitness feedback and environment rather than by example from a human playthrough. Thus, the two methods that were chosen incorporate some form of random exploration of the state space to converge on a desired solution.

## Genetic Approach

The genetic approach is inspired by natural selection. In each generation, a specific amount of agents is created and each has a distinct neural network (genes). These genes can be mutated, recombined, and evaluated for fitness. Based on their fitness, individuals are selected for reproduction, which involves combining their genes to create new offspring. The new population then replaces the old one, and the process is repeated until the agents play the game to a satisfactory level.

Our approach is based on the simple evolutionary algorithm that is presented in the article *Evolving Controllers for Virtual Creature Locomotion*. As it is described in the article, An evolutionary algorithm is a type of search method that imitates natural evolutionary processes. It is used to solve optimization problems by defining a fitness function that measures the quality of the solution candidate. In this case, the solution candidate is a description of a locomotion controller, and the fitness function is based on how far a virtual creature can move during a physically-based simulation within a fixed time.

The approach that is utilized includes three phases. The first phase is to create 100 agents where each has a distinct neural network. For the first generation, the weights for each neural network are randomly assigned. When the agents are created they perform actions that are based on inputs of their limbs' locations. After the time that is specified passes, the program moves to the second phase. In the second phase, the top 50 performers are chosen based on their fitness function. Then the neural networks of the performers are randomly chosen to create new ones that will be used for the next generation. The process of creating a new neural network involves picking two random top performers, taking their neu-

ral networks, and mixing them. The mixing is based on the specified probability, in our case is 50%. This means that for each specific connection/weight between the layers half of the time the algorithm will choose the connection/weight of neural network *A* and half of the time of neural network *B*. Mutation probability is also included and is 5%. A mutation is important in this case since it introduces new genetic material into the population, which can help to avoid premature convergence of the algorithm on a suboptimal solution. It is important to mention that we keep the top five performers for the next generation and create 95 new neural networks based on the top 50 performers. Hence we have 100 neural networks for the next generation. Such a technique is called elitism and is useful to preserve the best solutions found throughout all generations. Elitism allows us to build upon the best solutions and converge toward a desirable solution more quickly. The third phase is to create new agents which will have the newly created neural networks and let them play again. The process is repeated until the desirable solution is achieved.

## Deep-Q Learning Approach

The proposed method is based on deep Q-learning, a model-free, bootstrapped, off-policy learning algorithm. This implies that the agent does not require any prior knowledge of the environmental dynamics. The agent learns by interacting with the environment and collecting experience samples. This learning approach is promising as the agent learns to perform various skills through trial-and-error, thus reducing the need for human insight (Peng et al. 2018). The algorithm constructs estimates of the action-value functions, which represent the expected return of taking each action in a given state, based on previous estimates. This is a bootstrapping process, where one estimate is used to update another. The algorithm also employs an off-policy strategy, where it uses an epsilon-greedy policy to generate actions that explore the state-action space with a probability epsilon and exploit the current best action with a probability 1-epsilon. The data generated by this exploratory policy is used to update a purely greedy policy that maximizes the action-value function. We implement decay into our epsilon based on the considerations made in the article *RBED : Reward Based Epsilon Decay*. We want the epsilon value to decay so that the agent employs more exploitation than exploration as it gains more information about its environment, as once the agent acquires the necessary information to interact optimally with its environment, allowing the agent to exploit its knowledge makes more sense.

The agent possesses various types of memories, namely state memory, new state memory, action memory, reward memory, and terminal memory. These memories are utilized to calculate action-value functions, which are crucial for learning. The terminal memory stores the done flags that indicate when the exploration period ends and the states that need to be updated with the agent's estimates of the action-value *Q* are passed. To enable the intelligent learning of the agent, batches of memories are used. The batches allow the data to be used for updating the agent's neural network when they are full. By processing the training data in batches, the

neural network can update its weights more frequently and efficiently.

Mean Squared Error (MSE) is a loss function that is applied. A loss function measures the difference between the estimated action-value function ( $Q(s, a)$ ) and the target action-value function. The target action-value function represents the expected return after taking an action in a given state, and it is computed as the sum of the immediate reward and  $\gamma$  (discount factor) times the maximum value for the next state.

$$\text{target} = \text{reward} + \gamma \cdot \max(Q(s', a'))$$

The function produces gradients with respect to the weights of the neural network. The gradients are the measures of how much each weight contributes to the overall solution using backpropagation.

