Reinforcement Learning with Swingy Monkey

Kevin Eskici*, Luis A. Perez†, Aidi Zhang‡
Harvard University

Abstract

This paper explores model-free, model-based, and mixture models for reinforcement learning under the setting of a Swingy-Monkey game [1]. SwingyMonkey is a simple game with well-defined goals and mechanisms, with a relatively small state-space. Using Bayesian Optimization [2] on a simple Q-Learning algorithm, we were able to obtain high scores within just a few training epochs. However, the system failed to scale well after continued training, and optimization over hundreds of iterations proved too time-consuming to be effective. After manually exploring multiple approaches, the best results were achieved using a mixture of ϵ-greedy Q-Learning with a stable learning rate,α, and $\delta \approx 1$ discount factor. Despite the theoretical limitations of this approach, the settings, resulted in maximum scores of over 5000 points with an average score of $\bar{x} \approx 684$ (averaged over the final 100 testing epochs, median of $\bar{m} = 357.5$). The results show an continuing linear log-relation capping only after 20,000 training epochs.

Introduction

Recent advancements in reinforcement learning have led to an increase in activity in the field. In particular, three main approaches have surfaced [2].

Overview of Reinforcement Learning

The first approach can be thought of model-based reinforcement learning in which the agent, in our particular case termed Monkey, attempts to learn a model of the world. Learning a model is as simple as learning the transition probabilities for each (s, a, s') triplet in the world, where $s, s' \in \mathcal{S}$ are predefined states (labeled P). We also have that action a at state s leads to s' with some non-zero probability. The essence of the model based-approaches is to learn these probabilities. In addition, the model based approach also learns a model of the rewards received at each state-action tuple, R. Once the model (P, R) is known, training occurs in the form of finding the optimal policy, π^*, which is typically accomplished using either Policy Iteration or Value Iteration as described by Avi Pfeffer [1]. Typically, the agent is allowed a phase or episode of a certain number of epochs during which he uses the previous optimal policy π_i^* to make decisions. At the end of each episode, the agent now has a better model of the world and performs an optimization on this new model, leading to

$$\pi_{i+1}^* \leftarrow \text{OPT}\{(P, R)_i\}$$

where $(P, R)_i$ is the currently learned model.

A second approach is the well-known and often-used Q-Learning Algorithm. This is a model-free learning mechanism which only implicitly learns transition probabilities through the maximization of the $Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ function. The objective is to take the rewards obtained from different sequences of actions in particular states as samples in approximating the true total value of a state-action tuple (s, a), given by $Q(s, a)$. The approach is theoretical grounded in the idea of stochastic gradient ascent, and typically involves a learning rate parameter $\alpha > 0$ which can be tweaked, as we do in this paper, to improve the rate of convergence onto an optimal policy (convergence is not always guaranteed). The simplicity with which Q-Learning can be implemented is surprising, as it is the method which led to some of the best results after optimizing the parameters.

A third approach utilizes a mixture of model-free and model-based learning. While the model-based approach learns a model of the world and then solves for an optimal action policy π^*, a method known as Temporal Difference Learning does away with finding the optimal policy. It instead collects information on the world, developing a model P of transition probabilities on state-action-state triplets and a model R of the rewards for each state-action tuple. It then uses an approach where it attempts to estimate the value function $V(s)$ for each $s \in \mathcal{S}$. This leads to faster propagation of negative rewards, and typically faster learning. Once the value function has been defined, the action of choosing the optimal policy involves a simple expectation maximization procedure.
Lastly, in this paper we also introduce the topic of parameter optimization through Bayesian optimization, as discussed by Snok, Larochelle, and Adams [3]. TD Learning and Q Learning were both heavily optimized, and model based learning only slightly so. While the theoretical properties of convergence of both model-based and TD learning would have led us to belief in them performing better , we actually found that Q-Learning, along with Bayesian Optimization of the parameters, led to both an incredibly short training epoch (under 10 epochs) as well as an impressively robust, long-term model.

*keskici@alumni.harvard.edu
†luisperez@alumni.harvard.edu
‡aidizhang@alumni.harvard.edu
Computer Science 181: Machine Learning
Practical 2, Harvard University
[1]The code is hosted on a public repository here under the prac4 directory.
[2]The optimization took place using the open-source software made available by HIPS here.

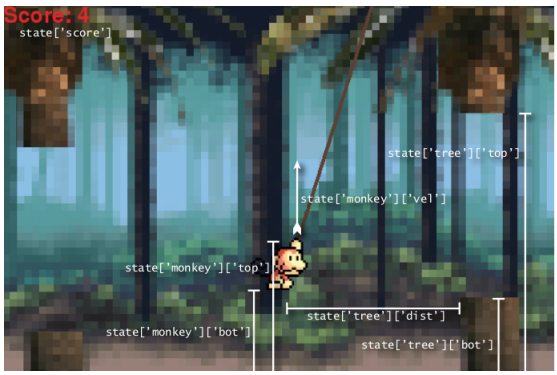


Figure 1: The SwingyMonkey game windows, with different state parameters labeled.

SwingyMonkey

SwingyMokey is a simple game, similar in kind to the popular, now defunct, FlappyBird [3]. The mechanics of the game are simple, and are discussed in more detail in Section . The purpose of the game is to cross have the Monkey cross through as many tree "gaps" as possible by "swinging" from branch to branch.

