Reinforcement Learning II

Table of Contents

[-Greedy 2](#_Toc138878186)

[Exploration Functions 2](#_Toc138878187)

[Regret 3](#_Toc138878188)

[Approximate Q-Learning 4](#_Toc138878189)

[Feature-Based Representations 5](#_Toc138878190)

[Approximate Q-Learning 6](#_Toc138878191)

[Least Squares 7](#_Toc138878192)

[Policy Search 8](#_Toc138878193)

To ensure that our agent learns the optimal actions, we need to ensure that it is exploring its options enough. We also need to make sure it knows when to stop exploring an option. There are several ways to go about this, the simplest of which is called -Greedy.

## -Greedy

Under the **-Greedy** algorithm, at every transition, we make a random move with a small probability, , and follow the policy with a large probability, . The problem with this algorithm is that, once the agent is done learning, it keeps thrashing around.

To avoid the issue of thrashing, we can either lower the value of over time, or we can use an **exploration function**.

## Exploration Functions

An **exploration function** allows us to automatically decay the probability of exploring an action based on how much we have learnt about the action. To do this, we have to keep track of not only the utility of a state and action pair, but also a count of the number of times we have explored that pair. This is then plugged into the exploration function.

Here, is a hyperparameter. Notice that as the value of increases, the value of will decrease.

The original equation for Q-value updates was as follows:

This is now changed to:

Thus, actions that have not been thoroughly explored are given a bonus. The bonus also propagates back to states that allow us to reach a state in which we have unexplored actions.

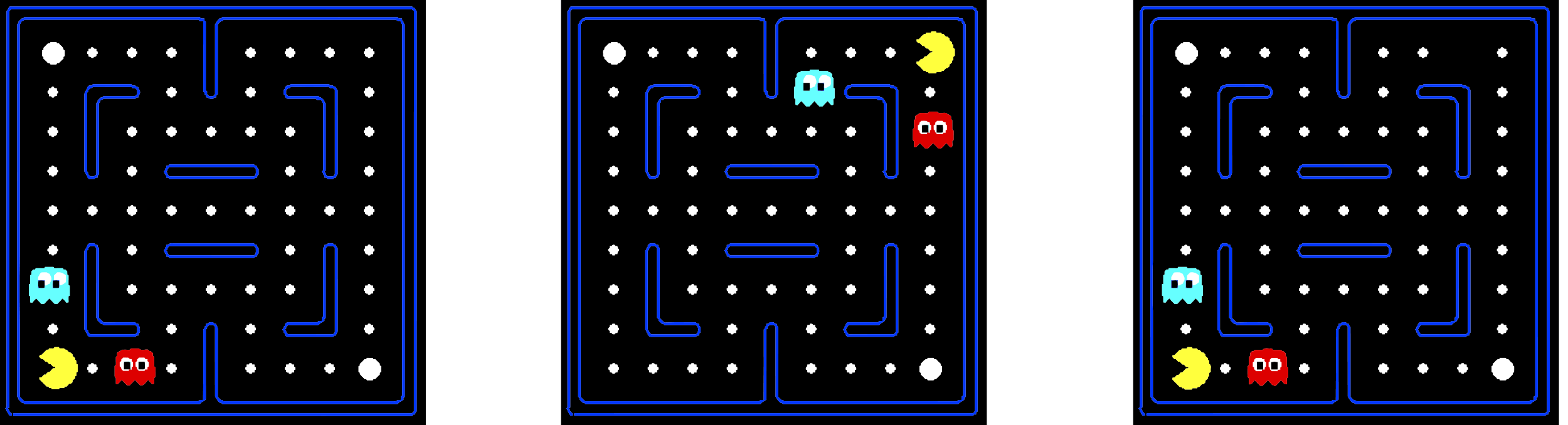
### Regret

The use of an exploration function ties in with the concept of **regret**. Regret is a measure of the amount of mistakes we make. Minimizing this is not just about learning to be optimal, it is about learning to be optimal in an optimal manner. If we compare random exploration with an exploration function for example, using an exploration function has a lower regret.

## Approximate Q-Learning

Basic Q-Learning keeps a table of all the Q-values. In real-life situations, this is not possible. There are too many states to visit during the training process and too many values to hold in memory. Instead, we want to **generalize**. We want to learn about a small number of training states from experience and then generalize that experience to new but similar situations. This is a fundamental idea behind machine learning. A model cannot be shown every picture of a cat, but we want it to learn from a few pictures and then be able to classify new but similar pictures of cats.

Consider the three examples below:



In naïve Q-Learning, all three of these states would be different and would require exploration individually. If we can generalize the approach, our agent would only need to explore the first situation to be able to learn that the other two are also bad.

## Feature-Based Representations

One approach to generalize naïve Q-Learning is to use **Feature-Based Representations**. Every state is described using a vector of features. For Pacman, this could be the distance to the closest ghost, the distance to the closest food pellet, etc. The vector of features for a specific state is provided by a **function**. We can also describe Q-states in this manner, i.e., the function will return the vector of features that result from taking an action when at the state .

Using this feature representation, the Q-function can be written as:

Here, we are taking every feature and multiplying each with a **weight** that represents how important that feature is. These weights are what the model needs to learn.

The benefit to this representation is that we just need to store weight values for the features instead of having to store every state. The disadvantage is in choosing which features to use. If we choose too few, the model will not have enough information to make correct decisions, a situation called **underfitting**. If we choose too many, the model will start paying attention to features that are not relevant, a situation called **overfitting**.

