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.

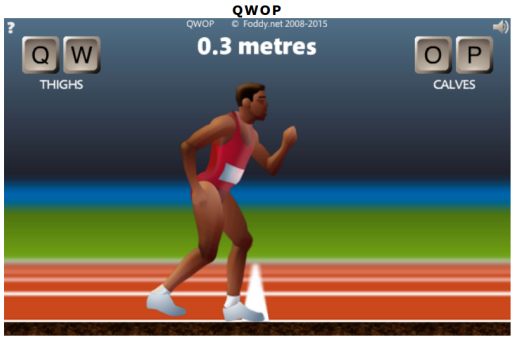
The genetic approach is inspired by natural selection where the best performers get to reproduce and spread their genetic information to their offspring. The new population then replaces the old one and plays the game again. This process is repeated until the agents play the game to a satisfactory level.

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. Two main implementations that will be considered and attempted will both be based on neural networks. The first method is a genetic algorithm approach to learning while the second approach will be a deep Q method of learning. A neural network is chosen for both methods as deep neural networks have successfully reproduced highly-detailed human movements including walking, running, cartwheel, and flipping (Peng et al. 2018).

Figure 1: The original browser-game QWOP



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 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 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 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 (EA) is a search method modelled on natural evolutionary processes. The search is formulated as an optimization problem - we define a real-valued fitness function the value of which increases with the optimality of the solution candidate. In our case, a solution candidate is a description of a particular locomotion controller and the fitness function is a measure of how far the virtual creature moves over a fixed time during a physically-based simulation"(Sanders, Lobb, and Riddle 2003).

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 neural 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 technic 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 an agent does have the information it needs to interact optimally with the environment, allowing it to exploit its knowledge makes more sense" (Maroti 2019).

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 γ (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.

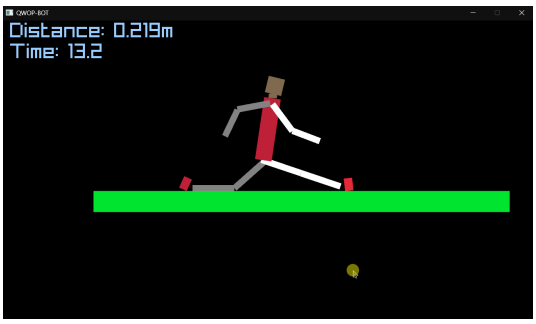
Current Progress

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.

The initial progress on the project started with a basic simulation of physics bodies where the joints that connect them were actuated by a key press. This basic simulation later evolved to create legs and then hips, and then the upper body. Various tweaks were needed to produce a correct feeling simulation based on the original game. In order to accomplish this, the ability to directly control the character with the keyboard was added even though it would not be used in the final result as the simulated agent would be controlling the character.

Figure 2: A single physics-simulated character



Once the progress on the physically-simulated character was acceptable, work begun on creating a neural network. The neural network was originally created without 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 are called weights. 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 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 weight the numerous inputs. The current implementation sends the global x and y positions of each limb 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).

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.

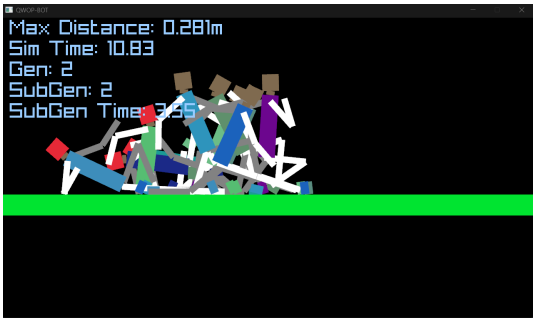
The number of characters could be increased for the sub-generation but there is a speed limitation where it would reach a point where the number of characters would slow down the simulation thus negating the added benefit of parallelism.

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

Figure 3: A simulated batch of characters during a sub-generation



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. Eventually, we deiced to move forward with 24 input nodes which would take in all the x and y positions of the limbs.

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 area 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 an 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 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 created. 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 decided to utilize the Pandas Python library to visualize our data, as we are already working with Python for our program, and we have experience using Pandas. The JSON files are imported into the visualization program where they are concatenated into a single data frame. It is not useful to visualize the network biases, so they are 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 (Figure 4), a graph of the average fitness values for each generation of runners (Figure 5), and a dual-axis graph that shows the fitness and epsilon values for each runner.

Figure 4:

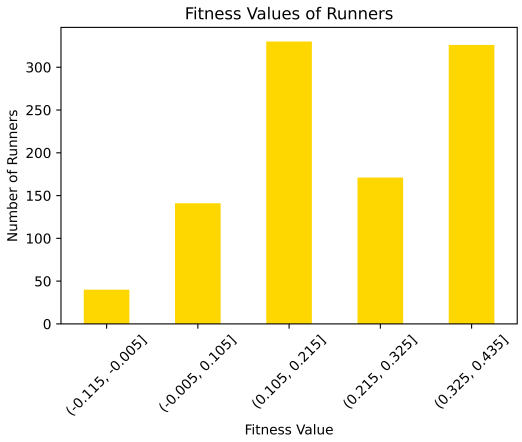
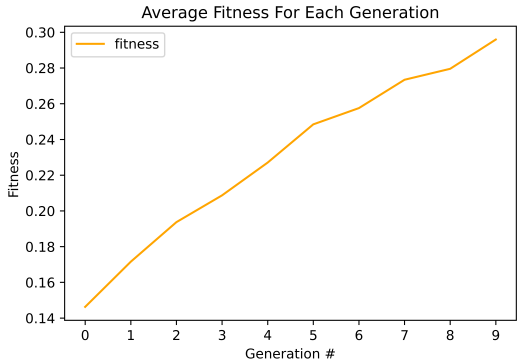


Figure 5:



Future Considerations

Some future considerations for the inputs of the neural network would be to send information regarding the velocity of the character. 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 variation 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 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.

References

Maroti, A. 2019. RBED: Reward Based Epsilon Decay URL https://arxiv.org/abs/1910.13701.

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.