CS181 Practical 4: Swingy Monkey!

Victor Domene, Henrique Vaz, Joshua Meier

April 28th, 2016

1 Overview

In this practical, we performed machine learning on Swingy Monkey, a game based on the viral game Flappy Bird that took off a few years ago. In Swingy Monkey, at any point in time, there are only two possible actions: to jump, or not to jump. The game's interface allows us to have access to some kind of state of the game, such as the position of the monkey and of the trees. Thus, differently from the previous practicals, we do not have a given dataset to work with and do some kind of prediction; we simply have to teach out monkey to play the game well. Of course, in the context of games, Reinforcement Learning is a clear choice. We chose to implement a Q-Learning algorithm, with an optiomal ϵ-greedy heuristic. We spent quite a bit of time working on the ways to improve the performance of Q-Learning by tuning the several parameters involved, and by changing the features used to represent each state. Ultimately, we discovered that the monkey would learn a lot faster and reasonably well with a smaller state space. Increasing the state space would have an eventual benefit, but it would take many more epochs to achieve the same kind of results. Finally, we implemented (just for fun) a supervised way of learning: an SVM would learn from a human player, and play the game itself.

2 Feature Engineering: Handling the State Space

2.1 Choosing the Features

By default, Swingy Monkey provides us with a state that contains the following information:

1. Current score;
2. Horizontal distance from monkey to tree;
3. Top and bottom position of the tree trunks;
4. Top and bottom position of the monkey;
5. y-axis velocity of the monkey.

Also, from one run to the next, the game's gravity may change (a value that is either 1 or 4). These game states already suggest some kind of state that we could use to run Reinforcement Learning. We decided to use the following features:

1. Horizontal distance from monkey to tree;
2. Vertical distance from center of monkey to center of tree gap;
3. y-axis velocity of the monkey;
4. Gravity.

Initially, we calculated the vertical distance from the monkey to the tree gap by using the bottom of them. This changed in the parameter tuning, and more details will be given later. Also, notice that to get the gravity, we simply subtract the position of the monkey in the beginning and the position of the monkey one a bit later (after it has changed).

2.2 Binning

Even though we have only 4 features that are used, the state space is still obviously very, very large. We ran some quick scripts to find some information on our state possibilities. The horizontal distances may range from -115 to 485; the vertical distances may range from roughly -300 to 300; velocity would range from -44 to 15 (with gravity 4); and gravity, of course, was always either 1 or 4. This gives a total of roughly $43 \cdot 10^6$ possible states. Obviously, this is not only not feasible, but it will also take a very long time to learn anything at all. Because of this, we thought it would make sense to bin many of the distances in some way: after all, -100 is not that much different from -99 or -101.

It was not obvious how many bins would be appropriate, or even if we would get decent performances by doing so. We experimented with several numbers of bins. The results of this tuning are discussed later.

3 Q-Learning

Our main attempt at solving this problem involves Q-Learning. Basically, we follow the Q-Learning update rule:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t(s_t, a_t) \cdot \Big(R_{t+1} + \gamma \cdot \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)\Big)$$

This could be done easily using the callbacks defined by Swingy Monkey. Through the reward system, we could very easily produce a Q table.

3.1 ϵ-greedy

Q-Learning on itself would take care of learning, but it doesn't address the issue of exploration that well. Because we wanted our monkey to learn from a lot of different possible combination of states, we decided to use ϵ-greedy.

With ϵ-greedy, every time the monkey takes an action, he has a certain probability of doing something random. This way, the monkey will end up exploring the world more often than before. Of course, we would not want this rate to be way too high: this parameter needs a lot of tuning.

3.2 Parameter Tuning

For all of the parameter tuning, we used the "Fixed Bins" approach, described in the "Number of Bins" section. This seemed to give results earlier on, so we could do a better job at verifying the influence of each parameter.

