Deep QWOP Learning

Hung-Wei Wu

Submitted under the supervision of Maria Gini and James Parker to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Sciences *cum laude* in Computer Science.

12/2/2017

1 Abstract

We apply a deep learning model to an unconventional control task of keeping a ragdoll runner in a simulated environment alive as long as possible. The model is a convolutional neural network trained with Q-learning. By training the model with only raw pixel input, we show that our model is capable of successfully learning a control policy associated with playing QWOP. This model was successfully applied to a non-deterministic environment in the form of a ragdoll physics flash game.

2 Introduction

2.1 QWOP

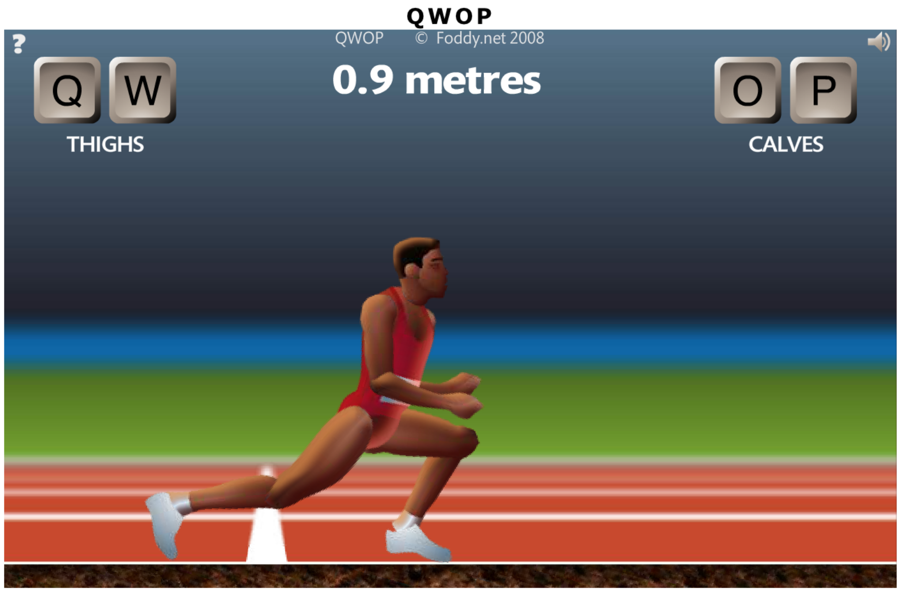

Figure 1. QWOP game play

QWOP is a free-to-play flash game created by Bennet Foddy infamous for being ridiculously frustrating to play. In QWOP, the user controls a ragdoll sprinter using the four keys: “Q”, “W”, “O”, and “P”. Each key controls the left thigh, left calf, right thigh, and right calf respectively. With the right inputs and timing, this can be used to simulate real-world human-like running. However, this is not how we, as humans are used to running. We don’t think about how specific muscles have to move in order to maintain balance and move forward. This means that in the context of QWOP, the player's collective knowledge on balance and movement is essentially useless. The goal is for the user to attempt to move the ragdoll figure 100 meters without falling over. The game is reset when any section of the upper torso touches the ground.

The game implements a ragdoll physics environment where normal physical interactions are greatly simplified. In particular, this means that any limbs that are not being directly simulated is dormant, meaning it just falls in the direction that it is already traveling. If the runner gets slightly out of balance and without the player's intervention, it will fall. The articulated figure has little to zero joint stiffness, often leading to it collapsing into comically improbable or compromising positions. The game is notoriously difficult and achieving any sort of forward movement is considered a significant achievement.

2.2 Deep Q Learning

Deep Q Learning was made famous by DeepMind in a 2013 paper "Playing Atari with Deep Reinforcement Learning" describing a deep reinforcement learning system that combines neural networks with reinforcement learning to master a diverse range of Atari 2600 games using only the raw pixels and score as inputs. Until this point, it has only been possible to create individual algorithms capable of mastering a single specific domain. Deep Q Learning represents the first demonstration of a general-purpose agent that is able to continually adapt its behavior without human intervention. However, it has only been applied to deterministic tasks, where a given action produces a given result that can be inferred from the environment. The task of playing QWOP poses a different type of problem. It is significantly more difficult due to the ragdoll physics environment. Each key press is not guaranteed to have the same results or effects on the simulation. Miniscule differences in the runner's position and momentum can often have unforeseen impacts.