In this paper, we explore described reinforcement learning approaches and provide both a theoretical foundations as well as experiment results. The code for the simulations is open-source and available on gitHub (git@github.com:kandluis/machine-learning.git). Where deemed relevant, we present snippets from the code.

Methods

In this section, we introduce a thorough representation of the game dynamics as well as in-depth theoretical foundations for the RL methods utilized. Additionally, we discuss the parameters for each method and their expected effect on training. We finalize the section with a short discussion on parameter optimization through the use of the Spearmint software package.

SwingyMonkey Mechanics

We now introduce the mechanics of the SwingyMonkey game, as well as a description of parameters used throughout the models described in subsequent sections.

In Figure 1, we have the SwingyMokey game window. Looking at the image, we see that we actually have access to a total of seven variables for use in the description of the state space. The window size used for the monkey were 600×400[4].

Reward definitions are as follows:

- Reward of +1 for passing a tree trunk
- Reward of -5 for hitting a tree trunk
- Reward of -10 falling out of the game screen

At this point, it's of interest to note that the above rewards are well geared for negative reward propagation in Q-Learning. Q-Learning tends to suffer from the inability to propagate negative rewards quickly (due to the maximization aspect of the propagation, which tends to favor positive values).

[3]Information on FlappyBird an online version of the game can be found here.

[4]Other parameters utilized are defined in **parameters.py**.

The first challenged in the training of the monkey the bucketing of the above state space in order to facilitate learning. While it is possible to train on a large state space, faster learning can occur with a smaller set of feasible states. Additionally, some states are redundant. It's not necessary to know the absolute location of the monkey or the trees. Given that the tree itself never moves off screen in the vertical direction, simply having the distance from the monkey to the tree vertically and horizontally should be enough. We therefore define our states $s \in \mathcal{S}$ as $s = (h, w, v)$ (triplets) of three values. Let d_v be the vertical distance in pixels of the the monkey from the tree, d_h be the horizontal distance, and v_m be the reported vertical velocity of the monkey in pixels per second. The values are given by:

$$h = \text{round}(d_h / p_h) \tag{1}$$
$$w = \text{round}(d_w / p_w) \tag{2}$$
$$v = \text{round}(v_m / p_v) \tag{3}$$

where $p = (p_h, p_w, p_v)$ is our current parameter vector. Intuitively, the value $p_w \times p_h$ defines a discretization of the screen into buckets of that dimension. The parameter p_v does the same thing for the possible values of the velocity. With this definition, not only have we reduced the state space, we have introduced just three parameters which allow use to tweak the total size of the state-space. Naively, we would expect smaller state-spaces to learn more quickly but perform more poorly than larger state spaces.

Given the above definition of our parameterized state space, from now on referred to as $\mathcal{S}$ for succinctness, we now focus on defining the action space, $\mathcal{A}$. In the case of SwingyMokey, the action space is relatively simple, with only two possible actions in every state, "jump" and "no jump". For simplicity, we will define this action space as $\mathcal{A} = \{0, 1\}$ where 0 corresponds to "no jump" and 1 to "jump".

The goal of the monkey is to receive as high a reward as possible, which is directly correlated with how many trees is can successfully pass without dying. The monkey is always attempting to maximize his reward, with the exception of a few of our implementation which include an ϵ-greedy approach.

For the above reasons, we chose to consider this problem in the general case. We define then our reward function in terms of the total reward that the monkey obtains after T ticks of the clock (which is how often Monkey reports his state and reward).

$$R_{total} = \lim_{T \to \infty} \sum_{t=1}^{T} R(s_t, a_t) \tag{4}$$

With the above in mind, we have generalized our problem to a infinite horizon reinforcement learning problem.

Models

Model-based Learning As discussed in by Pfeffer [1], RL can be thought of as a Markov decision process. We can therefore attempt to learn by first creating a model for the world. We label our approximation for the model at episode i as $\mathcal{M}_i = (P_i, R_i)$ where $P_i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ and $R_i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ are our summaries of our collected statistics (the transition probabilities and the rewards). This information can easily be collected, and we do so, during the each action of

the monkey takes [5]. We keep a running total of:

$$R_{\text{total}} : \mathcal{S} \times \mathcal{A} \to \mathbb{R} \tag{5}$$
$$SA_{\text{total}} : \mathcal{S} \times \mathbb{A} \to \mathbb{N} \tag{6}$$
$$SAS_{\text{total}} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{N} \tag{7}$$

The purpose of Function 5 is to simply keep a running total of the rewards received for each state-action tuple. Similarly, Function 6 keeps a running count of the number of times state $s \in \mathcal{S}$ has been visited and action $a \in \mathcal{A}$ has taken place. Lastly, Function 7 keeps count of the number of times that state $s' \in \mathcal{S}$ has been reached after taking action a from state s.

With these parameters, it becomes possible to reconstruct the model [6]. The parameters can be estimated using the formulas:

$$\mathbb{E}[R(s,a)] = \frac{R_{\text{total}}(s,a)}{SA_{\text{total}}(s,a)} \tag{8}$$
$$P(s,a,s') = \frac{SAS_{\text{total}}(s,a,s')}{SA_{\text{total}}(s,a)} \tag{9}$$

We now have a way to approximate our transition model, $\mathcal{M}$. Therefore, after each epoch, our implementation of model-based learning [7] simply needs to calculate the optimal policy, $\pi_i^* : \mathcal{S} \to \mathcal{A}$. This can be done using either Value Iteration or Policy Iteration. For factors of simplicity, our implementation make use of Value Iteration to approximate the best policy. Our initial policy is defined as $\forall s \in \mathcal{S}$:

$$\pi_0^*(s) = 0 \tag{10}$$

In addition to the paramers defined previously, model based learning also takes as input a discount rate, γ.