An optimizer, Adaptive Moment Estimation (Adam), is utilized. Adam uses the gradients of the loss function to update the weights of the neural network that correspond to the global solution. Adam knows in which direction the weights should be nudged by calculating the first and second moments of the gradient from the loss function. The first moment is the moving average of the gradients. The second movement is the moving average of the squared gradients. Adam then adjusts the learning rate for each weight based on these moments, making it larger for weights with low variance, meaning the gradients that are consistent and stable, and smaller for weights with high variance, meaning the gradients are more volatile and unpredictable. This way, Adam can find a good balance between exploring and exploiting the search space and converge faster to a minimum of the loss function.

After initial implementation and testing, it was clear that improvements needed to be made to the inputs of the neural network. A major improvement was inputting relative positions of the character's limbs rather than absolute positions. This change was an improvement because as the character would move further to the right, the inputs would change drastically even if the character's body configuration is still similar which is undesirable. Another improvement to the inputs of the network was the addition of velocity information. The x-axis velocity for both feet and the x-axis and y-axis velocity for the torso were added. This was done to help give the character more information about how it is moving rather than just positional information. Another change was more experimental and it was to add an extra output to loop back to an input. The intention was to allow the character to have some sort of short-term memory by passing an output value from one step of the simulation to the next.

The outputs of the simulation were also modified. First, there were four outputs, each corresponding to the Q, W, O, and P keys to move its body. However, in the original game, it is possible to press multiple keys at once. Thus, four more outputs were added for the Q-O, Q-P, W-O, and O-P combinations for pressing multiple keys at once since only one action can be chosen for a given simulation step. These eight outputs combined with the one loop-back output discussed previously complete the current nine-output network.

With these changes made, another run of the simulation

yielded results that indicated further modifications were necessary. As the epsilon decreased, the character would perform random actions less frequently. This is an intentional and expected result of the learning process. However, there seemed to be a point, in multiple runs, where the character would start performing worse as epsilon decreased further rather than better. After analyzing the results, it was hypothesized that the character was relying on random muscle movements at times to help propel itself forward and so when epsilon was decreased too much, the character performed worse. With this hypothesis, two strategies were employed to mitigate the issue. The first was to add another input that is a random number generator. This input would send a random value to the character at every simulation step. The idea was that if the character relies on random movements, then adding a random input might help it retain this behavior as epsilon continues to decrease. The second strategy was to decrease epsilon much slower and run the simulation longer. It could be that epsilon was decreasing so fast that the behaviors the character learned were not able to adapt fast enough to a lower-epsilon environment where there was a lower frequency of random muscle movements.

## Implementation of Genetic Algorithm

In order to maximize the efficiency of learning for the genetic algorithm portion of the project, it is important to simulate many generations at a decent speed. To maximize the throughput of the simulations, each generation is simulated in batches where each batch is simulated all at once. For example, a generation of 100 characters is split into 10 sub-generations each containing 10 characters. These characters are all simulated at the same time. This is able to be done easily as PyMunk (the physics library used) is able to simulate multiple isolated simulation spaces. Since the graphics and physics simulation is decoupled, this allows us to visualize all 10 characters in the same sub-generation while they are all actually being simulated in their own simulation space, not influencing each other.

Figure 2: A simulated batch of characters during a sub-generation



The number of characters could be increased for the sub-generation but there is a speed limitation where it would reach a point where the number of characters would slow down the simulation thus negating the added benefit of parallelism.