3 Related Work

3.1 DeepMind Atari

Google DeepMind published a paper in 2013 describing the first deep learning model to successfully learn control policies directly from sensory input using reinforcement learning. The input is raw pixels and the output is a value function estimating future rewards. Their method was able to learn to play seven Atari 2600 games and even surpass a human expert on three of the games. These games include Pong, Breakout, Space Invaders, Seaquest, and Beam Rider. Their model is a convolutional neural network trained with a variant of Q-learning, using stochastic gradient descent to update the weights. They also implemented an experience replay mechanism which randomly samples previous actions and state transitions to smooth out the training distribution over past behaviors. This will be the primary basis of our model.

3.2 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms and techniques. This platform provides many environments that agents can interact with in a unified way. It provides an interface that allows agents to step the environment by one timestep and return new observations, rewards, and exit statuses. Examples include the "CartPole-v0" environment. In this environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track.

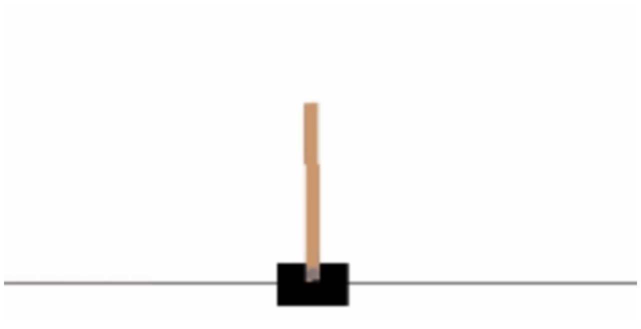

Figure 2. OpenAI CartPole environment

This system is controlled by applying a force of +1 or -1 corresponding to left and right movement to the cart. The pendulum initially starts upright and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The current episode ends when the pole is more than 15 degrees from vertical or when the cart moves more than 2.4 units from the center. CartPole is one of the simplest environments in OpenAI gym. An agent can move the cart by performing a series of actions of 0 or 1 to the cart, pushing it left or right. The QWOP game interface is written to follow a similar environment architecture where an agent has access to the reset and step methods. An example agent found in the documentation implemented a simple three-layer convolutional neural network and is trained using Q-learning. After around 500 episodes, the agent learned how to maximize the score by keeping the pole upright and the cart in the center of the environment. It is then consistently able to survive all 500 timesteps in each episode.

3.3 Stanford CS229

Gustav Brodman and Ryan Voldstad used reinforcement learning to play QWOP for their final project. Methods included discretization of state spaces with both regular and fitted value iteration using a set of reward features. Instead of using raw pixel inputs, other variables were used to better quantify the QWOP runner's state. Distance alone was not enough to determine the state of the runner; therefore, other variables such as number of feet on the ground, left and right knee angles, angle between the left and right legs, and thigh rotational velocities were used to represent the state instead. Through some experimentation, they settled on a feature mapping using the difference between thigh angles, the angles of each knee, the overall “tilt” of the runner, and the runner's horizontal speed. Evaluating their model showed fairly good results. The QWOP sprinter was able to travel around 30000 units (arbitrary distance units.) Initially, a shuffling gait was observed; however, after 10 iterations, a gait that resembled bipedal walking was observed.

4 Background

Our QWOP agent implements the Deep Q Learning algorithm using a neural net and reinforcement learning. Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given finite Markov decision process.

4.1 Markov Decision Processes

Markov decision processes provide a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision maker. At each timestep, a Markov decision process is in some state “s”, and the decision maker may choose any action "a" that is available in that particular state. The process responds at the next timestep by randomly moving into a new state "s`", and giving the decision maker a corresponding reward. We are attempting to model QWOP as a Markov decision process.