Model based learning is theoretically able to learn faster. However, it runs into issues of over-exploitation rather than exploration. Our simple implementation does not address these issues directly, though it's possible to do so using something like ϵ-greedy learning or Boltzmann temperature learning.

The results of our implementation can be found in the file **modelbased.py**. The model can be used using the command:

```
@: python run.py --live-train=50 ModelBased
```

TD-Learning In addition to a model-based learner, we also implemented a mixed approach termed Temporal Difference Learning [8]. The theory behind temporal difference learning is that rather than calculating an optimal policy every episode, as is done in model-based learning, requiring the use of either Policy Iteration or Value Iteration, it is possible to instead sample the rewards and perform stochastic gradient ascent on the value function, $V : \mathcal{S} \to \mathbb{R}$. For each $s \in \mathcal{S}$, we have that $V(s)$ is the expected value of the state. If we can approximate the value function, then at each step, we can choose the optimal policy:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \left[R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s,a,s') V(s') \right] \tag{11}$$

Intuitively, Equation 11 simply attempts to pick the action a which maximizes our expected reward. The reward we receive now, plus the expected reward from whatever new state s' we transition into. The action is then the optimal. How does one approximate the function V. Note that by the above, we can actually consider $V(s)$ to be a random variable, and borrowing ideas introduced in the Q-Learning Section, we can generate the following update rule for the value on visiting state s' and receiving reward r:

$$V(s) = V(s) + \alpha \left[(r + \gamma V(s')) - V(s) \right] \tag{12}$$

where we can note that $r + \gamma V(s')$ is the sample for $V(s)$, and α is a new parameter, the learning parameter, which we use to control the importance of new observations.

The advantages of TD-Learning over Model-Based learning lies in the simplicity of implementation of TD-Learning (you don't need to implement either Policy Iteration or Value Iteration). The disadvantage, of course, is that the model might not learn quickly enough. Due to the α factor, there exists some level of exploration that occurs. However, the risk of exploitation is still present, so introducing an ϵ-greedy approach is still recommended with this approach.

The results of our implementation can be found in two files, **tdlearner.py** and **tdlearnerbayes.py**. The models can be run one after the other using:

```
@: python run.py [opts] TDLearner TDLearnerBayes
```

Q-Learning Q-Learning is a model free approach to reinforcement learning. In fact, this is the approach we found the most successful for our problem, and is therefore the one for which we made the most optimization and attempted the most number of iterations. We begin our discussion with the theory behind Q-Learning. Reinforcement learning can be thought of as simply attempting to solve for the function $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Once this function has been obtained, the task of calculating the optimal actions is simple:

$$\pi^*(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a) \tag{13}$$

Q-learning takes the approach of stochastic gradient ascent, which makes sense in the problem setting given that the rewards received can be taken to be samples from a distribution. Recalling the definition of the Q function for the infinite horizon case with γ as the discount factor, we have:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a') \tag{14}$$
$$= R(s,a) + \gamma \mathbb{E}_{s'} \left[\max_{a \in \mathcal{A}} Q(s',a') \right] \tag{15}$$
$$= \mathbb{E}_{s'} \left[R(s,a) + \gamma \max_{a \in \mathcal{A}} Q(s',a') \right] \tag{16}$$

where we have defined $P(s' \mid s,a)$ as the probability of transitioning into state $s' \in \mathcal{S}$ given that we take action $a \in \mathcal{A}$ from state $s \in \mathcal{S}$. If we take the above look at Q, an approximation can be done relatively simply. On each transition to a new state $s' \in \mathcal{S}$, we can simply update our value for $Q(s,a)$ for our previous state s and our previous action a given our new observation of the reward, $r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[(r + \max_{a' \in \mathcal{A}} Q(s',a')) - Q(s,a) \right] \tag{17}$$

[5] See **tdlearner.py** (from whom **modelbased.py** inherits).

[6] See **tdlearner.py** for more information.

[7] The implementation of the model based learning class can be found in **modelbased.py**.

[8] A good explanation of TD-Learning can be found at Wikipedia.

The intuition for this is that we're updating our current estimate of $Q(s, a)$ by our observed sample of the value of $Q(s, a)$. We have now extended our set of possible parameters further with the following:

$$p = p \circ [\gamma, \alpha] \tag{18}$$

This simple implementation of Q-Learning can be found in the file **qlearner.py**.
Something of particular interest, however, is the learning rate α used by our model. In particular, the code in **qlearner.py** contains a definition of alpha as follows:

```
lambda i : alpha
```

The idea behind this anonymous function is that we can actually make the learning rate depend on some counter value i. In fact, looking at at the file **qlearner2.py**, we find a slightly more complex implementation of Q-Learning which sets the discount factor, γ, at the Bayesian optimized value [9], and replaces then further replace the learning rate α with two new parameters, x_0 and k. The two parameters are in fact intended to model an inverted logistic function with the following formula:

$$\text{lambda}(i) = \frac{1}{1 + e^{k(i - x_0)}} \tag{19}$$

While this does not satisfy the theoretical properties guaranteeing convergence of Q-Learning [2], the model does work relatively well once it is optimized. Note that the input parameter i is simply:

$$i = SA_{\text{total}}(s, a)$$

where (s, a) is the state-action tuple that our Equation 17 is updating.

