CS181 Practical4
Reinforcement Learning

Ben Cook and Scott Shaffer

April 2016

1 Technical Approach

1.1 Basic Assumptions

The general approach to solving the Swingy Monkey game is to directly approximate the *Value-Action Function* or $Q(S, a)$, a measure of the "value" of taking an action a (swing down $= 0$, swing up $= 1$) given a particular state of the game S. We implemented two separate methods of approximating Q, which are detailed in the next subsections. Here, we discuss the common assumptions made in both methods.

Through Asynchronous Dynamic Programming, we update our estimate of the Q-function in the following manner:

$$Q(S_t, a_t) \leftarrow (1 - \alpha_t)Q(S_t, a_t) + \alpha_t \left[r_t + \gamma \max_{a'} Q(S_{t+1}, a') \right].$$

This equation represents updating the current value of the Q function (given the state and action at time t) with the received reward r_t and the maximum value of the *Value-Action Function* at the state which the learner landed in at time $t + 1$. This change is weighted by a learning parameter α_t, which decays with time to allow convergence to a solution. γ represents the weighting of future over current rewards. A small value of $\gamma \approx 0$ is akin to the Q-learner expecting the game to end at any second.

The state S is, initially, comprised of multiple variables, including the monkey's position and velocity, distance to the nearest tree, etc. The state-space is essentially continuous, so it is not feasible to actually develop a Q-matrix for every particular state and action. Therefore, we must approximate Q in some way.

Our first simplification to the problem is to drop a few of the original state parameters. In particular, we know (from both playing the game and from initial test runs with the Q-learner) that both the height of the monkey (state['monkey']['top'] − state['monkey']['bot']) is fixed, as is the height of every tree gap (state['tree']['top'] − state['tree']['bot']). This allows us to drop two parameters (we choose the top of the monkey and top of the tree-gap), which will always be degenerate with other state parameters. We thus have shrunk the State-Space volume by two dimensions without losing any information.

Furthermore, we choose to ignore the current score of the game (state['score']), as presumably the optimal action to dodge the obstacles will not change with how many points have already been accumulated. This is confirmed through noting that the gameplay does not change (ex: get faster) once the player has passed many trees.

As hinted at in the assignment, the state of the game also has some inherent stochasticity, in that the gravity g changes from one game to another. We thus make the choice to choose $a = 0$ (swing down) at the first time-step of every game, which allows us to calculate an estimate of the current gravity at the subsequent time-step, by simply subtracting the current velocity from that of the initial time-step.

Our state-space S is thus made up of the following 5 parameters:

Quantity	Notation
Monkey Velocity	V_m
Monkey Position (Bottom)	B_m
Distance to Tree	D_t
Tree Gap Position (Bottom)	B_t
Gravity	g

With these commonalities, our two approaches diverge.

1.2 Linear Function Approximation

1.2.1 Q-Learning Model

One approach to solving the problem of a continuous state-space is to approximate the Q-function as a linear combination of basis functions $\{\phi\}$:

$$Q(S, a) \approx \sum_d m_d \phi_d(S, a).$$

As suggested in the assignment, one possibility is to approximate the function using a *Neural Network*, which combines the state-space parameters into sets of non-linear features. However, given our time constraints and the lack of access to pre-developed Neural Network models, we decided not to pursue this path.

Instead, we decided to use a simple *Linear Regression* model, and approximate the Q function as a linear combination of known features. We note that this approach was not guaranteed to work, as it is entirely possible that there is no simple linear combination of input features that approximates Q in such a way as to find an optimal solution.

Nonetheless, we decided to attempt a solution with this model. Using some simple domain logic, we decided to manually add *two additional features* which are non-linear combinations of the initial state features. These two features were chosen to be $(B_m - B_t)^2$ and $V_m \times (B_m - B_t)$.

The reasoning behind these two features are as follows. Using only a linear model, it would be difficult to train the Q-learner that it wants to be above the bottom of the tree gap but not too far above, such that it hits the top of the gap (ideally, $B_t < B_m < B_t + h$ at all times, where h is the constant height of the gap). By using the feature $(B_m - B_t)^2$, which is always positive, the Q-Learner can learn that it wants to make this value as close to zero as possible (if it is very large, then the monkey is either below the gap or very high above it). The second feature also allows the Q-learner to make a simple linear division between "good" and "bad": if the monkey is moving towards the bottom of the gap (good), then $V_m \times (B_m - B_t) < 0$, and if it is moving away from it (bad), then it is > 0.

