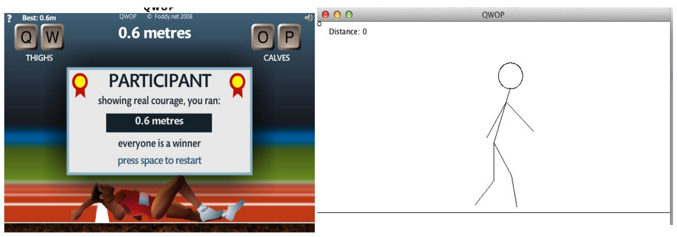
QWOP Learning

Gustav Brodman
Ryan Voldstad

1. Introduction

1.1: QWOP

QWOP is a relatively well-known game made in Adobe Flash, available for free on the Internet. It was created by the British hobbyist-game-maker Bennett Foddy. In QWOP, the user plays as a sprinter whose goal is to run a 100 meter dash. The game is notoriously difficult and beating the game has been used online as an example of something that is impossible The main difficulties in QWOP come from the physics engine and the control scheme available to the player.



Left: QWOP. Right: Our QWOP simulation

1.1.1: Control Scheme
QWOP gets its name from the controls available to the user trying to run the 100 meter dash. Instead of usual control mechanisms, the user is given direct control over four muscle groups. Q and W move the runner's left and right thighs forward, and O and P move the runner's right calves backward. With the right motions and inputs, this can be used to simulate actual running. However, this is not how we, as humans are used to running. We don't generally think about how specific muscles have to move in order to maintain balance and move forward; it just happens naturally. This means that the user's collective knowledge on balance and movement is essentially ignored. Figuring out how to mechanically move each limb is part of the difficulty.

1.1.2: Physics Engine
QWOP uses a physics system known as "ragdoll physics." This means that normal physical interactions are greatly simplified. In particular, this means that any muscle that isn't being directly stimulated is dormant, meaning it just falls in the direction that it happens to already be traveling. This is useful because it makes the system simpler, able to be simulated in an Internet browser game run by Flash, but it makes the physical world less intuitive. If the runner gets slightly out of balance, generally, he falls without the user being able to help at all. This, combined with the control system, means that the game is very unintuitive and unforgiving.

1.2: Why QWOP

On the surface, QWOP learning seems like a silly project. Though difficult, it's not a famous problem or industry staple, but it is still important. Though it's just a game, solving QWOP can help us learn about solutions to many other problems in machine learning:

- Simulating mechanical motion helps in creating and automating bipedal robots
- Having such a multidimensional action space (positions of joints, limbs, and their velocities can all be different and depend the result in different ways depending on the others)
- In general, it can help how us how various learning algorithms can help solve complex problems.

So even though the game is silly and the project seems silly, by working on the problem, we can gain a greater knowledge and understanding of complex machine learning problems, especially in the fields of motion and higher-dimensional input.

1.3: General Strategies

We want to use reinforcement learning to make an AI capable of beating QWOP, just like how we've seen reinforcement learning applied to complex physical tasks such as driving a car, flying a helicopter, or even just flipping a pancake. Simulating a running motion as an MDP is difficult, so we'll have to have ways of dealing with infinite state spaces. In particular, we'd like to use:

- Discretization of the state space with regular value iteration
- Fitted value iteration using a set of reward features.

Though these only scratch the surface of techniques available, we figured we'd try these to see how well we could apply them to a complex problem.

2. Relevance / Other Research

The task of learning to play QWOP may be seen as an optimal game-playing problem or as a parallel to a robotic motion planning problem. We would like to emphasize connections to problem in the latter category. Under this lens, the game itself can be viewed as a model for the potential real-world task of getting a robotic biped to learn a walking - or running - motion. Rather than data gathered from sensors about the positions, speeds, or acceleration of a robot's limbs, we can instead use less noisy data inferred directly from running QWOP.

In many ways, learning to play QWOP is a dramatic simplification of the task of learning bipedal walking in robotics. This task has been the subject of considerable research, as in Morimoto et al [1] and Tedrake et al1,[2].