However, if we wish to decay our learning rate so that the following properties hold:

$$\sum_{i=1}^{\infty} \alpha_i = \infty \tag{20}$$

$$\sum_{i=1}^{\infty} \alpha_i^2 < \infty \tag{21}$$

we take a modified harmonic simple harmonic function. That is, we have:

$$\text{lambda}(i) = \frac{p_\alpha}{i} \tag{22}$$

where we take the above value $\iff$ lambda$(i) \leq 1$. We take p_α as an additional parameter.

This is in fact implemented within the file **qlearer3.py**. The default parameters have been optimized (only briefly) using the method explained in Section .

Furthermore, in **qlearner3.py** we also introduce the idea of ϵ-greedy learning. In order to balance exploration with exploitation, our learner also utilizes a scaled function similar to that defined in Equation 22 with parameter p_ϵ. Then, at each decision-making step, the algorithm will, with probability ϵ, take a random action. Note that ϵ scales with the number of decisions that the monkey has made.

[9] See Section for more information

The above learner had good results, as is discussed in Section . However, we went a step further and modified the learning so that a different decay function and ϵ-exploration function could be used. The code for this final iteration of our learner can be found in **qlearner4.py**.

All of the above Learner can be run relatively simply with:

```
@: python run.py [opts] QLearnerX
```

for $X \in \{1, 2, 3, 4\}$.

Optimizing the Parameters

One particular problem with an ever expanding set of model parameters p is selecting the correct value for each parameter. It's not immediately clear what the values should be, though we can typically bound them by both their type $\tau : \mathcal{P} \rightarrow \{\textbf{INT}, \textbf{FLOAT}\}$ and their ranges. For example, we have the following restrictions on the parameters common to all of our models, p_h, p_w, p_v:

$$\tau(p_h) = \tau(p_w) = \tau(p_v) = \textbf{INT} \tag{23}$$

$$1 \leq p_h \leq 400 \tag{24}$$

$$1 \leq p_w \leq 600 \tag{25}$$

$$1 \leq p_v \leq \tilde{}2000 \tag{26}$$

We can define similar bounds for other parameters. For example, we have the following:

$$\tau(\alpha) = \tau(\epsilon) = \tau(\gamma) = \textbf{FLOAT} \tag{27}$$

$$0 \leq \alpha \leq 1 \tag{28}$$

$$0 \leq \gamma \leq 1 \tag{29}$$

$$0 \leq \epsilon \leq 1 \tag{30}$$

Similar restrictions apply to all of the parameters used in our implementations. If no restriction can be placed, the simply giving a large enough state space should suffice for finding a local optimization. In restricting the possible values of our parameters by both type and size, we create a space $\mathcal{Z}$. The problem is then to optimize some objective function, f, by traversing this space of possible values. For each $z \in \mathcal{Z}$, we calculate the objective $f(x)$ and use this value to inform us as to what the optimal might be and where it might occur. We then choose a new $z' \in Z$ which is likely to inform us the most about the optimal value and location.

The above process can be accomplished quickly using Bayesian optimization, as discussed in by Adams et al.[3]. For our purposes, the description of how this is accomplished is outside the scope of this paper. Instead, we simple explain the software used and the parameters that needed to be set for our uses.
The software used is the open source system Spearmint, hosted on github. [10]. We defined are objective function rather simply as follows:

1. First, we load our SwingyMonkey game and train over 100 epochs (live/death cycles)
2. Next, we continue our training for another 100 epochs.
3. Lastly, we average the results of the last 100 epochs and return the negation of this value, $-\bar{s}$.

The negation is returned because by default the Spearmint software system serves to minimize a function. In our particular

[10] https://github.com/HIPS/Spearmint

example, we wish to maximize. The reason we chose the above parameters is due to our definition of "good" learning as learning both quickly and well. After multiple iterations, we found that 200 iterations appeared to be a good middle ground. In the git repository for this project, the files used by Spearmint can be found in the root directory under different names.

Results and Discussion

In this section, we present the empirical results of our different models, and discuss the possible trade-offs between each. First, we note that as unsupervised learning, the first difficulty faces is in what is termed to be a "good." Intuitively, we now illustrate this dilemma with a quick example. Let $\mathcal{S}$ be our state space. The suppose we have $|\mathbb{S}_1| << |\mathcal{S}_2|$. Then it makes sense for training on $\mathcal{S}_2$ to take significantly longer. Simply exploring the state space will take longer. However, it might be possible that the second state space will lead to better results in the long-run, while the initial state space, because of its limited size, will learn quickly and converge quickly.

We've discussed this problem already in Section . As discussed there, one metric that we used involves quick learning. We actually optimized the majority of our models so that they would learn as quickly as possible over 200 epochs by running them through the SpearMint optimizer.

First, it is important to mention that the results for our model-based learner we rather disappointing. We're still exploring the possibility of a bug keeping us from achieving good results. As such, we exclude any preliminary results of the model based approach from discussion, focusing instead on TD-Learning and Q-Learning.

