CS181 Practical 4 : Reinforcement Learning

Team [J.Q.G]
Jeewon (Grace) Hwang, Qing Zhao, and Jing Wen

April 2016

1 Technical Approach

The goal of this practical is to train our agent to play *Swingy Monkey* using Reinforcement Learning. Our goal is to come up with an optimal policy that would maximize our action-value expectation, i.e. take actions to passing a tree trunk and avoid hitting a tree trunk, failing off the bottom of the screen, and jumping off the top of the screen. To achieve our goal, we mainly apply *Q-learning* to find optimal policy.

1.1 Features Engineering

In total, we have 9 sets of features:

- Features revealed in the gaven states:
  - 'tree_dist': pixels to next tree trunk
  - 'tree_top': height of top of tree trunk gap
  - 'tree_bot': height of bottom of tree trunk gap
  - 'monkey_vel': current monkey y-axis speed
  - 'monkey_top': height of top of monkey
  - 'monkey_bot': height of bottom of monkey
- Features inferred from given information to reduce number of dimensions
  - 'relative_top': relative distance between the top of the monkey and top of tree trunk
  - 'relative_bot': relative distance between the bottom of the monkey and bottom of tree trunk
- Features inferred during monkey's movement
  - 'gravity': the gravity varies from game to game, making the game a light-weight POMDP. We are able to **infer the gravity by making the monkey swing downward (action = 0) in the first two actions**, getting first two vertical velocity values and therefore compute gravity at the very beginning

Figure 1 illustrates the most-often-used features in our reinforcement learning process.

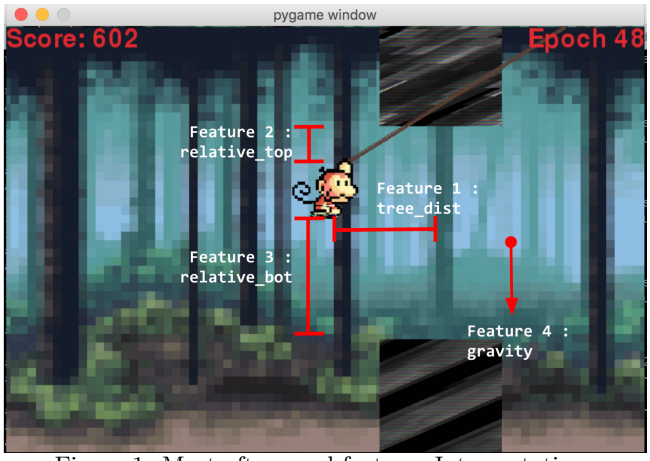

Figure 1: Most-often-used features Interpretations

In practice, we create one dimension in Q score space for each feature (detailed implementation will be covered in "Method: Code Implementation" part), which will make the Q score space overwhelmingly large and sparse if we use all the 9 features listed above in their raw format. To reduce the number of dimensions and the length of each dimension, we introduce the following two approaches:

- **Discretization**: instead of using the continuous values for distance and velocity, we discretize them into a certain number of bins to reduce the number of values in each dimension. The number of bins ranges from 5-30, and we'll experiment with different combinations to find those which yields best learning results.
- **Subset of Features**: obviously relative distance between monkey and tree chunk is enough for the monkey to make the decision of whether to jump or not. By using "relative_top" and "relative_bottom" to replace 'tree_top', 'tree_bot', 'monkey_top', 'monkey_bot', we are able to reduce the number of features by 2. In fact, in our experiment, only taking 'relative_top', 'relative_bottom', 'tree_dist' and 'gravity' into account is enough to give satisfying results.

1.2 Method

Q-Learning

The main reinforcement learning method we are using here is *Q-learning*. It is a good candidate method because it is model-free and allows us to learn the Q function directly. Since the states may grow exponentially as the bin_size gets smaller or the number of features describing a state grows, it is advantageous to ignore the transition model of getting from one state to another. In addition, because the game involves a lot of states, planning may take a long time and can be exceptionally hard to solve. Q-learning allows the agent to skip planning, but it needs to gain a number of periods of experience, in this case, plays the game many epochs, to learn a good policy.