4.2 Q-Learning

In Q-learning, there is an action-value function called the Q-function, which is used to approximate the reward based on a state. It ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter. A policy is a rule that the agent follows when selecting actions. When such an action-value function is learned, the optimal policy can be constructed by simply selected the highest values in each state. We use a convolutional neural network to model the Q-function. The loss function used to train the network is shown below in Figure 1.

$$loss = \left(\underbrace{\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \max_{a`} \hat{Q}(s, a`)}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}}\right)^2$$

Figure 1. Q value and loss calculation

An agent first carries out an action "a" and observe the reward "r" and the resulting state "s`". Based on the result, we calculate the maximum target Q-value and then discount it so that the future reward is worth less than immediate reward.

4.3 Convolutional Neural Networks

This Q-function is modeled using a convolutional neural network. A regular neural network receives a single vector as input and transforms it through a series of hidden layers. made of a set of neurons. Each neuron is fully connected to all neurons in the previous layer. Neurons in a single layer function completely independent of each other. The last layer of a network is called the output layer and in classification settings, it represents the class scores. Convolutional neural networks take advantage of the fact that the input consists of images and thus it constrains the architecture in a more sensible way. In particular, unlike a regular neural network, the layers of a convolutional neural network has neurons arranged in three dimensions: width, height, and depth. The neurons in a layer will only be connected to a small region of the layer before it, instead of all the neurons in a fully-connected manner. This architecture is visualized below in Figure 2 and Figure 3.

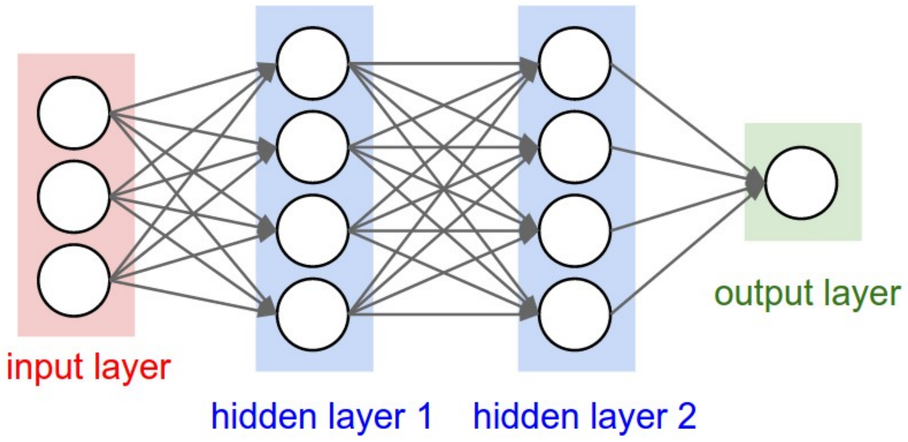

Figure 2. Regular neural network architecture

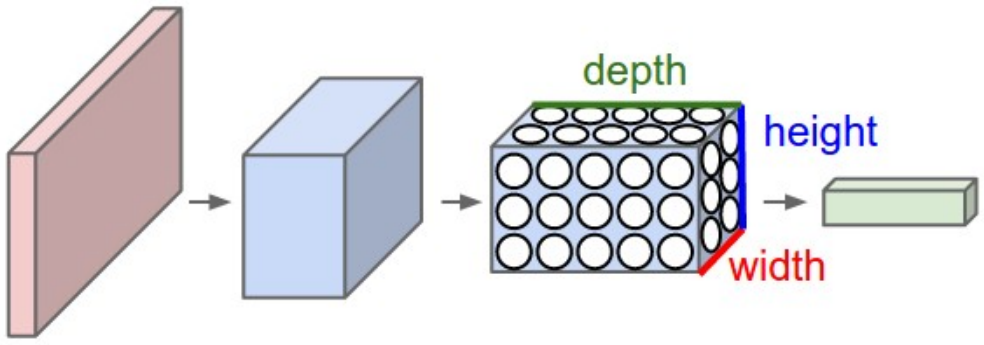

Figure 3. Convolutional neural network architecture

4.4 Remember and Replay

The most notable features of the Deep Q Learning algorithm are the "remember” and “replay” methods. One of the challenges of Deep Q Learning is that the neural network used in the algorithm tends to forget the previous experiences as it overwrites them with new experiences. Thus, methods are needed to remember previous actions and rewards and retrain the neural network to retain previous knowledge. To ensure the agent performs well long term, we need to take into account not only the immediate rewards, but also the future rewards. In order to accomplish this, a discount rate is specified. Thus, the agent will learn to maximize the discounted future reward based on the given state.