Long-Term Learning

For Figure 3, the mean for the final 1000 iterations is $\approx$ 130.107 and the median is 89. These are the results of TD-Learning on the same parameter space as Q-Learning without optimization. Neither takes advantage of ϵ-greedy learning or of decaying learning rates. Intuitively, we would have expected TD Learning to perform significantly better than Q-Learning. However, this does not appear to be the case. The statistics for Q-Learning are a mean of 603.99 and a median of 406. Clearly, the Q-Learning performed better. We can also observer this in the distribution of scores, as can be seen in Figure **??**.

In an attempt to improve TD-Learning, we tried different sets of parameters. As we can see in Figure 2, the parameter changes led to an improvement in the scores. The mean and median became $\approx$ 215.23752 and 144 for the final 1000 epochs. However, this is still much worse than our best performing Q-Learner.

Another interesting property in all of the below graphs, which can be seen most clearly when analyzing the Q-Learning results (Figure 4), is that the initial learning phase is clearly exponential in nature. In the log-plot, we can see a clear linear relationship in the first 2000 or so iterations. After that point, the learning begins to slow significantly. This is likely due to a saturation in the state space or a limitation of the model itself. The limitation seems least apparent in TD learning using the same parameters as Q-Learning, but this might have just been due to the fact that the learning mechanism was spectacularly poor.

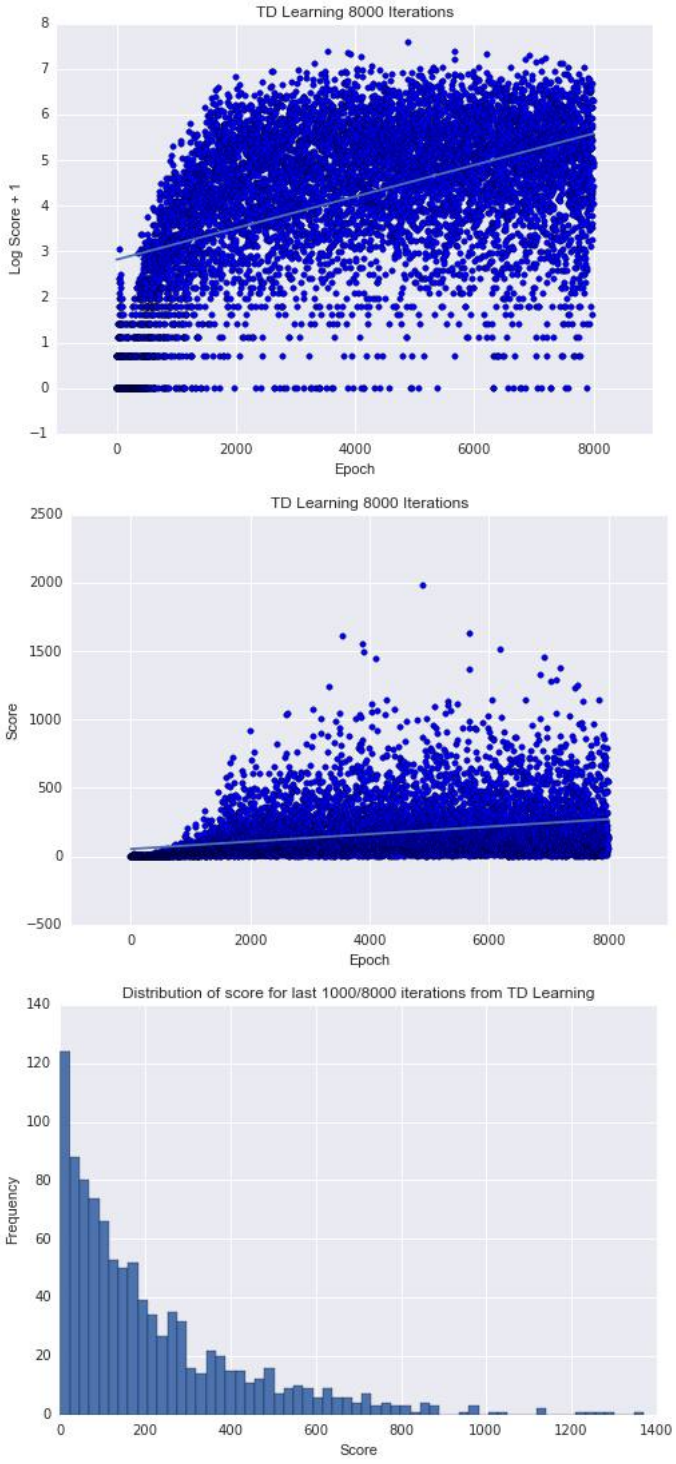


Figure 2: Plot of score vs epoch for our TD-Learner, along with distribution of scores. Parameters used: $\alpha = 0.186493, \gamma = 1.0, p_h = 236, p_w = 530, p_v = 816$