## Implementation of DQL

In our implementation, we gave 500 games for the agent to play and learn. The initial setup of the agent involves setting its gamma to 0.99, epsilon to 1.00, minimum epsilon to 0.01, learning rate that is used by the optimizer to 0.001, and epsilon decrement to 0.000001. The same agent is used from one episode to the other but with the developed neural network. The development was conducted using the optimization algorithm (Adam), which is described in more detail in section . From one episode to the other the epsilon decreased by 0.000001, meaning the agent was using its best-known actions more often as the simulation progressed. The decrement of the epsilon stops as the epsilon gets as small as 0.01, which means that by the end of the simulation, the agent uses only a low percentage of the random actions. The percentage of the random actions taken by the agent is not specific due to the fact that we have an input node that takes in a random floating value due to the reasons mentioned in . During each episode, the agent goes through the process of taking the best known or random action for which it receives the reward which is calculated by equation:  $(fitness \times 0.5 + character.position.x) \times 0.01$ . The fitness function is initially set to be 10, and decrements by 0.01 with every step. If the agent moves forward, the fitness is increased by the distance the agent traveled. If the fitness function gets to zero, which means the agent does not produce any meaningful progress the current episode ends, the changes to the neural network are made, and the new episode starts.

## Problems Encountered and Avoided Environment

Throughout development, several problems/difficulties were encountered along the way. First, we had to understand the library that we used to develop a physical character and the environment that it would be in. The first difficulty that we had on the way was figuring out how the libraries that we used to build the character and the environment worked. To create a character, we had to figure out the limits and capabilities of these libraries. To create a character and the environment, we had to figure out how to set up physics, how to create a graphical object, how to give the object physics, how to connect this object with other objects, and how to give them muscles so it can be controllable.

## Agent

The next difficulty we encountered is the structure of the neural network for agents. The difficulty involved choosing how many input nodes were needed, what kind of data will be used for the neural network, and how many hidden layers were needed. As it is mentioned above the decision was made to have distinct neural networks for different approaches. The structures of neural networks that were used in this project are described in more detail in the section .

The second difficulty concerning the agent was setting up the collision detection. The initial idea was to have collision detection in the environment so that when specific objects collide inside of it, we would get some type of output. The

approach turned out to be inappropriate as we progressed forward since we needed to know which agent had contact with the ground. The fix to the problem was to include collision detection inside of the agent class from which each agent was created. That way each agent had its own collision detection, which indicates when the agent touched the ground with its upper body.

### Simulation Time

An unexpected issue during development was the use of real-world clock time in areas of the simulation. The learning process of the simulation can take numerous iterations, especially for the genetic algorithm to converge on a satisfactory result. Thus a method for increasing the simulation rate was implemented. However, since real-world clock time was used, this resulted in unexpected behavior in multiple areas of the application. One example is the time for each generation. Each generation would last three seconds after the maximum distance that was achieved by one character has not changed. However, when the simulation rate was increased this timing became inaccurate. All time-related code was converted to a simulation time which has no relation to real-world time. This allowed us to increase the rate of the simulation while maintaining the exact same results as if it were run at a slower, more normal-looking rate.

### Visualization

To collect data from our program, we initially implemented the ability to save the program's current state as a JSON file. As we further developed our program, we expanded the save functionality to separate the data of each generation that was generated in the genetic algorithm. The JSON that we save includes the color of each runner, the bias values of the neural network, and the fitness value of the runners. We replicated a similar saving process for the deep q-learning algorithm to save each episode's values for epsilon, score, average score, and maximum reward. We decided to utilize the Pandas and Matplotlib libraries in Python to visualize our data, as we were already working with Python for our program, and we have experience using those libraries to manage and visualize data. The JSON files are imported into the visualization program where they are concatenated into a single data frame. For the genetic algorithm, it is not useful to visualize the network biases, so they were dropped from the data frame. The visualizations that we intend to use include a histogram of the fitness values of each of the runners in a given generation, a graph of the average fitness values for each generation of runners, and a dual-axis graph that shows the average score and epsilon values for each episode of the Deep Q-Learning algorithm.

## Results

### Genetic Approach

For the genetic approach, the simulation was run for about 8 hours, which allowed the algorithm to go through 756 generations. By doing so it was possible to capture enough data to support our hypothesis which involves the agents performing better over some time due to better genetics being mixed.

