TTIC 31170, Spring 2021
Final Project: Double Deep Q-Learning QWOP

David Huang

June 4, 2021

Summary: Deep Q-Learning (DQN) has been effectively applied to many deterministic environments, as demonstrated by the attainment of "superhuman" abilities on Atari games[1]. As a successor to the fundamental reinforcement learning of general Q-learning[2] for arbitrary Markov decision processes, the leveraging of a neural network allows one to avoid having to handcraft features and representations of the environment. Assuming neural networks are capable universal function approximators, the value function Q is estimated and learned via training of an agent acting upon the environment. Actions by the agent within the environment reap rewards in return, which ultimately shapes the agent's eventually trained behavior. In this project, the Double Deep Q Learning[3] (DDQN) variant of DQNs was deployed to learn the ragdoll physics-based game QWOP[4]. QWOP is a notoriously difficult game that has stoked the ire of even the most patient humans, and it appeared to be a prime opportunity for the application of DQNs given its deterministic properties and discretizable action space. With careful engineering of the environment and rewards, I was able to train an agent that played the game in full successfully.

QWOP Background: The game QWOP appeared first as a simple flash game around 2005. As a player, one must control the lower-body limbs of a bipedal sprinter from a race start to the finish line. Four keys, Q-W-O-P, are the only controls one uses to contort the ragdoll-like sprinter's thighs and calves into forward motion measured in virtual meters by the game. The only objective is to go as far as possible without causing the sprinter to fall on any part of its upper-body. The race has a finite distance of 100 meters, which is the highest attainable score within the game. With unreal physics and unintuitive controls, it is a seriously challenging game. There exists a leaderboard for world-record speed runs, and the 34rd quickest run from five months ago clocks in at over 28 minutes[5]. A screenshot of the game is shown in Figure 1.

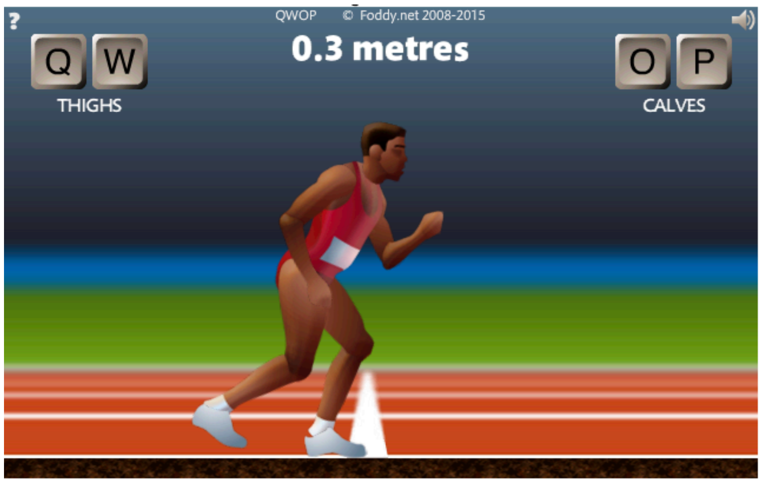


Figure 1: The game QWOP

Despite its cartoonish appearance and simple objective, there has been some prior work and study of reinforcement learning on QWOP. In a previous study[6], authors Brodman and Voldstad detailed the implementation of the value-iteration algorithm upon a model of QWOP's physics and objective. From their work, the authors showed that their agent learned a very slow movement strategy that avoided death but endowed with the ability of traversing long distances without fail. Despite the authors' success, my objective here was to implement and apply the more modern reinforcement learning algorithm DDQN to QWOP, and to test if I could incentivize the DDQN agent to learn both distance and speed.

QWOP Agent – Double Deep Q-Learning: The general objective of Q-learning is to maximize the discounted cumulative reward $R_{t_i} = \sum_{t=t_i}^{\infty} \gamma^{t-t_i} r_t$, which describes the potential future rewards from some state at time t_i. Without knowledge of state transition probabilities or rewards, an agent will seek to learn the probabilities of transitioning between states by observation of transition frequencies and corresponding rewards associated with transitions to new states (the Q function). There is not an explicit learning of a policy π, because the "optimal"