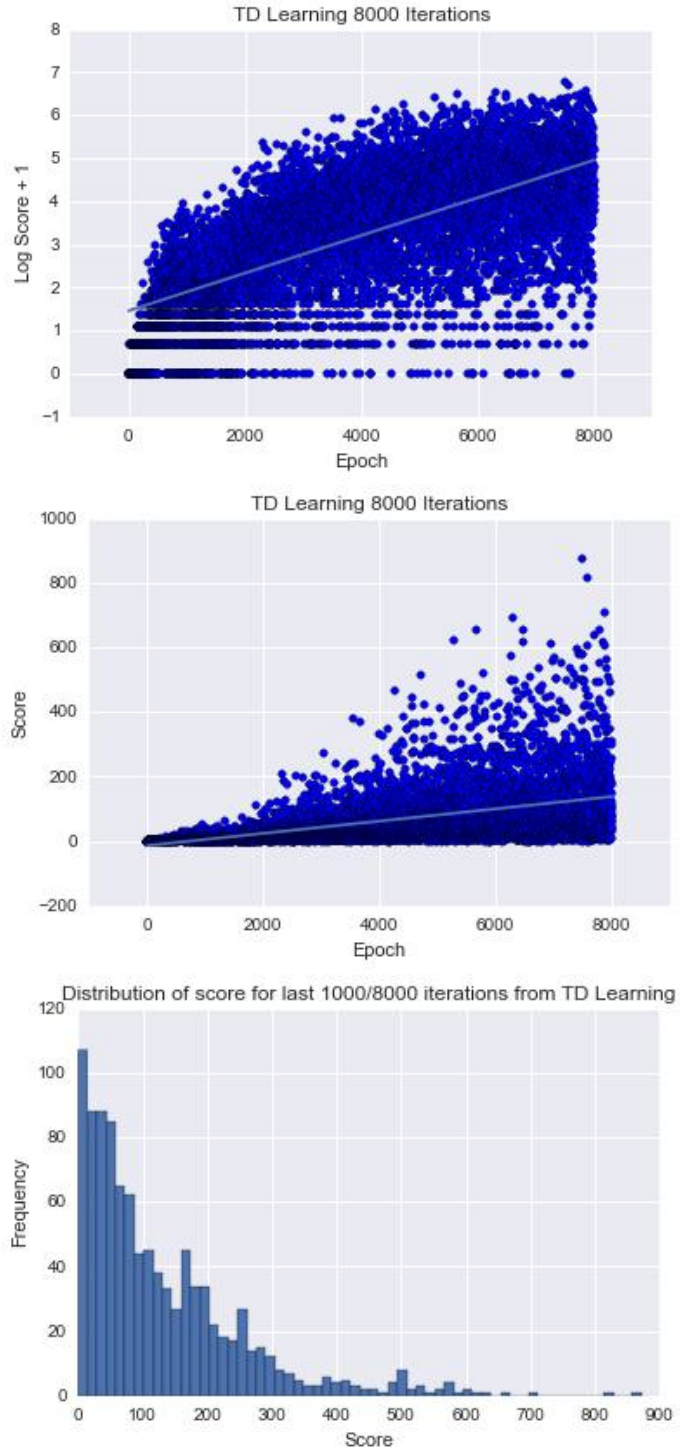


Figure 3: Plot of score vs epoch for our TD-Learner, along with distribution of scores. Parameters used: $\alpha = 0.15, \gamma = 0.95, p_h = 2, p_w = 2, p_v = 2$ (note these are the same as Figure 4)