The Q-learner itself was implemented using a *Stochastic Gradient Descent* (SGD) regressor from *Sci-Kit Learn*. Using SGD allows for the model to be continually trained on new data (making the Q updates), while also making predictions along the way (estimating $Q(S, a)$ for a new state and action).

The Q-learner now has several free parameters, which must be tuned to create a model that can be experimented on to find the optimal learner. These parameters are:

Free Parameter	Notation	Optimal Value
SGD Regressor Regularization	α	10^{-3}
Learning Rate Parameter	β	10^{3}
Future Weight Factor	γ	10^{-1}
SGD Learning Rate	η_0	10^{-1}

1.2.2 Learning and Exploration Model

The SGD Regressor uses both regularization (α) and its own learning rate, which we selected to be a constant (η_0) since our Q-update algorithm includes a changing learning rate. Our learning rate α_t was chosen to match the following form:

$$\alpha_t = \frac{\beta}{\beta + t},$$

which approaches $\alpha \propto t^{-1}$ after $t \gg \beta$ time-steps.

Our Learning rate parameter β is also used to transition between *exploration* and *exploitation*. Our goal is for the learner to act semi-randomly early-on, so it fully explores State-Action space, but then to cease acting randomly and act optimally to score points.

To decide an action, our model either acts randomly (here "randomly" acting involves choosing to jump up only often enough to roughly offset gravity) with probability p_t or optimally (choose $\arg\max_a Q(S_t, a)$) with probability $1 - p_t$. This "exploration" probability was chosen to be $p_t = e^{-t/\beta}$.

The combination of $\alpha_t = \frac{\beta}{\beta+t}$ and $p_t = e^{-t/\beta}$ results in three distinct phases of learning and exploration. In Stage 1 ($t \ll \beta$), the Q-learner is actively learning its environment ($\alpha_t \approx 1$) and randomly exploring State-Action space ($p_t \approx 1$). In Stage 2 ($t \approx \beta$), it is still actively learning (β is still of order 1) but it not acts optimally ($p_t \ll 1$). Finally in Stage 3 ($t \gg \beta$), the Q-learner is entirely in exploitation mode, having converged (optimally or not) to a final $Q(S, a)$ function and acting optimally according to that function.

We experimented with this model a few of times, changing the free-parameters above and recording the scores achieved. While we did not attempt to fine-tune the parameters very much (to an order of magnitude only), the "optimal" values shown above stood out clearly as doing significantly better than other combinations of parameters.

1.3 Matrix Discretization

Another approach we explored was to reduce the state space S size to something computationally tractable, both in terms of its memory storage requirement and in terms of the likelihood that a state would be visited during training, where our training occurs within the span of 200 game epochs.

The state space reduction was accomplished via the following approximation: we took the four original state space parameters (V_m, B_m, D_t, and B_t) and binned them into a significantly smaller state space ($V_m Bin$, $B_m Bin$, $D_t Bin$, $B_t Bin$). We originally chose 20 bins for each of these parameters but that still proved to be too many states to visit during training. Through trial and error, we found that using five (5) bins for each parameter enabled our Q-Learning implementation to learn. Our six dimensioned Q(S,a) table approximation took the following form:

$$[g][B_m Bin][B_t Bin][D_t Bin][V_m Bin][action]$$

where:

State/Action Variable	Number of Possible States	Description
g	2	Gravity (1,4)
$B_m Bin$	5	Binned Monkey Position (Bottom)
$B_t Bin$	5	Binned Tree Gap Position (Bottom)
$D_t Bin$	5	Binned Distance to Tree
$V_m Bin$	5	Binned Monkey Velocity
$action$	2	Action (1,0)

As with the Linear Function Approximation model, we chose a learning rate that slowly decayed to zero, defined by $\alpha_t = \frac{\beta}{\beta+t}$, which allowed sufficient time for a gradual transition from $exploration$ to $exploitation$. During our testing, we found that $\beta = 1000$ provided the best learning results. We also tested using various Discount Factor, γ, settings. However, we didn't discern much difference in learning performance as we varied this parameter, we eventually settled on $\gamma = 0.6$.

For action selection, we decided to introduce some randomness when $t < 1000$ such that this model either decides to act (swing up) with probability $prob\,[a_t = 1 \mid Q]$ given by the following softmax function:

$$prob\,[a_t = 1 \mid Q] = \frac{\exp\,[Q(S_t, a_t = 1)]}{\exp\,[Q(S_t, a_t = 0)] + \exp\,[Q(S_t, a_t = 1)]}$$

Otherwise, the optimal action as defined by $\arg\max_a Q(S_t, a)$ is selected with probability $1 - prob(Q(S_t, a_t) \mid a_t = 1))$. If $t \geq 1000$, then the optimal action as defined above is taken, since we want to focus on $exploitation$ at that point. In practice, we found the swing action, $a = 1$, was being selected too often, which we corrected by setting $prob(Q(S_t, a_t) \mid a_t = 1))/4$.