This is often seen as a harder task, met with less success, than learning other types of robotic motion, such as walking using more legs or flight. This has to do with the inherent impracticality of bipedal motion- stability is difficult and leg motions must be finely coordinated with each other and with information about foot placement, traction, slopes, etc. Getting physical robots to learn a sprinting motion has proven especially difficult. As Tedrake et. al describes, "[a]chieving stable dynamic walking on a bipedal robot is a difficult control problem because bipeds can only control the trajectory of their center of mass through the unilateral, intermittent, uncertain force contacts with the ground."[2]

Learning to walk or run in a simulation such as QWOP simplifies many aspects of the problem. In fact, learning first against a model such as this would often be the first step in a real bipedal motion robotics problem. First, since we're not using a physical robot, it is possible to run a vastly greater number of tests and generate more training data. Each simulation can be run at speeds much faster than real time and be run unsupervised. Additionally, much of the uncertainty and complications of real-world motion planning are eliminated when using a simple model. To name a few: the effects of the actor's choices are better-understood and more reliable- the model is deterministic and the range of state spaces is dramatically reduced. The range of the actor's possible decisions is also just 4 controls, as opposed to a more realistic scenario- varying the power output on each of the motors in a physical robot. Lastly, we're modeling a much more controlled environment than in real life- a perfectly flat surface with consistent traction, with no barriers or obstacles.

3. Building a Model of QWOP

3.1: Motivation

Our original idea was to have a reinforcement learning system integrate with the actual Flash game on the website. We came to the decision, however, to program our own version of QWOP, and learn on this program rather than use the actual game. This decision was motivated by several difficulties that would arise in trying to work with the Flash program. These hinge around retrieving data from the Flash program - it's easy enough to feed keystrokes into the browser to automate the game-playing process, but retrieving feedback would be prohibitively hard. For reinforcement learning, we needed to develop models for the desirability and transition probabilities of states in order to build a reward function based on a particular state. This requires information for both modeling the current state and information the relative success of each trial for use in the reward function. For the former category, this potentially includes the positions and velocities of limbs, location of the runner, and his vertical speed, horizontal speed, and angular momentum. For the reward model, we would want to be able to build functions based on total distance traveled, time spent before falling, and the stability of the runner. We concluded that it would be pretty much impossible to read all of this data from the window of a Flash game without the source code (and we couldn't find the source code online). As a result, we implemented our own game.

3.2: The Program

Our game uses a stick-figure model of the QWOP model. In order to facilitate reinforcement learning, our game takes the form of a series of states. Each state contains the following:

- The state of the body parts
    - For each limb (modeled as lines for arms, legs, and torso, and a circle for a head), its coordinates and angle with the y-axis (for the lines)
    - The limbs' current angular velocities
    - Angles between adjacent limbs- knee angles, angle between legs at the crotch, etc.
- Time spent (number of states to reach this state)
- Center of mass
- Distance traveled

Each body part contains two points determining its start and and location, as well as a mass (point mass), and current angle to the vertical. We store the angles of the body parts (thighs, calves, and a single point mass for torso+head) so that we can keep track of limitations. For instance, the knee joint should have a range of motion of at most ~10 degrees to 180 degrees. By comparing thigh and calf orientations, we can make sure that body parts stay in locations that are at least close to anatomically possible.

If the current state is a fail state, that is, the body has fallen over, we stop the iteration. We define this as any part of the runner's body except the feet and knees touching the ground. Otherwise, we branch off. At any point in time, the user can be pressing some combination of 4 buttons, so there are 16 possible transitions for each state. If either thigh key is pressed, we add angular velocity to the corresponding thigh and some fraction of this angular velocity to the other thigh, in the opposite direction. Calf keys behave in the same fashion.

After applying these rules for each keystroke registered, we apply the following transition model to generate the next state:

-Move the body based on horizontal and vertical velocities.

-Move the limbs based on angular velocities, restricting based on location of the ground and maximum joint angles. If they try to propel themselves against the ground, move them horizontally forward. If a joint angle comes up against a limit, angular velocities are adjusted to move the angle away from this limit.