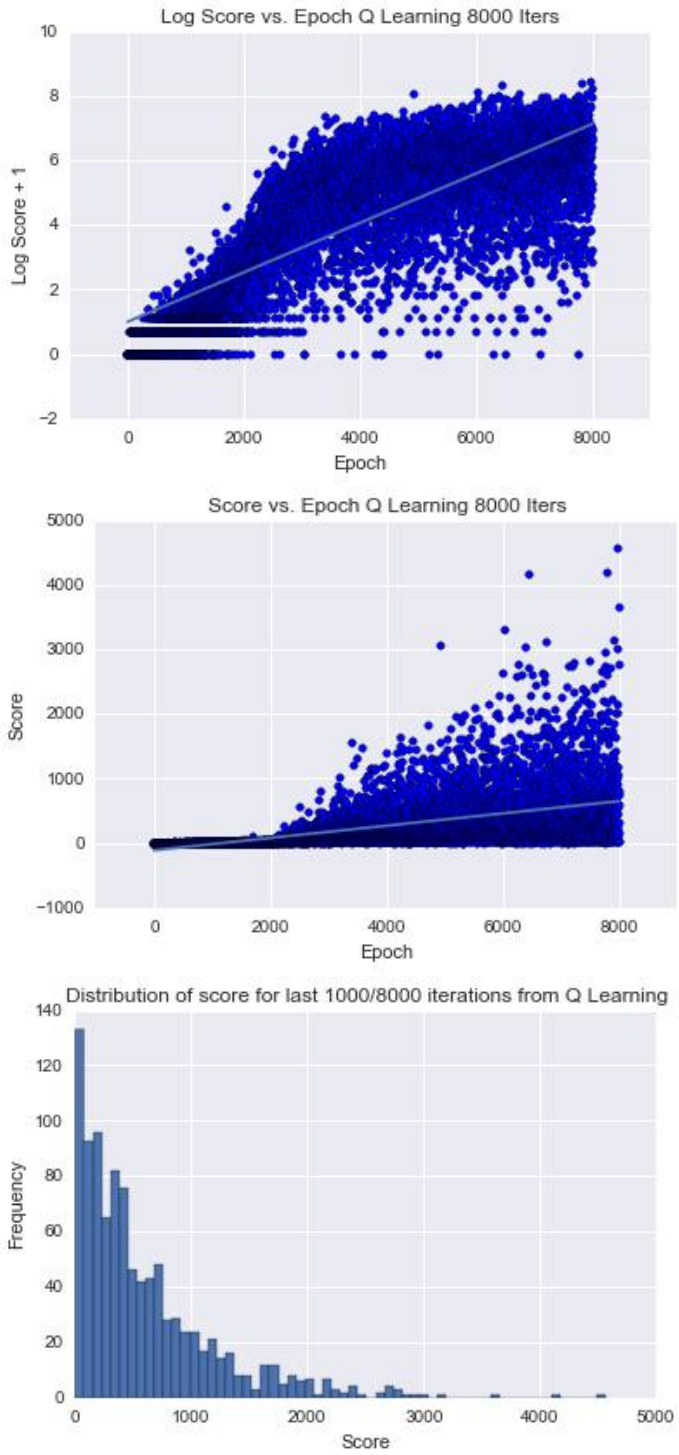


Figure 4: Plot of score vs epoch for our Q-Learner, along with distribution of scores. Parameters used: $\alpha = 0.15, \gamma = 0.95, p_h = 2, p_w = 2, p_v = 2$

Short-Term Learning

We now turn our attention to the rate of learning. As discussed previously, we actually utilized the Spearmint software to obtain parameters over the first 200 iterations. The results are shown below in Figures 5 (For TD-Learning) and 6 (For Q-Learning). It's incredible to see how quickly the monkey learns, in both models. For average statistics, we have that the Bayes optimized TD-Learner achieved a mean of 118.67 and median of 65.5 and the Bayes optimized Q-Learner achieved a mean of 44.5 and median of 26.5.

In this scenario, it's surprising to see that the Q-Learner is outperformed by the TD-Learner. We propose two possibilities for this:

- Propagation of negative reward. The TD-Learning algorithm will learn more quickly not to die. We can see that TD-Learning takes rises much more sharply at the beginning. However, this does not explain everything, since the score is actually an average of the final 100 epochs.
- TD-Learner was optimized for a longer period of time using Spearmint. The optimization took over 300 data points, while the optimization for Q-Learner only utilized 156 data points.

We lean towards the second explanation, given the fact that Q-Learner does much better than TD-Learner in the long-run for the purposes of our tests.

Conclusion

In conclusion, our analysis demonstrates a remarkable improvement of any method through the use of Bayesian parameter optimization. Given more computational resources and time, we are led to believe that optimizing the parameters for longer training periods (such as periods of 2048, 4096) and averaging over longer testing periods (such as 1024) would have led. For futher work, we would expect that allowing for more complex decay functions in the learning rate would improve the final results. In our preliminary tests with **qlearner4.py**, we already see some of these results. However, another area of interests would be to apply Dyna-Q or other non-converging algorithms to this problem. More research remains to be done into this particular problem, but from our results above, it seems that ϵ-greedy Q-learning with optimized parameters using Spearmint perform well, learning both quickly and well in the long-run.

In conclusion, a Bayesian optimized Q-Learning algorithm appears to do well enough for the purposes of training the monkey to swing.

References

[1] Avi Pfeffer; revised by David Parkes and Ryan Adams. "Markov Decision Processes (continued)". Lecture notes given for Spring Semester 2015, Harvard University.

[2] Avi Pfeffer; revised by David Parkes and Ryan Adams. "Reinforcement Learning". Lecture notes given for Spring Semester 2015, Harvard University.

[3] Hugo Larochelle Jasper Snoek and Ryan Precott Adams. "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Advances in Neural Information Processing Systems* (2012).