Figure 3:

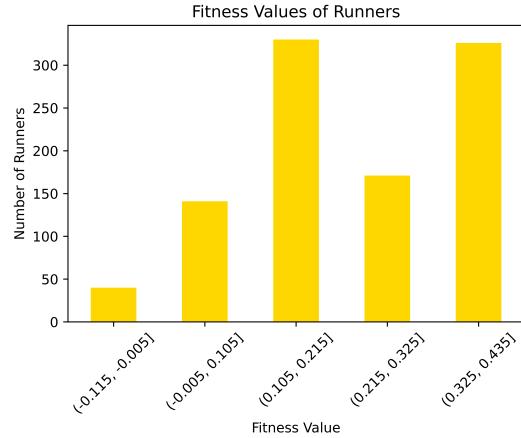
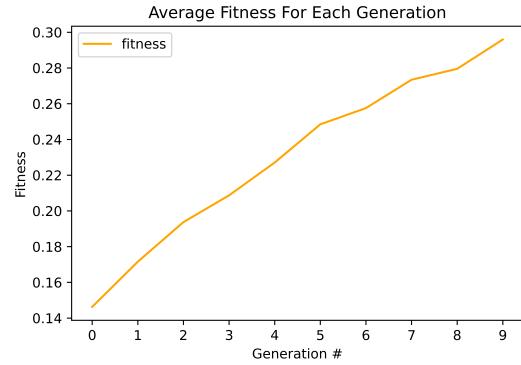


Figure 4:



As a result of 755 generations, the agents were able to move nearly 10 meters forward, which can be considered good progress.

Based on the provided data above it can be conducted that there was a spike in progress in the first 50 generations. From the 100th generation to the 755th generation a steady increase of the average fitness of all agents for each generation can be seen.

The other aspect that is important to mention is the frequency of fitness values. Based on the data above it can be seen that the most frequent fitness values for the agents are around 1 meter. Nevertheless, it is very important to mention that fitness values such as 7, 8, and 9 meters appear as well, which appeared by the end of the 8 hours simulation period. This means that by the end of the simulation, the genetic algorithm was able to produce neural networks that were able to produce action given the limbs' positions which would move the agent forward. Based on the results that were received the number of agents that were moving as far as about 7 meters increased and consisted of 3 agents. If the simulation was to run for a longer period of time we are to expect to see more agents/neural network to produce similar or better results due to the logic that was implemented in our approach, which involves keeping top 5 performers for

Figure 5: Average fitness for each generation

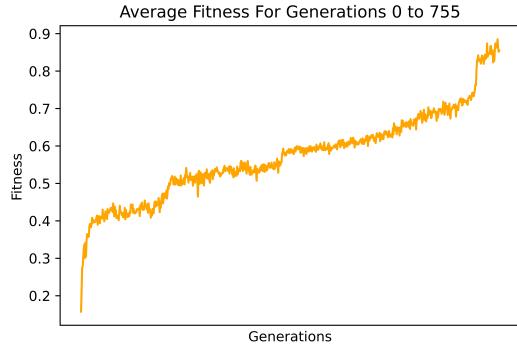
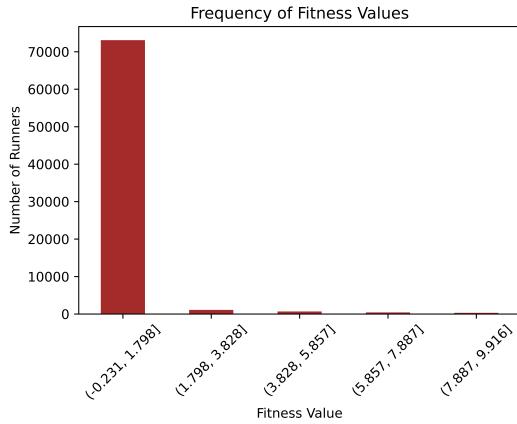


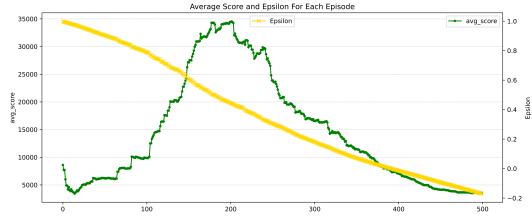
Figure 6: Frequency of fitness values



the next generations.