The Q function is defined as below:

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a' \in A} Q(s',a')$$

And we can use the following rule to update the Q function:

$$Q^{new}(s,a) \leftarrow Q^{old}(s,a) + \alpha \left((r + \gamma \max_{a' \in A} Q^{old}(s',a')) - Q^{old}(s,a) \right)$$

Where r is the reward received on transitioning from s to s' under the action a'. α is the learning rate, i.e. how much do we update our Q function with the error term $(r + \gamma \max_{a' \in A} Q(s',a')) - Q(s,a)$. γ is the discount rate of future reward, as a measure of the importance of the measure of future return. Since the agent makes decision based on Q functions and α and γ defines how Q functions will be updated, tuning parameters, such as α and γ, is just as important as feature engineering. More will be discussed under **Parameters Tuning**.

Code Implementation

As shown in Figure 1, we used 4 features(tree_dist, relative_top, relative_bot, gravity) to represent our state. So we created Q-matrix to have 5 dimensions, including action as following.

self.Q[tree_dist][relative_top][relative_bot][gravity][action]

We initialized the Q table to be all zeros at first, and then updated in reward_callback function by referring 2 previous states. The code implementation is as following.

```
def reward_callback(self, reward):
    ...
    #*********************************************************************
    # oldQ: old Q value given state and action
    # [tree_dist_index][tree_top_index]\
    # [tree_bot_index][self.gravity_index][self.last_action]
    # specify the current state
    #*********************************************************************
    oldQ = self.Q[tree_dist_index][tree_top_index]\
                [tree_bot_index][self.gravity_index]\
                [self.last_action]
    #*********************************************************************
    # newQ:  a future estimate of what that reward is based on where we go.
    # [new_tree_dist_index][new_tree_top_index]\
    # [new_tree_bot_index][self.gravity_index] specifies the new state ended up
    # by taking action in terms of bin indices
    #*********************************************************************
    newQ = np.max(self.Q[new_tree_dist_index][new_tree_top_index]\
                         [new_tree_bot_index][self.gravity_index])

    #*********************************************************************
    # Updating Q values according to the Q-learning equation
    #*********************************************************************
    self.Q[tree_dist_index][tree_top_index][tree_bot_index][self.gravity_index]\
        [self.last_action] = oldQ + self.alpha*((reward + self.gamma*newQ) - oldQ)
    ...
```

ϵ-greedy policy

We also adopted ϵ greedy policy, where the optimal action is taken with probability 1-ϵ, and with probability ϵ, a uniformly random action is taken to induce exploration. Since we initialize the Q table to be all zeros, it wouldn't be filled up yet enough to give us reasonable action in the earlier epoch. So it is necessary to use ϵ-greedy policy to explore more states.

We used ϵ to a small number that decreases as the time goes by. For example, we experimented with $\epsilon = \frac{1}{t}$, where t is the number of action-reward iterations within a epoch, and $\epsilon = \frac{1}{epoch}$ so that the agent refers Q table only in later epoch.

Parameters Tuning

Since learning-rate α, and future reward decaying rate γ are very important when we update Q function, i.e. the performances of our agent, we spend a great effort tuning them. In addition, to balance exploitation and exploration, we also need to tune the parameter ϵ to ensure that the model have explore enough at the beginning and explore more when it learn enough about Q functions.
Few ways that parameters can alter the update rules:

1. Have a fixed value.
2. Start with a fixed value per epoch, and update as more actions taken.
3. Update by certain number of epochs, e.g. let α or ϵ get smaller as the agents have played several epochs.

2 Results

In practice, our monkey is able to get score of 700+ as shown in Figure 2, but the result is not stable – after a high peak the score quickly drops down and stays low. In the following part of the report, we'll present and analyse all the results we get by trying out different set of parameters. Note that to observe better trends and steer away random noise, all the scores are moving average with a span of 10.

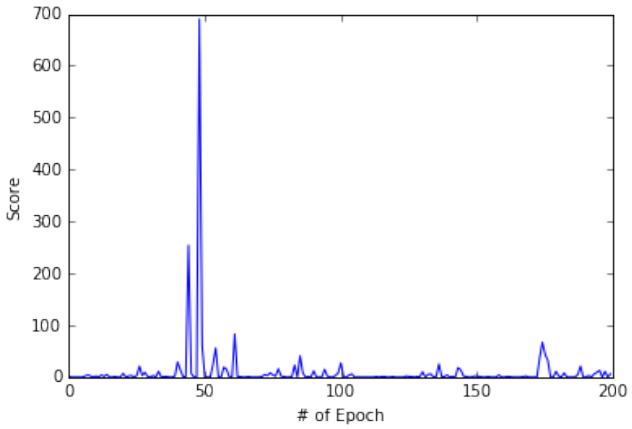

Figure 2: Our best Result with $\alpha = 0.1, \gamma = 0.95, \epsilon = \frac{1}{t}$ and bin_size=[10,10,10,4]

2.1 Not Updating Q function after failures

At the beginning, our code does not update Q function whose action leads to failures. Under this setting, we experiment with several alternation of parameters aiming to achieve higher scores.