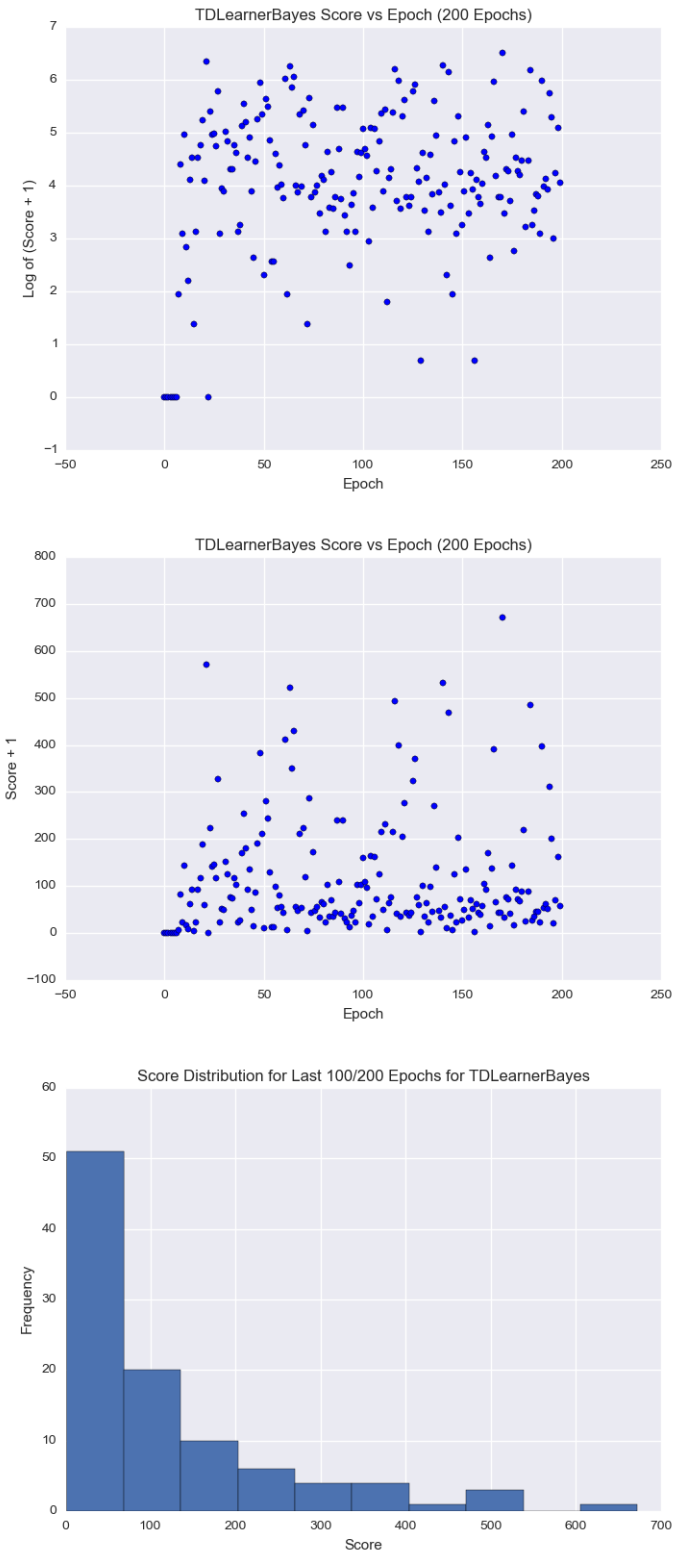
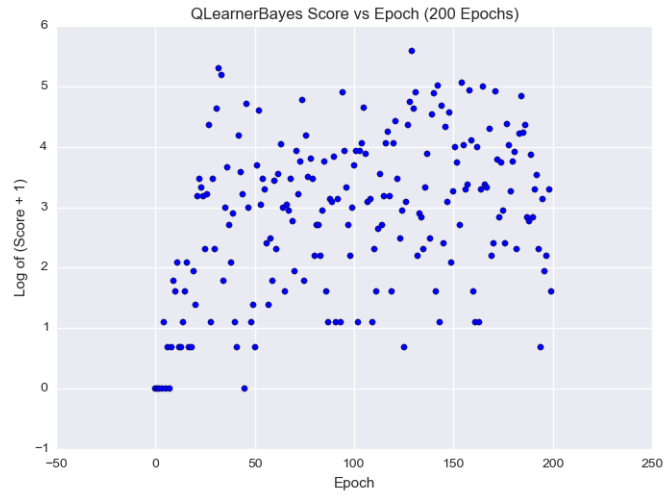
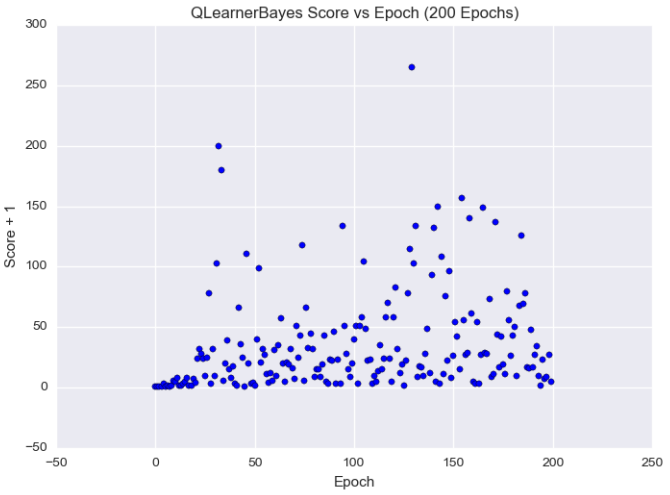


Figure 5: Plot of Score vs Epoch for our TD-Learner with Bayes Optimization over 200 Epochs, along with distribution of scores. Parameters used: $\alpha = 0.303168, \gamma = 0.954306, p_h = 227.0, p_w = 442, p_v = 440$





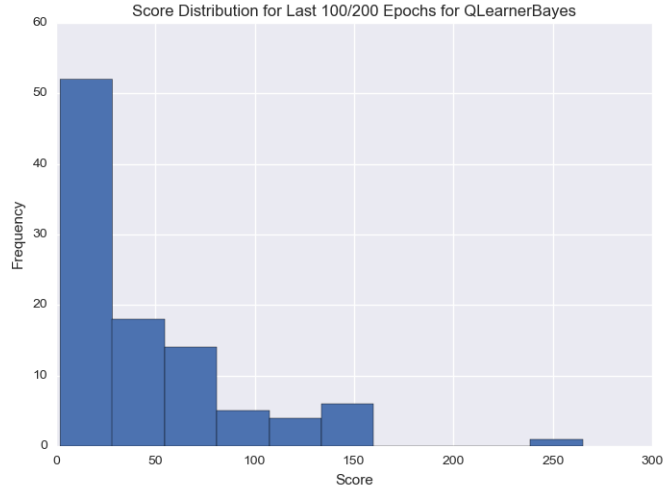


Figure 6: Plot of score vs epoch for our Q-Learner, along with distribution of scores. Parameters used: $\alpha = 0.368372, \gamma = 1.0, p_h = 16, p_w = 85, p_v = 1000$