-Calculate the horizontal center of mass. If there is at least one foot on the ground, then depending on where the center of mass is located, increase or decrease rotational velocity of thighs and the torso in the appropriate direction (tilting towards the direction of imbalance). If one foot is on the ground we use the distance of this foot to center of mass in order to calculate these adjustments. If both are on the ground, then we use the the average of the two feet x-coordinates.

-Based on the updated angular velocities, translate some fraction of the angular velocities into vertical and horizontal velocities for the entire body.

-If no feet are on the ground, add to vertical velocity for gravity.

-Apply a decay constant (multiplier) on every velocity to model friction / air resistance

4: Discretized Value Iteration

Our first strategy was to do something similar to what we've done before in the homework, in the inverted pendulum problem. That is, given the position and velocity of the object at a given point in time, translate that position and velocity into one state in a set of finite states. Then we could use value iteration to determine positive and negative reward functions for each state. To implement this, we ported much of the code that we wrote for PS4 into our project in Java.

Value iteration is fairly simple. We implemented, from the notes

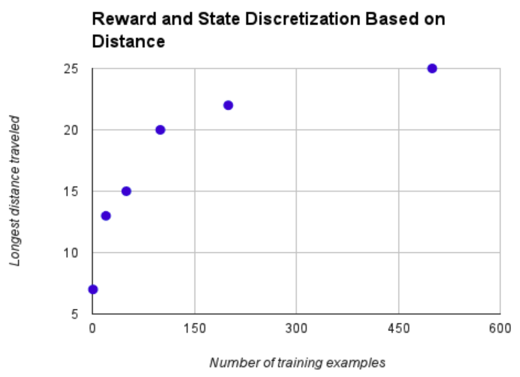
(repeat until convergence)

$$V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s') V(s').$$

where V(s) is the value for each state, R(s) is the reward function, A is the set of actions, and s' is the state achieved by taking a particular action from state s. In our case, the action is selected from one of 16 actions, each corresponding to some combination of buttons selected. Values, rewards, and probabilities observed are all similar to how they were controlled in PS4's Question 6, so we'll ignore them here. The real trick is in how we control the state discretization and how we control the reward function R(s). We tried several options for both, and here we can show what happened.

4.1: Simple State/Reward

The simplest way of controlling the state and reward of the system is to have both be controlled solely on how far the bot has traveled. This means that a higher distance will lead to a higher state and a higher reward. This was done as kind of a control group, as we didn't think that we would achieve good results at all. Here is the graph of our results, showing number of test runs versus length of maximum run:



Number of training examples versus distance of longest run.

The graph data is unsurprising. The more trials we run, the better the maximum result, but it doesn't look like the AI is learning much. The longer test runs are short enough to, in this case, be attributed to luck and random chance. In this model, we had to have the model choose random actions many times at the beginning of the simulation, and we believe that random choices alone are not enough to lead the model to achieve good results. We figured we would need a smarter way of discretizing the state space.

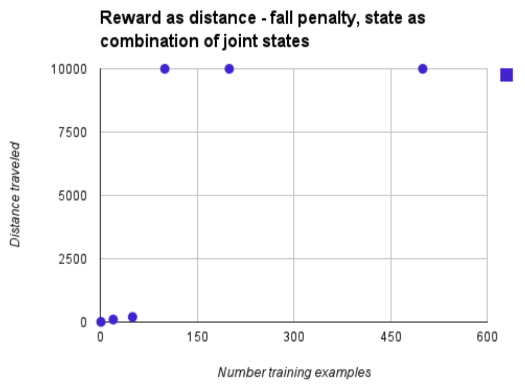
4.2: First Modulation in Discretization/Reward

The next step in our process was to figure out how to achieve a smarter discretization and reward process. We realized that distance alone isn't what we want to reward, but rather distance combined with continued distance and stability. In other words, we wanted to introduce a penalty for failure. To do this, we just have the reward function be positive for distance traveled, and negative for being in a fallen state.