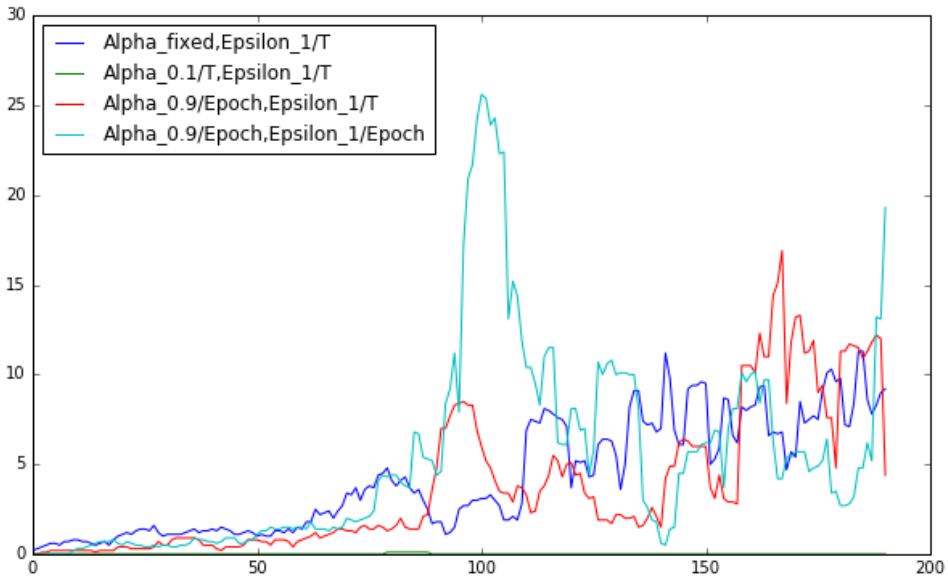

Figure 3: Moving Average of Scores under different parameters tuning (window=10)

The graph above summarizes the performance after we tuning our parameters. The blue line is our default setting, i.e. α is fixed at 0.1, and ϵ is decreasing over time within an epoch. Then, we make α decrease over time within an epoch or let α change over epoch. In addition, we also let ϵ decrease per epoch. As it shows in the plot, updating α per epoch is better than updating per action. Updating ϵ per epoch gives us some exceptional

high performance, while updating ϵ per action within an epoch allows a more consistence improvements or learning curve.

2.2 Updating Q function after failures

Theoretically Q function should be updated even receiving negative rewards. Such updates would then reflect the whole system. We fix our code to update Q function after our agent fails each time. As a result, we are able to get score as high as 680 for one epoch, see Figure 4. Such improvement in the performance of our agent demonstrates the importance of learning from failures.

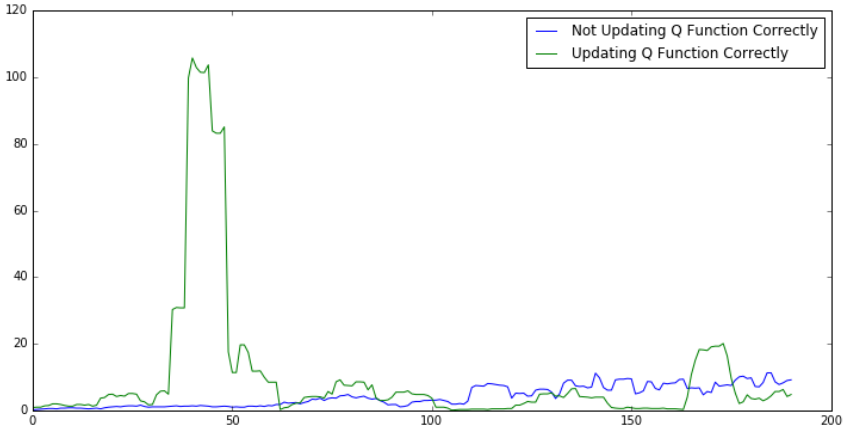

Figure 4: Moving Average of Scores with Q function updated correctly

At the same time, we notice that there are some randomnesses in the our method. With the same set of parameters, if we run 200 epochs twice, the learning curves of our agent are quite different. It may learn pretty well, i.e. achieve higher scores than other times. Thus for future analysis, we need to take this randomness property into consideration.