3.2.1 Learning Rate: α

For the α learning rate, we tried two different approaches. First, we considered keeping it constant. We varied several values. Below, we show a table that summarizes the maximum scores achieved within 100 epochs, varying only α and keeping the other parameters fixed.

α	Maximum Score
0.01	8
0.1	87
0.2	123
0.5	110
1.0	91

We settled for $\alpha = 0.2$; as long as the value was high enough and not low enough, it did not seem to matter much. However, we wanted to try another approach as well, even though a constant α seemed to give reasonable results. We then tried to set

$$\alpha_{s,a,t} = \frac{1}{N_{s,a,t}}$$

where $N_{s,a,t}$ is the number of times we have been in that state and took that action in the past. This approach led to a maximum score of 131, which is actually comparable to constant $\alpha = 0.2$. We decided to, in our submission, keep these updatable α rates. This approach seems to be one of the most common in the general literature we found on Reinforcement Learning.

3.2.2 Discount Rate: γ

In order to optimize the discount rate, we decided to keep it constant and then simply vary it. The summary table was the following:

α	Maximum Score
0.01	50
0.1	107
0.2	111
0.5	123
1.0	108

In general, as long as the value was not too low, γ did not affect our results too much. We chose it to be simply 0.5.

3.2.3 Randomness: ϵ

Similarly to what we did with α, we decided that our randomness property of ϵ-greedy should change according to the number of times we have seen that state. This makes intuitive sense: if we have been many times on a state, we should not need to explore it because we already explored in the past. This is particularly true of a game with only two actions (jump or not to jump).

Because of this, we decided to use ϵ as follows:

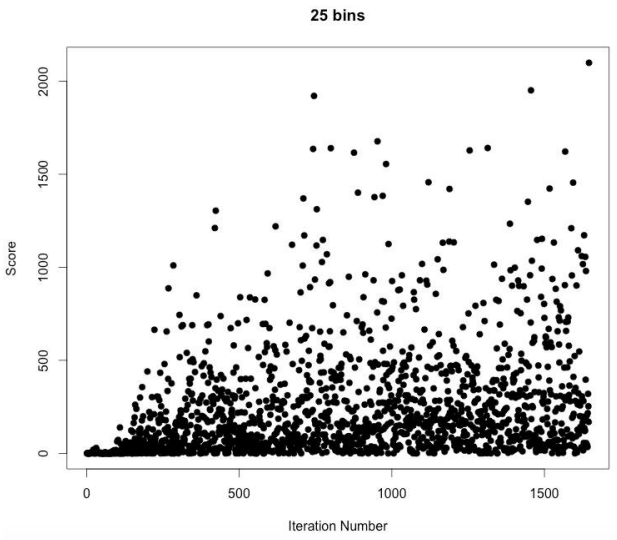
$$\epsilon_{s,a,t} = \frac{\epsilon}{N_{s,a,t}}$$

where ϵ is some fixed proportion of randomness, initialized to 0.1. Higher levels of randomness usually made it difficult for the Monkey to learn. Lower levels of randomness did not seem to affect the results too much.