## DQL Approach

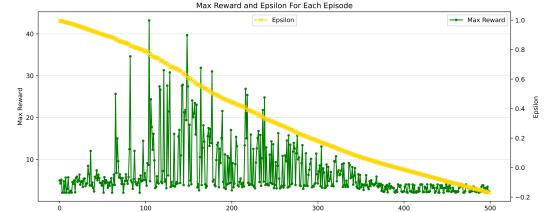
Figure 7: Average score and epsilon for each episode



Based on the graph, it can be inferred that the DQL algorithm exhibited superior performance in episodes 100 to 250, after which the average score began to decline. A closer examination of the graph reveals that the epsilon value was below 35% during the episode range where the average score started to decline. This suggests that the epsilon decay rate was high and could have directly influenced the overall development of the agent's neural network. If the epsilon decreases too rapidly, the neural network may not have sufficient time to adapt to a reduced reliance on random actions. Additionally, if the epsilon value is close to zero, the agent

may not be able to execute desired actions, which can negatively impact the agent's performance.

Figure 8: Max reward and epsilon



Based on the graph depicting the maximum reward, a similar pattern can be observed. Specifically, it can be noted that the maximum distance the agent traveled using the DQL algorithm was 40 meters, and this was achieved during episodes 100 to 250. However, after this period, the maximum distance for each episode gradually decreased due to the problem mentioned previously.

## Comparison

Both simulations were conducted over a period of approximately 8-9 hours overnight. The genetic algorithm approach exhibited a clear and consistent improvement over generations, as shown in the accompanying figure. Specifically, the simulation was able to run for a total of 755 generations, and improvements were observed in jumps followed by plateaus. This pattern indicates that once a breakthrough in behavior was discovered through some form of mutation, it quickly spread to the rest of the gene pool and plateaued until the next significant improvement occurred.

On the other hand, in the DQL approach, the development of the agent can be observed over a specific period, but the improvements are rapid and unpredictable to some extent. The simulation was able to run for a total of 500 episodes, and promising results were observed between episodes 100 and 250 when the epsilon value was sufficiently high. If we compare the algorithms based on the maximum distance achieved, the DQL algorithm exhibits an advantage over the genetic algorithm since was able to reach a 40-meter mark, whereas the genetic algorithm reached a 10-meter mark. Nonetheless, the genetic algorithm demonstrates stable improvements that could eventually lead to better results.

## Future Considerations

Some future considerations for the inputs of the neural network would be to send information regarding the velocity of the character for the genetic algorithm. This might better inform the network. However, it may not be necessary to input velocity for each individual limb and so an average velocity or velocity of just the center of mass might be sufficient. However, this could only be determined through experimentation. Another consideration is that we could have explored a niching evolutionary algorithm like the one described in the article Evolving Controllers for Virtual Creature Locomotion. Such an algorithm could maintain the genetic vari-

ation after several generations (Sanders, Lobb, and Riddle 2003).

Another idea might be to have a memory neuron which would be an output neuron whose output is connected to an input neuron. This idea was utilized in the DQL approach and could be useful for the genetic algorithm due several reasons. This could potentially allow for the network to retain some type of memory about its computational state. This may be a helpful addition as the problem being solved is highly temporal and so the ability for the network to retain some kind of information about the previous frame to the next could prove to be beneficial.

It was initially intended for the agent to walk forward instead of simply moving forward. However, the collision detection feature, which is inherent to the agents in each approach, was not implemented. This feature detects when the agent touches the ground with a limb higher than its hips. By utilizing this information, significant decrements could be made to the fitness function to encourage the agent to maintain its upper body at an acceptable height and prevent it from falling to the ground. This would potentially lead the agent learn how to balance the body, which would lead the agent a step closer to walking or at least moving forward by standing on its feet.

## References

- Peng, X. B.; Abbeel, P.; Levine, S.; and van de Panne, M. 2018. DeepMimic: Example-guided Deep Reinforcement Learning of Physics-based Character Skills. *ACM Trans. Graph.* 37(4): 143:1–143:14. ISSN 0730-0301. doi:10.1145/3197517.3201311. URL <http://doi.acm.org/10.1145/3197517.3201311>.
- Sanders, M.; Lobb, R.; and Riddle, P. 2003. Evolving controllers for virtual creature locomotion. *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia* doi:10.1145/604471.604522. URL <https://dl.acm.org/doi/10.1145/604471.604522#sec-cit>.