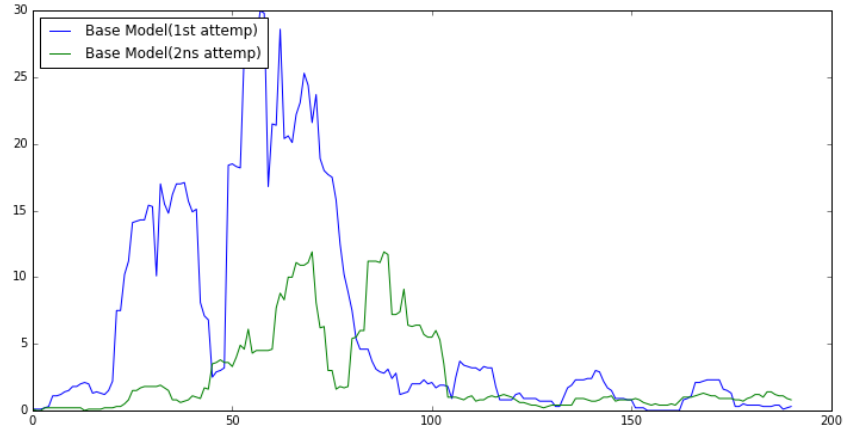

Figure 5: Stochastic property embedded in our method

To gauge the effect of each parameter and feature on the learning of our agent, we set our base model to be $\alpha = 0.1$,$\gamma = 0.95$,$\epsilon = 1/T$, i.e. it is updating per action within an epoch. We use *Tree Dist*,*Relative Top*, *Relative Bot*, and split the values into 10 bins.

Avoiding any confounding effects, we tune one parameter at a time.

Paramter α

First, we tune the learning rate α. In our base model, we set the $\alpha = 0.1$ for the whole time. We reasoned

that a large value of learning rate may result in better result, i.e. updating Q function with greater magnitude. However, as α gets larger, it may get harder to converge. Thus, we set α to be 0.5. In addition, we make α decrease as the number of epochs gets larger. Our agent would have a high learning rate at the beginning, and as it learns more, we expect the Q function to update less and converge. Based on the Figure 6, $\alpha = 0.5$ results in a better learning curve. This may not be conclusive due to the stochastic property of the method. A better way would be to repeat the experiment several times and compare the average performances across different settings of parameters.

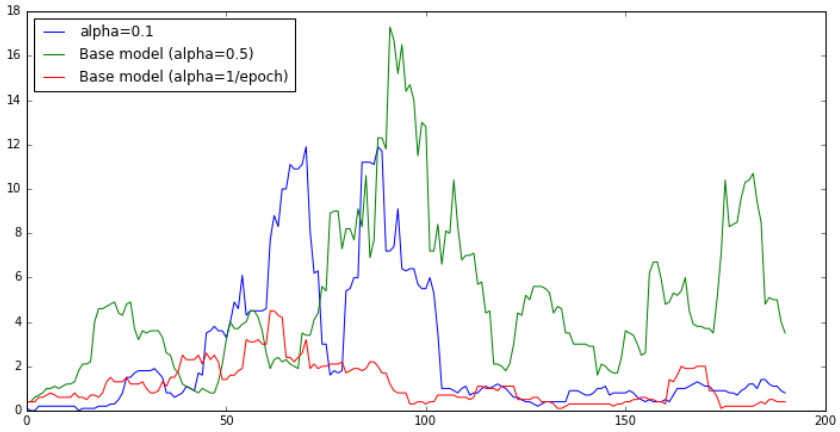

Figure 6: Fixed everything and alter α

Parameter ϵ

To balance exploitation and exploration, our base model lowers the ϵ value as our agent takes more action within an epoch,i.e.$\epsilon = 1/t$. Thus, at each epoch, we start with exploring a lot and then explore more after few actions. This update rule may be too frequent. After few epochs, the agent probably has learned a lot from before and should explore more instead of exploit at the beginning. Thus, we update ϵ as a function of the number of epochs that the agent learned. Based on the Figure 7, we conclude that setting $\epsilon = 1/Epoch$ performs substantial better than our based model.