State discretization was a bit more complex. Distance alone isn't enough to determine the state of the runner, so we needed some sort of measure of stability. In order to do this, we though of several things that might impact the runner's stability and ability to maintain an upright position:

- How many feet are on the ground
- Left and right knee angles
- Angle between the right and left legs
- Thigh rotational velocities (left and right)

We took these features and created a state discretization function based on the four. We tried to group like with like in order to create a viable state mapping that worked. On the next page is the graph of results, though it will require some explanation:

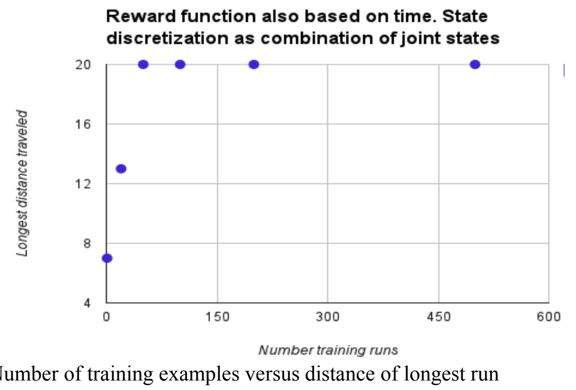


Number of training examples versus distance of longest run. Note: We cut off runs at 10,000 steps.

At first glance, the results seem too good to be true. Relatively quickly, the runner learned how to travel very far without falling. When we go past the raw numbers, though, we determine that the simulation was not behaving exactly as we would like. The runner was achieving runs of 10,000 distance, but was doing so in a very odd way. The runner, or QWOPper, would fall on to his knees at the beginning of the run, then shuffle forward on his knees constantly, legs spread apart. This created a very stable position, one capable of achieving long distances. However, it seemed almost like cheating. The purpose of QWOP learning is to create an AI that is capable of learning how to run, not an AI that is capable of achieving a stable position and inching its way forward, taking many minutes to get anywhere. We hypothesize that this is because the penalty for death and reward for stability meant that the robot would stay stable at all costs, never taking any risks to develop a better running motion.

4.3: Reward based on Time

To combat the conservative nature of the previous run, we though we'd try a different approach: to reward based on distance and stability but also divide by the amount of time necessary to reach that state. This means that the reward for a state is lessened the longer it takes to reach that state. We kept the state discretization function the same. Here again, is the graph of results, which will also need explaining:



Number of training examples versus distance of longest run

As we can see, after a certain number of learning trials, the runner simply stopped running and fell flat. In the graph presented, the distance achieved before that point is low because the time penalty was high. As we decreased the time penalty, for the most part, the runner was able to go farther before falling. We then hypothesize that by introducing a time penalty, we introduce a time cut-off, after which point the reward diminishes too much for the simulation to consider it valuable.

4.4: Overall Error Analysis:

Value iteration is a well-known algorithm. The only influence we had over it was in:

- Number of states
- State discretization function
- Reward function

As a result, we'll study those. Due to memory constraints, we were limited in the number of states we could have to around 400 or 500. This isn't a very huge number, especially when we consider that we had six variables contributing to that function, of which five could take on any number of values. This means that the contribution of each variable was diminished, because we just could not have enough states to handle them all. This is why we achieved odd results for the latter two tests. In addition, as mentioned before, since QWOP is based on ragdoll physics, small differences can become huge when the simulation is continued. A slight imbalance in one direction or another can lead to catastrophic failure a few time steps down the road. That slight imbalance, though, may not be captured by our discretization function, as it is fairly coarse.

The reward function also was inexact. Rewards based on distance alone meant that the AI would try to cheat the system, instead of learning how to run. Rewards of inverse time meant that the system would fall without becoming stable. As a result, we were never able to get a reward function we were completely satisfied with.

5. Using Fitted Value Iteration

5.1: Method

To address the shortcomings of our performance under a discretized state space, we decided to implement fitted value iteration, keeping the state space continuous. This gave us the freedom to use as many dimensions for our state space as we saw necessary. We decided to use a deterministic model for state transitions, using our (known) state transitions implemented in writing QWOP. We justified taking advantage of this extra knowledge by viewing our program under the lens of a model, rather than as the actual QWOP program. Writing our own version is akin to writing a simulation for a real-world problem; in this simulation we take advantage of our knowledge of the physics used to generate the simulation.