## Approximate Q-Learning

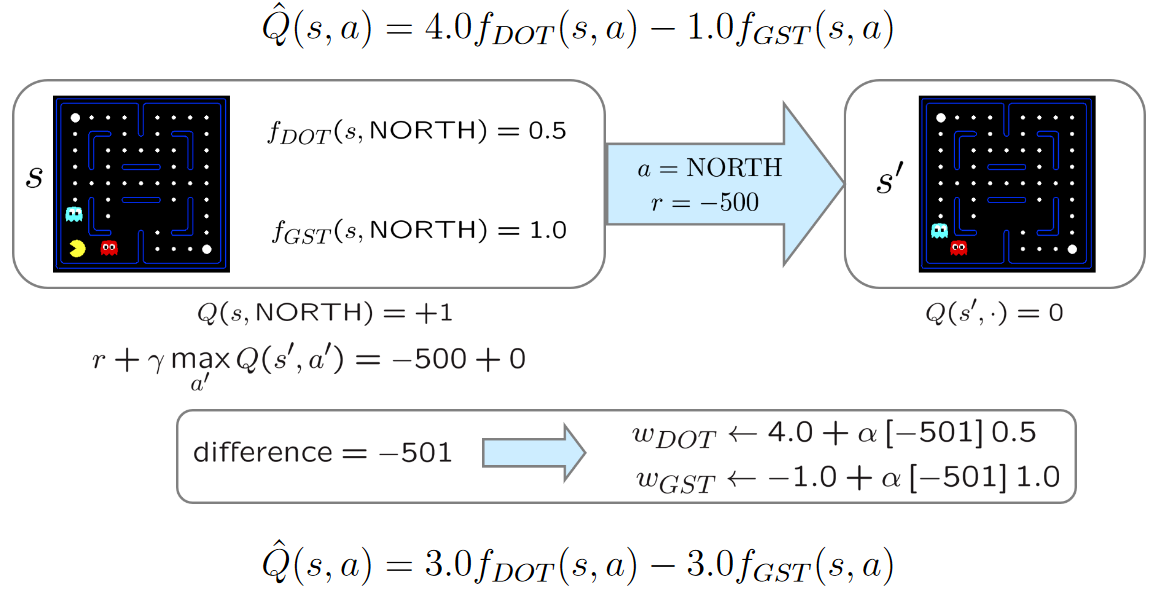
Initially, the weight values are random. As we start exploring, instead of updating the Q-values, we update the weights.

Here, is the difference between the new value we got and the old value.

If the difference is positive, that means the score should be higher, so the weight is increased. If the difference is negative, it means the opposite, so the weight is decreased.

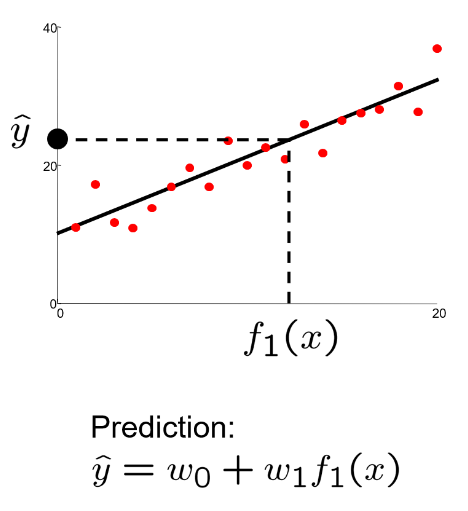
Note that this process is being executed for all the states, which means the weights depend on all of them. The states are thus comparable to samples of data.

The example below should make this process much clearer.

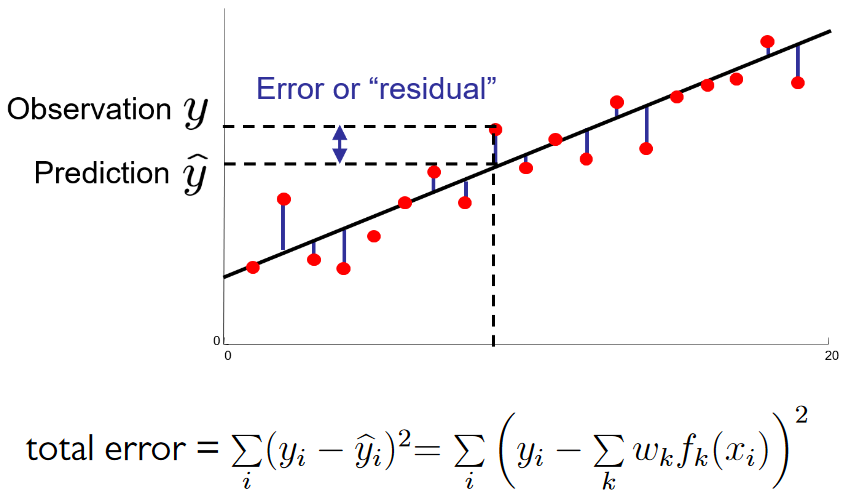


## Least Squares

The key idea behind feature-based representation is the same as that of linear regression. We have a line represented by the weights, and we want to adjust the weights so that the line best fits the results we are getting for the different states.



For a specific set of weights, we will get some error value.



Our task is to minimize the error.

Note: The above equations only consider a single point, .

In the above equations, we are multiplying the error by simply to make the following differentiation easier. The multiplication has no effect on the overall outcome.

The final equation shown above is exact what is happening in the approximate q-update.

## Policy Search

The feature-based representation has one big flaw though. It is trying to make sure the Q-values are correct. What we actually need is for the policy to be correct. Those two things won’t necessarily give the same outcome.

The workaround is to use something called **policy search**. This involves starting with an initial linear value function or Q-function and adjusting the weights a little to see if the resulting policy is better than before.

Policy search in turn has several issues:

* How do we tell if a policy is better?
* We need to run many sample episodes for each policy
* For a large number of features, policy search becomes impractical

Methods that are better than policy search try to avoid these issues by using lookahead structures, sampling wisely, changing multiple parameters, etc.