[1] https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf
[2] https://en.wikipedia.org/wiki/Q-learning
[3] https://arxiv.org/abs/1509.06461
[4] http://www.foddy.net/Athletics.html
[5] https://www.speedrun.com/qwop
[6] http://cs229.stanford.edu/proj2012/BrodmanVoldstad-QWOPLearning.pdf

action at any given state is the one that maximizes reward, i.e. $\pi^*(s) = \text{argmax}_a\, Q^*(s,a)$. The "deep" Q-learning implementation utilizes a neural network to learn and approximate $Q(s,a)$. All Q functions obey the Bellman equation: $Q(s,a) = r + \gamma Q(s',a')$, which partially describes the discounted cumulative reward R. In DQN, a single neural network is used for both learning updates and selecting actions. Empirically, this has been shown to be less than ideal with "over-optimistic" behavior and somewhat unstable as the network can converge upon non-optimal actions too quickly. In DDQN, the Bellman equation is left unmodified by is described by two Q functions: $Q(s,a) = r + \gamma Q'(s',a')$. There is no difference between Q and Q' except that Q' is frozen for some parameterized number of episodes or timesteps. This ultimately prevents the "online" Q network from converging on any particular behavior too quickly.

QWOP Environment: I describe the process and engineering of the various aspects of the environment below.

QWOP Environment – Game: QWOP is a flash game readily hosted on the creator's website. However, to have greater control on the game load-up, display, and page content, I set up a local instance of QWOP. To do this, I downloaded the game's JavaScript source code and the game's digital assets from the creator's website. Following some helpful directions of other QWOP-enthusiasts[7], I managed to get a local instance of QWOP running successfully by disabling the source code's domain check. On a local Python http server, the game runs in a Chromedriver instance controlled by *selenium*.

QWOP Environment – State: Without an API for the game, I decided that the best course of action for deriving game state information was with image processing for both the game's state and the game's live statistics. In order to obtain consistent and expected screen captures, I hardcoded placement and dimensions of the game instance's Chrome browser. Following some helpful tips on quickest screen capture methods[8], I used Python package *mss* to grab a perfectly centered screen shot of the game itself. I noticed that the game always centers the sprinter in the center of the game canvas, so I only take centered *(400 x 400)* screenshots of the *(600 x 400)* (width x height) game canvas. This centered shot adequately captures the sprinter and reduces unnecessary dead space for downstream image processing. For the DDQN, the game's screen capture is resized down further to a square *(40 x 40 x 3)* tensor.

QWOP Environment – State-Scores: I used optical character recognition to extract both the live score visible at the top of the game canvas and the final score displayed in the center (Figure 2) when the game reaches an end (sprinter dies or reaches the end). With some binary thresholding (Figure 3) on the blue channel of the screenshots, I was able to obtain some very discernable binary images of the scores. I used the Python *pytesseract* library to extract the numbers with satisfactory accuracy most of the time. The extraction of the final score sometimes provides faulty accuracy.