2 Results

Both methods (Linear Function Approximation and Matrix Discretization) show excellent results in learning to play the game, as shown in the figure below. Both models were able to consistently score very high in games with gravity set to $g = 4$, averaging around 40 points per game and regularly scoring in the hundreds. But they both struggled severely in learning a solution to the case of $g = 1$, averaging only at best only a handful of points.

3 Discussion

Given both models successfully learned to play the game when $g = 4$ but failed to when $g = 1$, it would have been worthwhile to focus further attention into improving the $g = 1$ case. It is clear even from playing the game manually, that the game is much more difficult to play when gravity is low. This is because each jump ($a = 1$) move takes many time-steps to be counteracted by gravity. Essentially, the player (or Q-learner) has more fine-grained control over their motion when $g = 4$ than when $g = 1$, because there is more room for error and self-correction when multiple jumps are required. Essentially, we can characterize the $g = 1$ case as a "hard-mode" of SwingyMonkey, a categorization that is justified by the difficulty our Q-learners had in succeeding at the game, given their shown success with the $g = 4$ case.

What makes this more difficult is the stochasticity in jump impulse. From studying the game code, we see that the upward velocity from jumping is drawn from a Poisson distribution ($\lambda = 15$). The positive skew of the Poisson distribution results occasionally in jumps with very large positive velocities. When

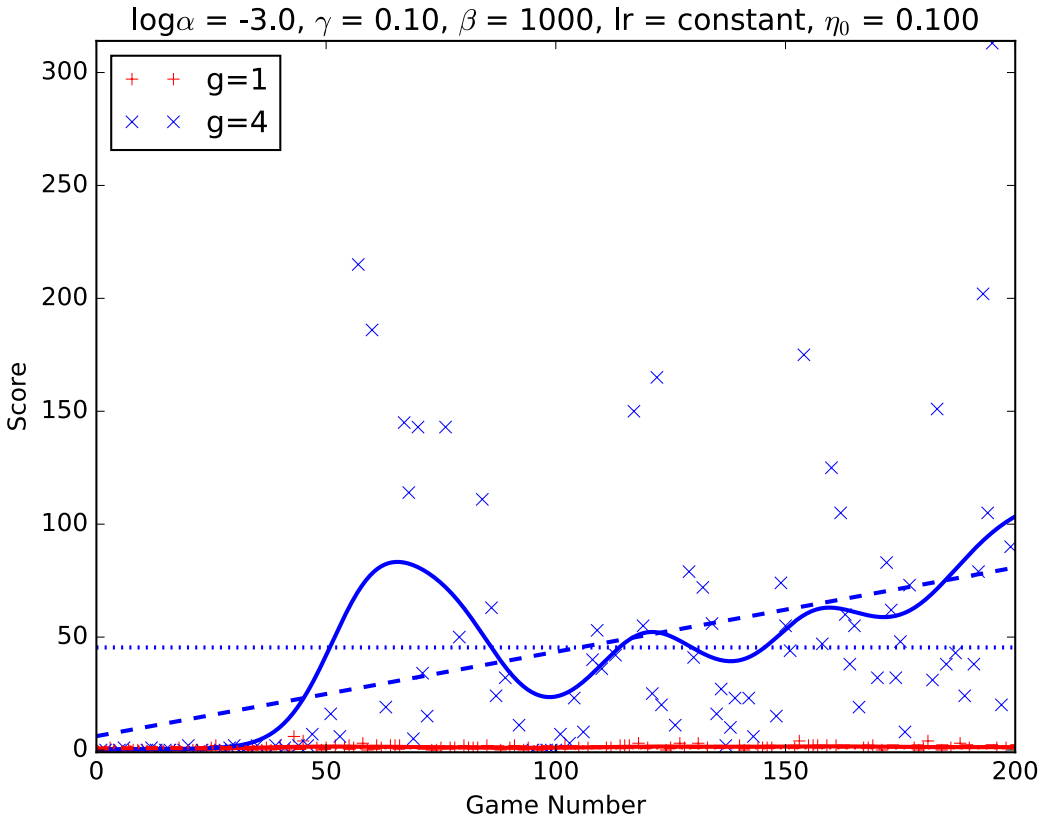

Figure 1: The results from the best run of the Linear Function Approximation model. Individual games are shown as points, and are color coded according to the gravity of that game. The dotted (dashed) lines show a mean (linear) fit to the data in each gravity class, and the solid lines are smoothed with a Gaussian Kernel. The model learns to play the game very well when $g = 4$, but fails to learn $g = 1$ gameplay. A video of this run (through the first ≈ 60 games) is available on Youtube.