4.5 Exploration

Initially, the neural network is not trained to maximize the Q-function. Thus, the QWOP agent will randomly select possible actions a set percentage of the time. This percentage is specified by the exploration rate. It is better for the agent to try different actions and observe the subsequent rewards and start converging on the optimal action-value function. However, when the agent is not randomly deciding its actions, it will predict the reward value based on the current state and pick the action that will give the highest reward. The exploration rate will decline gradually over time.

4.6 Hyperparameters

There are also some hyperparameters that has to be specified when the model is being trained. They are listed below in Figure 4.

Episodes	The number of games the agents are going to play.
Gamma	The decay rate used to calculate the future discounted reward.
Epsilon	The percentage that the agent will randomly decide its actions.
Epsilon decay	As the network gradually learns patterns, it will explore less and less.
Learning rate	How much the network learns in each iteration.

Figure 4. Deep Q Learning hyperparameters

4.7 ReLu

The two hidden layers in the neural network used to train the Q-function are composed of rectified linear unit neurons (ReLu). The ReLu is an activation function defined as the positive part of its argument. The function is shown below in Figure 5, where x is the input to a neuron. It was first introduced in 2000 with strong biological motivations and mathematical justifications. It has been used in convolutional networks more effectively than the widely used logistic sigmoid function. However, it is worth to note that ReLu neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. In this state, no gradients flow backward through the neuron, and so the neuron becomes stuck in a perpetually inactive state and "dies".

$$f(x) = x^{+} = \max(0, x),$$

Figure 5. ReLu activation function

4.8 Schema Definition

Since there are four possible inputs into the QWOP game interface and buttons can be pressed concurrently, instead of modeling the actions as four distinct outputs, an alternative key schema is defined. There are now 16 distinct outputs, each representing a combination of four keys. This schema is defined below in Figure 6. Each row represents one of the 16 possible 4-key combinations and the 1s and 0s respectively represent if that corresponding key is pressed or released.

	Q	**W**	**O**	**P**
A	0	0	0	0
B	0	0	0	1
C	0	0	1	0
D	0	0	1	1
E	0	1	0	0
F	0	1	0	1
G	0	1	1	0
H	0	1	1	1
I	1	0	0	0
J	1	0	0	1
K	1	0	1	0

L	1	0	1	1
M	1	1	0	0
N	1	1	0	1
O	1	1	1	0
P	1	1	1	1

Figure 6. Key input schema definition

5 Methods

In order for the agent to interface with the QWOP game environment, it had to be able to simulate keyboard input as well as read the raw pixels on the screen. This was achieved by creating a virtual environment in Python for the Deep Q Learning agent to get the current state and step through actions. Another variable that was needed was the current distance that the runner has traveled. However, due to the obfuscated nature of the native JavaScript game code, we had to rely on other methods to extract the current distance. We utilized the OpenCV library to find image contours of the numbers and corresponding wrapping rectangles. The raw pixels at those locations are then screenshotted,cropped and fed into a support vector machine trained to predict its corresponding number. The Python Imaging Library (PIL) was used to take screenshots of the game and to feed it as raw input into the agent. PyAutoGUI was used to simulate keyboard input.

Initially, an environment representing the QWOP game is instantiated. Then an agent is created. For each episode, the agent will either step through predicted actions and receive a reward until it falls over and the game resets or the agent will execute all 500 timesteps. Every tenth episode, the current weight and biases in the neural networks are cached in a backup file. We limit the input to be a small section of the runner's legs in an effort to reduce the time to train the convolutional neural network. The reward is defined by how long the agent stays alive. Thus, the longer the ragdoll runner is alive, the greater the reward will be. The Q-function is incentivized to choose actions that correspond with stability.