To get a sample of states, we simply ran the program a number of times with randomized starting angles for thighs, calves, and torso- confined so as to make the problem "solvable" in most cases (i.e that right actions should be able to prevent the model from falling and subsequently make forward progress). We then looped this (the loop of part 3 in the algorithm outline in the lecture notes) by taking the next set of states to be the successor states observed from the actions chosen for the previous set.

The algorithm is this, taken from Andrew Ng's notes[4]: (run until theta converges):

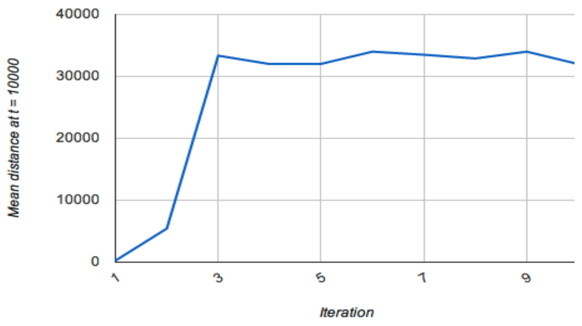
```
For $i = 1, \ldots, m$ {
    For each action $a \in A$ {
        Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).
        Set $q(a) = \frac{1}{k}\sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$
            // Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
    }
    Set $y^{(i)} = \max_a q(a)$.
        // Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.
}

Set $\theta := \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}\left(\theta^T\phi(s^{(i)}) - y^{(i)}\right)^2$
```

We used the Apache Commons Math java library to solve the ordinary least squares equation given in the last step. [3]

While using fitting value iteration helped by allowing to use a much more expressive feature mapping, it also presented the difficulty of requiring a feature mapping that could be directly translated a useful value function. Specifically, since we chose to use a linear combination of features, this constrained us to V = t(theta) * phi(s). Intuitively, this meant that we needed features that could be linearly combined to evaluate a state's likelihood of generating higher reward. Through some experimentation, we 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. For our value function, results varied similarly to those observed in the discretized model and so we used the same value function.

5.2: Results



Plotted above are the mean distances travelled by runners trained on fitted value iteration after 10000 time steps, against the iteration of FVI. Here an iteration is defined as follows: run *m* randomly initialized QWOP models for 10000 steps or until death - whichever comes first - updating theta at each time step. We used *m* = 1000, so each iteration represents 10^7 state examples. Performance converged quite quickly as a result of the large number of examples in each iteration.

Performance topped out at around 30000 (of our arbitrary distance units), compared to about 10000 for the discretized model. So there was considerable improvement over the previous model. Watching trials that used the policy after 10 iterations, we witnessed a gait closer to that of actual walking motion. However, as before the policy was very careful to avoid any imbalance- it was often more of a shuffle than a walk (and certainly not a sprint). Only very rarely did both feet come off the ground, as would be observed in an actual sprinting motion. It is worth noting however, that the "death rate" - the percentage of trials in which the model did not make it up to 10000 iterations, was considerably higher under fitted value iteration than in the discretized model. Thus this model was more inclined to a riskier strategy, which paid off in higher performance.

References:

[1] J. Morimoto, C. Atkeson, and G. Zeglin. "A Simple Reinforcement Learning Algorithm For Biped Walking," in *Proceedings of the 2004 IEEE*, pages 3030-3035, 2004.

http://www.cs.cmu.edu/~cga/walking/morimoto-icra04.pdf

[2] R. Tedrake, T. Zhang, and H. Seung. "Learning to Walk in 20 Minutes," in *Proceedings of the Fourteenth Yale Workshop on Adaptive and Learning Systems*, 2005.

http://groups.csail.mit.edu/robotics-center/public_papers/Tedrake05.pdf

[3] Apache Commons Math Java library for Ordinary Least Squares regression solving, found at http://commons.apache.org/math/

[4] Andrew Ng's CS229 Lecture Notes, found at http://cs229/materials.html