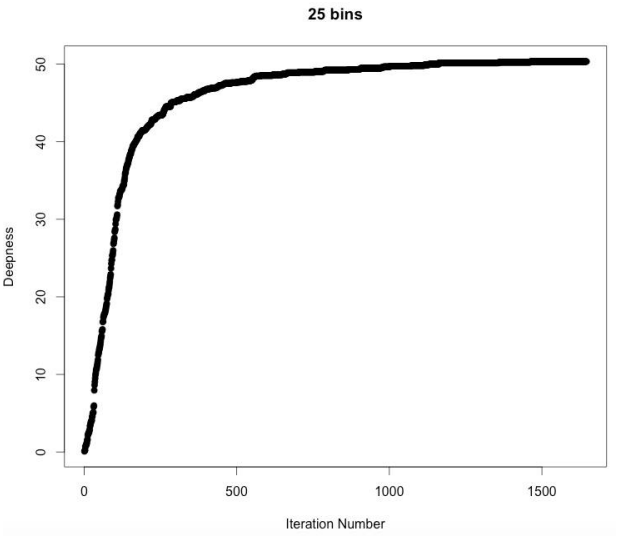
3.2.4 Number of Bins

Choosing the correct number of bins for Swingy Monkey was a quite difficult task. On the one hand, a larger number of states requires a way larger number of epochs to perform well. On the other hand, if we do not choose our bins well and we have very few bins, then the monkey will not learn well.

At first, we chose to run with (roughly) 25 bins for all of the features. In that case, we would not get significant results until over 500 epochs. However, as time passed by, we had a huge improvement: we could easily get over 100 points after 800 epochs, with a highest score of over 2000. This takes a long time to run. The following graph shows these results. The first graph shows the scores; the second one shows the percentage of the state space that had been explored until that epoch ("deepness").



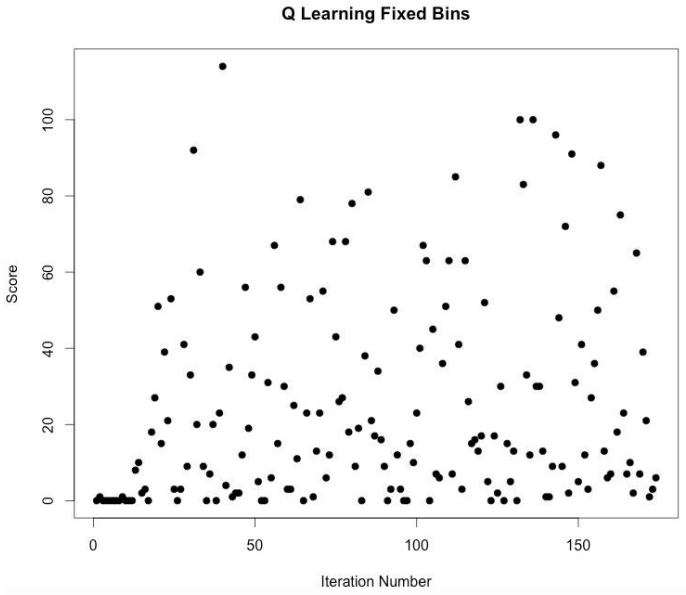
(a) Scatterplot of Scores



(b) Deepness

We were satisfied with these results. However, we decided to try a different number of bins. With (roughly) 50 bins for horizontal/vertical distances, the learning process was a lot slower, and we ended up not having enough time to produce significant graphs. We hypothesize that we would need to run many more epochs to achieve similar results.

Finally, we thought about reducing the state space. We decided to simply use 24 states. We binned horizontal and vertical distances in two, while binning velocity in three bins. Gravity, of course, was left at 2 bins. In this setup, we learned that the vertical distance should really be between the center of the monkey and the center of the gap between the trees. Using this feature, we got very exciting results. The following graph shows the scatterplot of the scores for the first epochs. If we compare it to the first epochs of the higher-bin graphs, we see that the Monkey learned much more quickly. It very easily reaches 100% deepness.



4 Gravity: A Special Case

We noticed that the game was a lot harder to learn when gravity was 1. Given enough epochs, it will be able to do around 50 points. However, the best results of around 1000 or more, were always achieved when gravity was 4. We were unable to explore why this was the case.

5 Learning From Humans: Using Classifiers

Besides implementing the usual and intuitive Q Learning Method for Swingy Monkey, we also decided to look for alternative methods that could possibly perform as better or even overperform Q Learning. We found a paper named "Obstacles Avoidance with Machine Learning Control Methods in Flappy Bird's Setting", which provided a variety of more "deterministic" techniques. We call them "deterministic" because they do not involve any kind of self-training. In particular, they do not even require us to implement a reward function. Notice that in this setting, we kept the gravity constant; we could have just as easily added it to the feature space, but we chose not to.