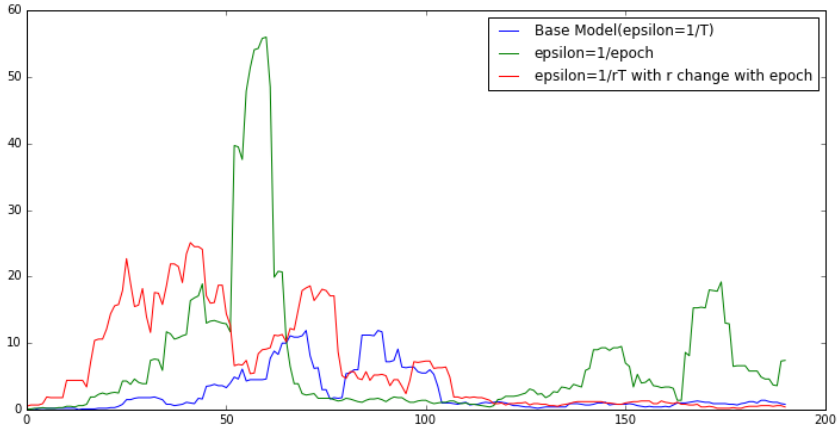

Figure 7: Fixed everything and alter ϵ

Parameter γ

We start with setting the value of future discount rate γ to be 0.95 because we expect the future Q function has a great impact in our current decision. For example, in one scenario, if the monkey does not jump at current state, it will hit the tree trunk or fail below. Thus, we want to select the action that would maximize the future expectation, i.e. passing the tree trunks. To explore the effect of γ on the performance of our agent, we try out

different values of γ. As shown in the Figure 8, there are not much differences in the learning curves given the stochasticity of our method.

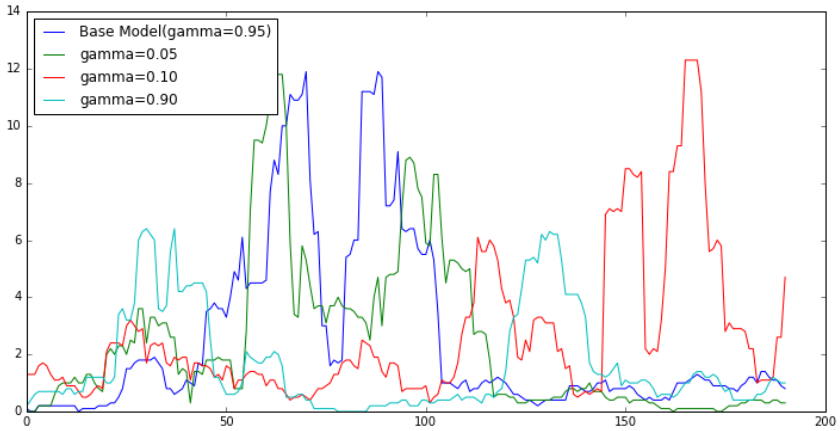

Figure 8: Fixed everything and alter γ

The number of bins

In addition to parameters, we also tuned the features. The number of the bins defines the number of state in our model. If there are too many states, it will take a long time for our agent to learn all the states. Thus, sometimes, we observe that the performance is not consistent: our agent can get a very good score at one epoch but worse scores afterward. This is because the agent was able to learn some states very well but not much about other states if there are too many states. On the other hand, if we set the number of bins to be large, our features can describe the state of the monkey more precisely and allow our agent to make a better decision.

Therefore, we experiment with different number of bins. Figure 9 shows that they does not improve the learning of our monkey much.