6 Results

Through personal experimentation and game play experience, one reliable way to stay alive is to either hold no keys down or press the keys that will result in the runner with its legs spread apart as far as possible. Initial trials with 1000 episodes of 500 timesteps each yielded promising results. The hyperparameters were set as follows in Figure 7. The agent will start off by guessing 100% of its actions and every subsequent episode will decrease the guessing rate by 0.5%. For fear of overshooting, the learning rate was defined to be extremely small. However, one tradeoff was that it takes a significant amount of time for the neural network to converge on the optimal Q-function.

Episodes	1000
Gamma	0.95
Epsilon	1.0
Epsilon decay	0.995
Learning rate	.0001

Figure 7. Hyperparameters for initial trials

As more episodes were executed, the agent learned to press the same key over and over again. The key combination that found the most success was "J", which corresponds to holding the "Q" and "P" key down. This configuration allowed the runner to get in a position similar to someone doing the lunges. This position proved to be the most stable, as repeated presses of "Q" and "P" after entering the lunge position is unable make the agent fall over. Due to the low learning rate and low epsilon decay rate, each training session took upwards of eight hours. However, given the hyperparameters, the Deep Q Learning agent learned to start pressing the same keys around episode 300. Then around episode 500, the key combination pressed became "J". This is the key combination that provides the most stability.

With a working Deep Q Learning agent, we attempted to shorten the training time by increasing the learning rate and epsilon decay, meaning that the agent will guess less initially and find the global minima faster. However, it is important to note that a very small learning rate will cause the network to converge extremely slowly, and if it is too high, we risk overshooting and never finding the global minima. By changing these hyperparameters, the agent was able to learn to press the keys "Q" and "P" repeatedly by episode 200. However, since the agent is staying alive longer, this did not significantly decrease our experimentation time.

By plotting the most common key schema over episode number, a trend is easily noticed. The agent initially starts to randomly guess actions and occasionally survives over five timesteps. As the agent crosses 250 training episodes, the network learns that pressing the same keys results in a higher reward. The plot is shown below in Figure 8.

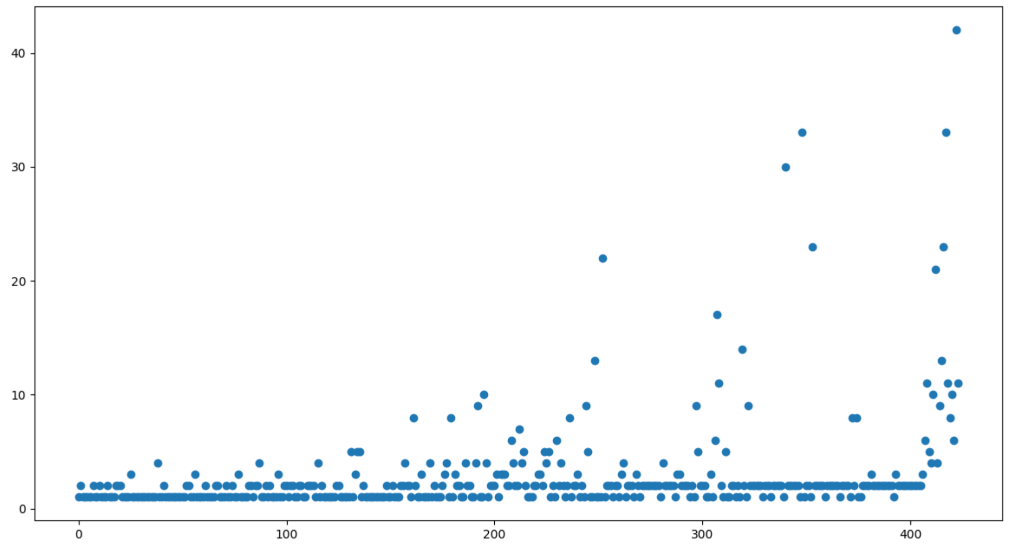

Figure 8. Most common key count every episode

By plotting the time alive over episode number, a similar trend to the one shown previously in Figure 8 can be observed. This plot is shown below in Figure 9. We can see that after 500 episodes, the agent was able to stay alive consistently through the 500 timesteps in each episode. Both plots show a slight exponential growth trend which is expected. As the network learns the correct sequence of actions to take, they are predicted more often and thus result in a higher reward, creating a positive feedback loop.