The most important idea behind this technique is the following: for each instant m, we will compute an array v_m of features particular to that moment (which is the same as described above). Then, we will play the game ourselves for as long as possible and record all of our actions in a file in which each line has the format

$$[\Delta x, \Delta y, V];\ \text{action}$$

where *action* is 1 if the monkey did jump at that location and 0 otherwise, and Δx, Δy and V are our features.

After recording these actions, we train a classifier on this points that will help us predict what should be done at new states during the game. Hence, for each instance in the game, we use the classifier to predict if the corresponding array v_m for that moment requires a jump or not. Simply based on that, we either jump or not.

5.1 Classification Methods

We tried three different kinds of variations for this method. The first possible variation was to use either an SVM of a Random Forest to make the classification. The second possible variation was to include some basis functions on v_m. We tried 6 dimensional arrays of the form

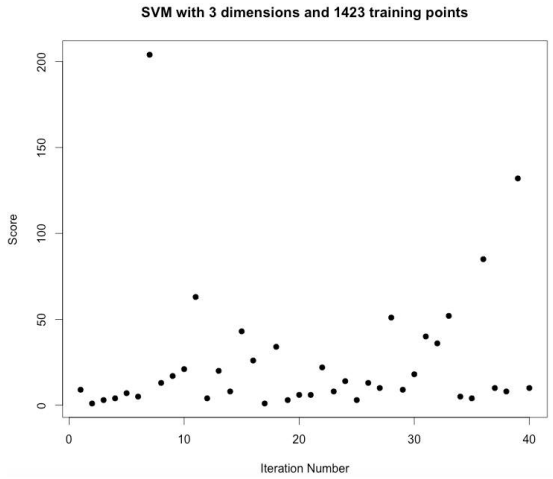
$$v_m = [\Delta x, \Delta y, V, (\Delta x)^2, (\Delta y)^2, V^2]$$

as well as 9 dimensional arrays of the form

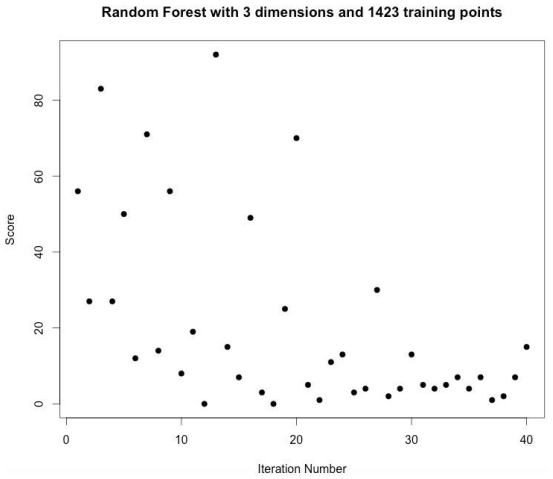
$$v_m = [\Delta x, \Delta y, V, (\Delta x)^2, (\Delta y)^2, V^2, (\Delta x)^3, (\Delta y)^3, V^3]$$

Finally, the third variation was the number of data points we included in our training set. We tried with as few as 30 data points and as many as 2000.

In terms of running time, it is important to say that the SVM is much slower than the Random Forest, the reason why we were not able to test arrays of more than 3 dimensions with the SVM. At the same time, we observed that even using only 3 dimensions, the SVM performed way better than the Random Forest.



(a) SVM



(b) Random Forest

Figure 2: Scatterplot of the score achieved by the SVM and Random Forest. We used the SVM to fit the data from a human playing and achieving a score of 50. Notice that the SVM can often perform much better than the human. Obviously, the SVM would perform even better if we had used the data of a human achieving a better score.

6 Code

Our code for this practical can be found in this GitHub Repository. There is a README file with some setup instructions.