QWOP Bot Progress 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 is 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 uses neural networks to approximate 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 labelled 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 be both 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.

Consideration of Backpropagation

A traditional method training a neural network is to use a method called backpropagation. Backpropagation is done by providing the network with input data and labelled output data where the term labelled refers to the desired output of the given input. The input is fed forward through the network in its current state and its output is compared to the labelled 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 children. The new population then replaces the old one, and the process is repeated until the agents play the game to a satisfactory level.

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 environment dynamics. The agent learns by interacting with the environment and collecting experience samples. 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.

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, which are estimates of the mean and variance of the gradient. Adam then adjusts the learning rate for each weight based on these moments, making it larger for weights with low variance and smaller for weights with high variance. 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 of the project and thus the approach of re-implementation 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 created 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 rather the simulated agent would be controlling the 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.

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.

Future Considerations

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