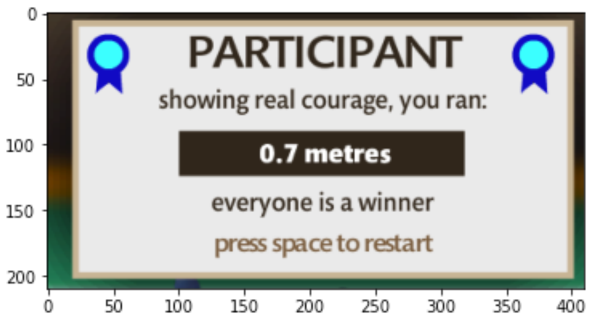


Figure 3: Final score display

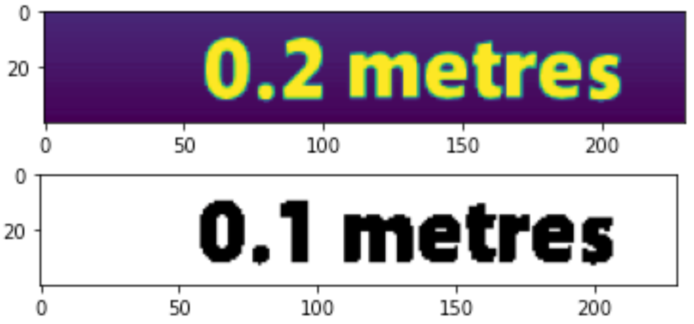


Figure 2: Binary thresholding of image text

[7] https://github.com/etopiei/QWOP-Bot
[8] https://github.com/juanto121/qwop-ai

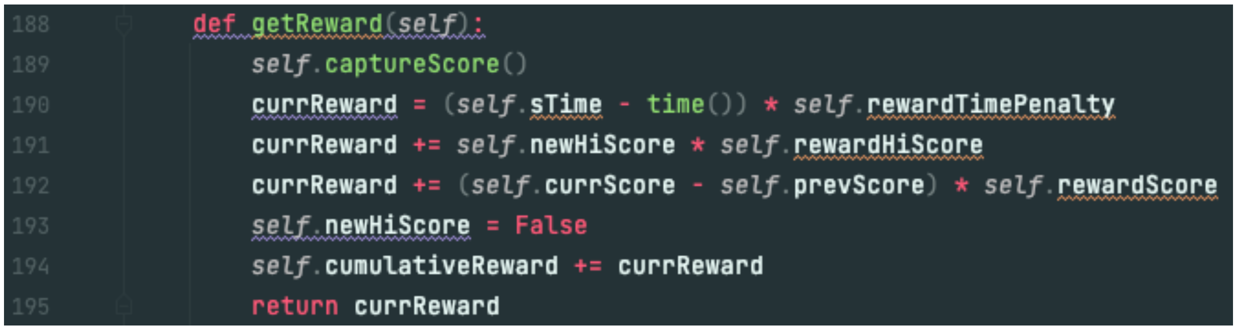
QWOP Environment – Game-Doneness: To determine when the game was over, I would take the average of the blue channel intensity of screenshot's middle 200 rows (Figure 2) to determine when a sufficient degree of "white" color appeared on screen. Luckily, this proved to be an accurate and reliable indicator of game done-ness because the game's colors are dark and contrasts well against the bright final-score display.

QWOP Environment – Action Space: The action space is limited to four discrete keys. However, it is also continuous because one can press a key for a continuous amount of time (but finite control range). I decided to discretize all four keys into two types of key presses distinguished by the duration of key-press time: short and long. Additionally, there is a no-operation action. In total, the action-space contains 9 possible actions. The actions are encoded numerically, and encoded actions are decoded into key presses of either short or long durations, which are enacted programmatically by *pynput*. Another reason for encoding two durations of keystrokes was because I could not find an adequate solution that would send multiple keystrokes simultaneously (for macOS that is). This is at odds with real-life, where one can utilize all four controls altogether at once. I implemented asynchronous keystrokes (threaded function) with varying durations to help simulate multiple overlapping keystrokes. The key stroke durations are tunable parameters.

QWOP Environment – Reward: This piece of the environment was the most impactful, and the most interesting (in my opinion) part of the entire project. With an undefined reward, I tested a handful of ideas ranging iteratively from bad to good (better, at least).

Simple Reward: A simple reward function I tested with minimal success would just be one that tracked and computed the difference between the current score against the previous score, which would reward forward motion and penalize backward motion symmetrically. I think with many magnitudes of training decent results could appear, but there was nothing that would incentivize the best (quickest) sprint possible.

Bad Reward: With Brodman and Voldstad's paper[9] and results in mind, I tried to prevent the agent from learning safe and slow actions. I decided to penalize slowness with a linear penalty of total runtime for a given episode. However, I also wanted to incentivize the agent's achievement of successful forward motion, especially if the agent would obtain a new episodic high score which would grow linearly with each new high score obtained. See Figure 4 for this reward function in code.



```
def getReward(self):
    self.captureScore()
    currReward = (self.sTime - time()) * self.rewardTimePenalty
    currReward += self.newHiScore * self.rewardHiScore
    currReward += (self.currScore - self.prevScore) * self.rewardScore
    self.newHiScore = False
    self.cumulativeReward += currReward
    return currReward
```

Figure 4: Bad reward function. Rewarded high scores and penalized an episode's total run time. The penalization of total runtime probably confused the agent's ability to learn state-reward, which does not account for time.

The bad reward function produced unintuitive cumulative rewards that did not result in any successful runs over the course of a thousand episodes. I realized that by penalizing the run time linearly, I was confusing the agent's ability to properly encode the state-reward, which was entirely independent and decoupled from any notion of time.

Good Reward: Inspired by old-school fighter games, where combo-moves would garner multiplicative cumulative rewards when "good" combinations of moves were made in quick-succession, I decided to

[9] http://cs229.stanford.edu/proj2012/BrodmanVoldstad-QWOPLearning.pdf

implement a “combo-multiplier” with exponential drop-off. The idea here was to encourage the agent to make “good” actions in succession with efficiency, which would theoretically allow it to reap a greater amount of reward for some given distance if done quickly than if it were to cover the same distance slowly. In logic, beginning with a combo multiplier of initial base value 1, one will multiplicatively grow the combo multiplier if the agent reaches a new high score; otherwise, one multiplicatively shrinks the combo multiplier until it’s no bigger than 1. The combo multiplier would effectively scale successive actions that move the sprinter along, the more quickly and efficiently, the better the reward. In pseudocode, it would look something like the following.

```
Def updateCombo(newScore):
     If newScore > currentHiScore:
          currentCombo *= comboGain    # comboGain > 1
     elif currentCombo > 1:
          currentCombo *= comboDecay   # comboDecay < 1
     else:
          currentCombo = 1             # Base combo multiplier

Def getReward():
     scoreGain = currentScore - previousScore
     currentReward = scoreGain * currentCombo
     return currentReward
```