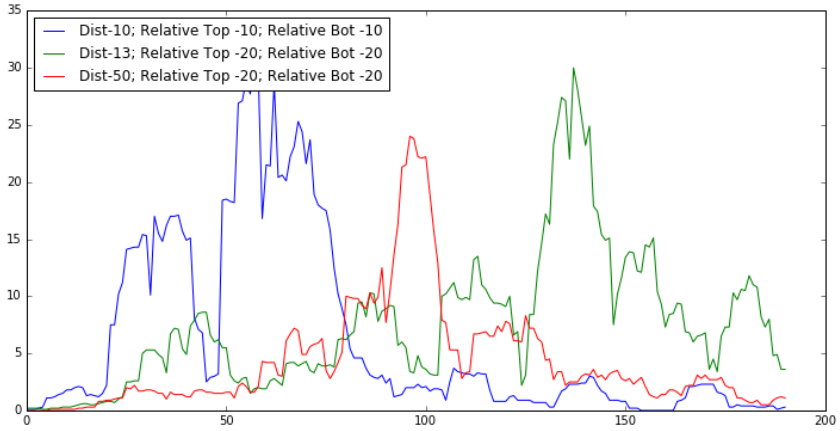

Figure 9: Fixed everything and bin size for features

Adding Features

Currently, we only consider the distance between the monkey and the tree, the vertical distance between the top of the monkey and the top of tree trunk, and the vertical distance between the bottom of the monkey and the bottom of tree trunk. The velocity of the monkey and the position of the monkey are not used to describe the states. However, these features may provide valuable information. We add the velocity and absolute position of monkey one by one to our based and test it with our agent. Figure 10 shows that adding absolute position improve the performance slightly.

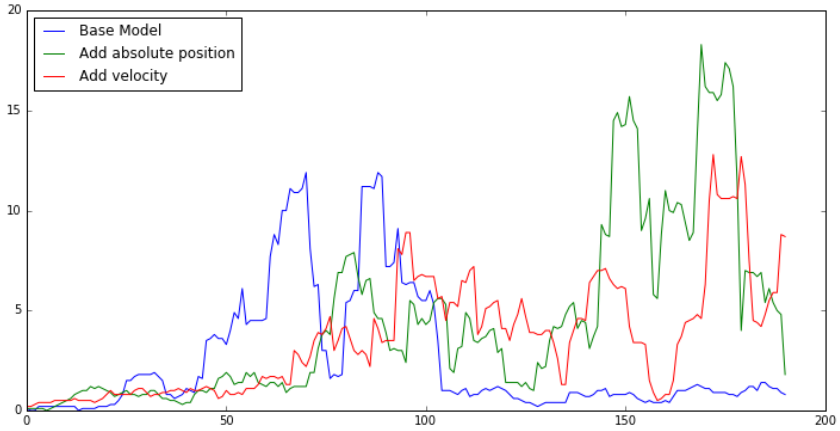

Figure 10: Fixed everything add more features

Integration of all the best parameters
Now that we have the exploration of all the parameters, namely: α, ϵ, γ, bin_size, $different\_set\_of\_features$, our next step is to integrate the best choices of each parameter and see if this could yield better result. Figure 10 demonstrate the result of the integration.

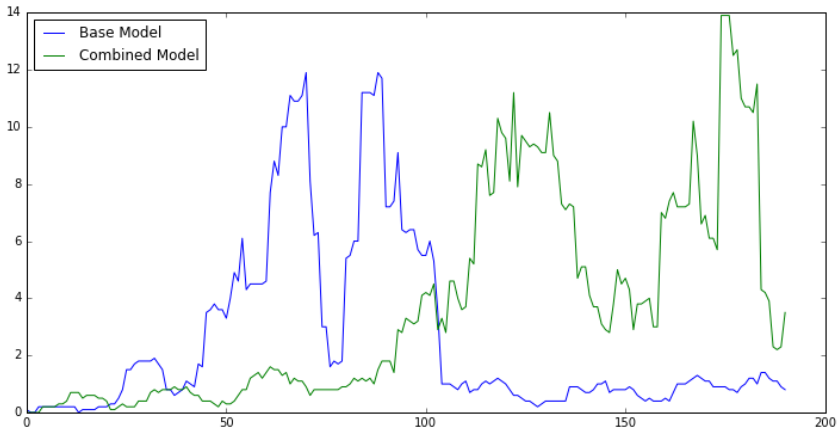

Figure 11: Combined Model

3 Discussion

Based on all the graphs listed above, we observe the following phenomenons which are counter-intuitive, and in the following discussion part we shall focus on the reasoning behind those.

Why Score moving average does not increase with the number of Epochs?

Shown in the Figures, the scores does not increase monotonically. Instead, we observe some peaks and flat curves after peaks. Since Q-learning does not require the agent to learn all the states before taking action, it is very likely that it was able to explore some states well but not so much for other states. Because there are a lot of randomness in our methods and in the game, the agent may encounter new states at each epoch. This explains that the moving averages do not increase with the number of epochs.

Why every time after score reaches peak, it quickly decrease to really low values and seldom rise again?

While doing the experiment, we realized that once the agent learned, meaning that once the agent performed significantly high score, there was a tendency that the performance suddenly decreased after that high score peak, especially after 100 epoch. This happened when bin_size was small, for instance when we use state space to be Q[10, 10, 10, 4].