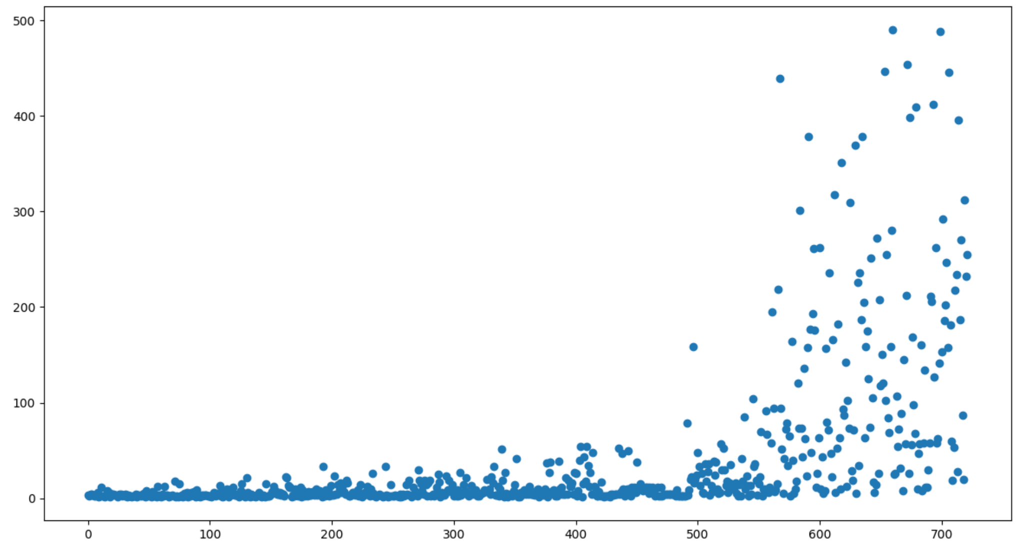

Figure 9. Time alive every episode

7 Conclusion

7.1 Summary

In this paper, we discussed applying Deep Q Learning to the nonconventional control task of keeping the QWOP runner alive as long as possible. This is in contrast to the traditional way that success is measured in QWOP. Typically, success is defined as distance traveled; however, we redefined the problem and were able to successfully apply our model. We have shown that with only raw pixel inputs, a convolutional neural network can converge to the optimal value of the Q-function. After roughly half of the expected training episodes, the agent learned to stay alive by repeatedly pressing the keys “Q” and “P”.

7.2 Future work

Work can be done to modify the current model to play the flash game QWOP as originally intended. Currently, the Deep Q Learning model is incentivized to stay alive for as long as possible. It will be interesting to modify the rewards to incentivize the agent to travel longer distances. Further work can also be done to decrease the latency of OpenCV image processing to find the contours of the distance numbers faster. Faster score detection would mean that there is less delay between consecutive key presses. This model can also be theoretically applied to more complicated environments in OpenAI Gym. Specifically, bipedal and quadrapedal walking environments.

8 References

Strehl, Li, Wiewiora, Langford, Littman. “PAC model-free reinforcement learning”, 2006.

Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, Riedmiller. “Playing Atari with Deep Reinforcement learning”, 2013.

Brockman, Cheung, Pettersson, Schneider, Schulman, Tang, Zaremba. OpenAI Gym, https://github.com/openai/gym, 2016.

Fran, Others. Keras. https://github.com/fchollet/keras, 2015.

Brodman, Voldstad. “QWOP Learning”, 2012.