A odd bit of behavior for the “good reward” function that I had to counter-act was the suppression of looping no-operations. In its first draft, after about 45 episodes, I encountered an agent that looped no-operations. I realized that without any penalization, the agent could settle on a non-optimal state-reward maximum of do nothing, which would neither penalize backward motion nor reward forward motion. As a fix, I decided to track no-operations and penalize no-operations, but only in the situation when there was no gain in score. I also decided to scale this penalty by the current combo multiplier because a no-op could be a valid move amidst a good succession of actions, and thus the penalty is reduced when that is the case. One can also note that backwards motion is also discouraged with greater weight when the sprinter has a combo-multiplier of value greater than one.

QWOP Results Discussion: Meaningful results of reinforcement learning tends to require many thousands to millions of training episodes. Sadly, I could not obtain adequate experimentation to share complete and good results for analysis. A good bit of time was wasted training on less than ideal reward functions, and my computer was effectively unusable during training (details on why in Limitations). However, with minimal training, on the 35th episode of training with the “good reward” function described above, the DDQN RL agent crossed the 100 meter finish line in under 38 minutes! See the final part of that run here: https://youtu.be/fMy_chcO2Ew. 38 minutes is not unreasonably far off from the world’s 34th quickest QWOP speed run of 28 minutes and 52 seconds set 5 months ago[10]. I’d like to believe that the success was attributable to the “good reward” function. The other reward functions did not yield a fully successful run with many more episodes of training. Although, this could just be luck; but the movements of the successful run seem to indicate otherwise. Like behaviors observed by Brodman and Voldstad, the agent appears to have learned that keeping the legs wide and spread helps maintain stability. I think that with more training the agent could eventually learn how to obtain better scores with successive combinations of efficient movements, making for speedier runs as a result.

Limitations and Potential Improvements: My implementation of QWOP requires the full and dedicated resources of any given computer. This is a total drawback for training scalability. The control of the keyboard requires the game instance browser window to always be the active, otherwise key inputs are sent to whichever

[10] https://www.speedrun.com/qwop

application window is active at the moment, i.e. one cannot click away from live training. This was a slight hindrance for general development workflow. This limitation was also the reason why I did not implement a function to run multiple instances at once, since keystrokes can only reach a single game instance. Finding a solution that would allow the program to send keystrokes directly to the game's identifiable running process would be a huge improvement, and one I would like to implement next. On top of directed keystrokes, finding a solution that would allow for multiple simultaneous keystrokes would be better than the current implementation of simulated overlapping keypresses.

Besides the program, the training of the agent can be very slow, simply since each episode cannot go faster than the game itself (unless the agent becomes extremely good). Training in parallel would provide a huge speed-up in training whether it with be multiple instances or multiple computers.

A tiny implementation bottleneck for the program's latency and accuracy would be the relatively slow operation of score parsing from the image. A lighter weight model than *pytesseract* could potentially be trained to exclusively parse the numbers image numbers with greater speed and efficiency, which would allow for a higher sampling rate of states for the agent, thereby reducing the lag between action and reward. Also, in consideration of images, the screen capture is very prone to interference, since any program window that covers the QWOP instance within the *(400 x 400)* capture range will introduce interference. Like directed keystrokes, a directed screen capture of the program (that is faster than *selenium*) would help safeguard this program from accidental interference.

Conclusions: A well-crafted reward function is important for getting an RL agent to learn the desired behaviors of some objective. I think that with the combo multiplier reward function, an RL agent is not only incentivized to make successful 100 meter runs in QWOP, but to also do so expeditiously. I believe that with more training, and possibly tweaking of environment settings, my implementation could achieve a world-record time (< 28m 52 s). Please consult the code README on details for running.

Final Thought: In the recent Google searches for QWOP world-records for writing this report, I discovered the recent "superhuman" level of QWOP skills achieved by another reinforcement learning implementation on 3/16/21[11] which appears to leverage both DDQN and imitation learning[12] to attain the current "superhuman" score of a run in 47.34 seconds, whereas the current human record-holder has a very impressive score of 48.33 seconds. The reward function appears to be the "simple reward function" described above but computed via very fine tracking of the sprinter's torso. Interesting.

[11] https://towardsdatascience.com/achieving-human-level-performance-in-qwop-using-reinforcement-learning-and-imitation-learning-81b0a9bbac96#

[12] https://github.com/Wesleyliao/QWOP-RL