However, when we used larger bin_size, setting state space to be Q[50, 20, 20, 4], the agent performed relatively stable compared to the smaller bin_size case, meaning that the agent consistently yielded high score several times over the game.

We tried to figure out the reason behind this by printing out Q values, then we concluded that this is because the Q-score diverged quickly in relatively small state space. This also explained why our agent in larger state space tend to show good performance in the later epoch. That being said, now our thought moved to tuning the learning rate α, to be small so that even in small state space the Q table can be converged.

Therefore, we tried to tune α to be decreasing after every 50 epochs, meaning $\frac{epoch}{50} + 1$, so that α will remain the same within 1 50 epoch, but get decreased after 50 epoch by integer division. However, judging by Figure 6, we see this does not increase the agent's performance. In fact, the case where $\alpha = 0.5$ gave us best result. We struggled a lot in thinking why the change of learning rate does not help as expected, but honestly it was hard

Why the "Best parameter combination" turns out to perform really badly?

As shown in Result part, after we plotted how different set of parameter affect the result, then we selected better parameter (α, γ, ϵ, bin_size, and additional features) among the experiments we did. Then we tried to run with that parameters, however, it did not give us any improvement. From this, we realized that the randomness has too much impact thus yet we were not able to infer whether a certain value for a parameter will keep showing the same effect as we've seen in our experiment. Besides, it was hard to capture general tendency of different set of values so we had difficulty to while setting the reasoning and logic behind values for parameters. This might be due to the reason that we did not do experiments several times so that we could not exclude the effect from randomness when inferring the influence of a certain parameter.

Also, when we do experiment, we fixed the rest of parameters and tuned one parameter at a time. Therefore, that parameter might be yielding the best result only under that environment, having the same values for the rest of parameters. Therefore, this can be the reason why the combination of Best parameter turned out to performing really bad.

What factors cause Randomness that leads to such huge difference between two experiments with same set of parameters?

First the game is not deterministic. Each time when our agent starts the game, it would start with a different state that it may have not explored before and since at first all Q's are 0, it might go to completely different states space. Secondly, it is very hard to describe a state given all the features are all continuous. Because all the features are continuous, there are actually many unique states, which requires a long learning period. Since we binned our features, this lost of precision may also explain the difference in performance.

In addition, the update rule is similar to stochastic gradient descent method. Thus two experiments with same set of parameters would generate different results, not mentioning the ϵ greedy algorithm incorporated in our approach.

More importantly, it also means that there are spaces for improvement. The set of parameters are not ideal, and we should keep tuning our parameters to get more consistent results.

4 Future Improvement

Our monkey generally performs well: under the best parameters, it usually starts to get high scores of over 100 and even 300 after 50 iterations. However, under some random cases, the performance is consistently poor. Furthermore, as shown in result and discussion part, there are a lot of randomness and counter-intuitive cases (which are still hard for us give sound explanations). These lead us to think a lot about what we can do in the future:

Parameter Tuning: Test multiple times and calculate average
Apparently, the conclusions we get from current tuning of parameters are still dubious given Figure 11. One straightforward way to improve this is to carry out multiple experiment for a single set of parameter combination and getting the average. This will greatly improve the confidence of conclusions and therefore provide more accurate insight of the learning process.

Try neural network as function approximation to reduce state space
This time we to reduce the number of state, we simply discretize the position space into bins. This naive approach could cause a lot of potential issues, as already been reflected in the result part. One major issue is the number of visits for each state is not evenly distributed, which means some bins are frequently updated and will never converge, whereas others are rarely visited. To address the problem, we think it could be a better idea to use function approximation, such as a neural network, to represent the value function or the Q-function. This way, the system will learn by itself how different parts of states should be weighed and connected.

Use temporal difference learning
Temporal difference (TD) learning is a prediction-based machine learning method. It has primarily been used for the reinforcement learning problem, and is said to be "a combination of Monte Carlo ideas and dynamic programming (DP) ideas."TD resembles a Monte Carlo method because it learns by sampling the environment according to some policy, and is related to dynamic programming techniques as it approximates its current estimate based on previously learned estimates (a process known as bootstrapping). Thus we believe by applying temporal difference model for monkey's reinforcement learning, we could do better in making decisions on monkey's next best action than what we have currently, which simply use Q-learning that keep updating Q score based on rewards and future states.