$g = 4$, these are not disasterous, but when $g = 1$ these random perturbations often result in the monkey flying uncontrollably into a tree or off the screen.

Despite the difficulties in solving $g = 1$, we are please with the success of our models at learning to play and win at SwingyMonkey when $g = 4$. Of particular interest is how well our models did, given their relative simplicity. Our first model was able to find a solution which relied only on a linear combination of the data and a couple manufactured features, not a complicated Neural Network. Our second model found great success by compressing the entire volume of state-space into only 5 bins in each dimension. This shows that Reinforcement Learning is a powerful tool, and can achieve great results even with a simplistic model. Perhaps incorporating a more complex model (such as Neural Network) would result in improved success with the "hard" $g = 1$ version of the game.

4 Code and Video

Our code is available in this Dropbox folder. A video of the run shown in Figure 1 is available on Youtube.

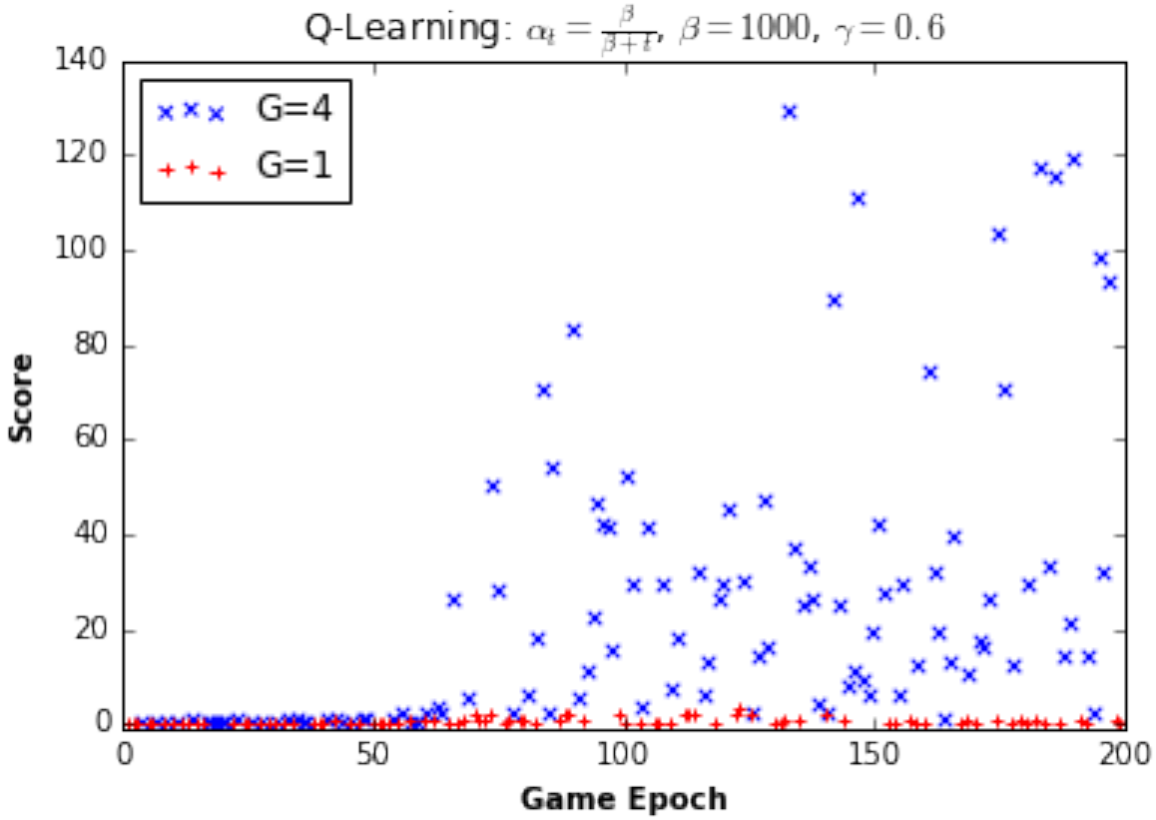

Figure 2: Q-Learning with Matrix Discretization. Plot of scores vs game epochs for a 200 game run. This plot was produced from the best run of our Q-Learning with Matrix Discretization model. Like the Linear Function Approximation model, the Matrix Discretization model was able to learn to play the game when $g = 4$ but not when $g = 1$