A Case Study in Unsupervised Learning

Now we'll move into a small case study of unsupervised learning- The $n-$armed bandit. Looking at this problem will allow us to further our experience in Matlab and simulation modeling.

The one armed bandit is slang for a slot machine, so the $n-$armed bandit can be thought of as a slot machine with n arms - or equivalently, you may think of a room with n slot machines.

The problem we're trying to solve is the classic Las Vegas quandry: How should we play the slot machines in order to maximize our returns?

First, let us set up some notation: Let a be an integer between 1 and n that defines which machine we're playing. Then define the expected return:

$$Q(a) = \text{ The expected return for playing slot machine } a$$

You can also think of $Q(a)$ as the *mean* of the payoffs for slot machine a.

If we knew $Q(a)$ for each machine a, our strategy to maximize our returns would be very simple: "Play only machine a".

Of course, what makes the problem interesting is that we don't know what the any of the returns are, let alone which machine gives the maximum. That leaves us to estimate the returns, and because there will always be uncertainty associated with these estimates, we will never know if the estimates are correct. We hope to construct estimates that get better over time (and experience).

Let's first set up some notation. Let

$$Q_t(a) = \text{ Our estimation of } Q(a) \text{ at time } t.$$

so we hope that our estimates get better in time:

$$\lim_{t\to\infty} Q_t(a) = Q(a) \tag{1.1}$$

Suppose we play slot machine a a total of n_a times, with payoffs $r_1, \ldots, r_{n_a}$ (note that these values could be negative!). Then we might estimate $Q(a)$ as the mean of these values:

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{n_a}}{n_a}$$

In statistical terms, we are using the sample mean to estimate the actual mean which is a reasonable thing to do as a starting point. We'll also initialize the estimates to be zero: $Q_0(a) = 0$.

We now come to the big question: What approach should we take to accomplish our goal (of maximizing our reward). The first one up is a good place to start.

1.2.4 The Greedy Algorithm

This strategy is straightforward: Always play the slot machine with the largest (estimated) payoff. If a_{t+1} is the machine we'll play at time $t+1$, then:

$$a_{t+1} = \arg\max \{Q_t(1), Q_t(2), \ldots, Q_t(n)\}$$

where "arg" refers to the argument of the maximum (which is an integer from 1 to n corresponding to the max) . If there is a tie, then choose one of them at random.

We'll need to translate this into a learning algorithm, so set's take a moment to see how we might implement the greedy algorithm in Matlab.

Translating to Matlab

The `find` and `max` commands will be used to find the argument of the maximum value. For example, if x is a (row) vector of numbers, then the following command:

```
idx=find(x==max(x))
```

will return all indices of the vector x that are equal to the max.

Here's an example. Suppose we have vector x as given. What does Matlab do?

```
x=[1 2 3 0 3];
idx=find(x==max(x));
```

The result will be a vector, `idx`, that contains the values 3 and 5 (that is, the third and fifth elements of x are where the maximum occurs).

Going back to the greedy algorithm, I think you'll see a problem- What if the estimations are wrong? Then its very possible that you'll get stuck on a suboptimal machine. This problem can be dealt with in the following way: Every once in a while, try out the other machines to see what you get. This is what we'll do in the next section.

1.2.5 The $\epsilon-$Greedy Algorithm

In this algorithm, rather than always choosing the machine with the greatest current estimate of the payout, we will choose, with probability ϵ, a machine at random.

With this strategy, as the number of trials gets larger and larger, $n_a \to \infty$ for *all* machines a, and so we will be guaranteed convergence to the proper estimates of $Q(a)$ for all a machines.

On the flip side, because we're always investigating other machines every once in a while, we'll never maximize our returns (we will always have suboptimal returns).

Implementing *epsilon*$-$greedy in Matlab

Using some "pseudo-code", here is what we want our algorithm to do:

`For each time we choose a machine:`

- `Select an action:`
  - `Sometimes choose a machine at random`
  - `Otherwise, select the action with greatest return. Check for ties, and if there is a tie, pick on of them at random.`
- `Get your payoff`
- `Update the estimates` Q

```
   Repeat.
```

Our first programming problem will be to implement the statement "Sometimes choose a machine at random". If we define $\epsilon =$`E` to be the probability of this event, and `N` is the number of trials, then one way of selection is to set up a vector with N elements which we'll call `greedy`, that will "flag" the events- that is, on trial j, if `greedy(j)`$= 1$, choose a machine at random. Otherwise, choose using the greedy method. The following code will do just that (`N` is the number of trials)

```
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N); %Randomly permute the vector indices
   greedy=greedy(m);
   clear m
end
```

And here's the full function. We assume that the actual rewards for each of the bandits is given in the vector `Aq`, and that when machine a is played, the sample reward will be chosen from a normal distribution with unit variance and mean `Aq(a)`.

```
function [As,Q,R]=banditE(N,Aq,E)
%FUNCTION [As,Q,R]=banditE(N,Aq,E)
%  Performs the N-armed bandit example using epsilon-greedy
%    strategy.
%  Inputs:
%        N=number of trials total
%        Aq=Actual rewards for each bandit (these are the mean rewards)
%        E=epsilon for epsilon-greedy algorithm
%  Outputs:
%        As=Action selected on trial j, j=1:N
%        Q are the reward estimates
%        R is N x 1, reward at step j, j=1:N

numbandits=length(Aq);       %Number of Bandits
ActNum=zeros(numbandits,1);  %Keep a running sum of the number of times
                             %  each action is selected.
ActVal=zeros(numbandits,1);  %Keep a running sum of the total reward
                             %  obtained for each action.
Q=zeros(1,numbandits);       %Current reward estimates
As=zeros(N,1);               %Storage for action
R=zeros(N,1);                %Storage for averaging reward

%**********************************************************************
%  Set up a flag so we know when to choose at random (using epsilon)
%**********************************************************************
greedy=zeros(1,N);
if E>0
   m=round(E*N);  %Total number of times we should choose at random
   greedy(1:m)=ones(1,m);
   m=randperm(N);
   greedy=greedy(m);
   clear m
end
if E>=1
    error('The epsilon should be between 0 and 1\n');
end
```

```
%*********************************************************************
%
%  Now we're ready for the main loop
%*********************************************************************
for j=1:N

    %STEP ONE:  SELECT AN ACTION  (cQ) , GET THE REWARD  (cR) !

        if greedy(j)>0  %Choose a machine at random
            cQ=ceil(rand*numbandits);
            cR=randn+Aq(cQ);
        else  % Choose using greedy algorithm
            [val,idx]=find(Q==max(Q));
            m=ceil(rand*length(idx));  %Choose a max at random
            cQ=idx(m);
            cR=randn+Aq(cQ);
        end
        R(j)=cR;

    %UPDATE FOR NEXT GO AROUND!
        As(j)=cQ;
        ActNum(cQ)=ActNum(cQ)+1;
        ActVal(cQ)=ActVal(cQ)+cR;
        Q(cQ)=ActVal(cQ)/ActNum(cQ);
end
```

Next we'll create a test bed for the routine. We will call the program 2,000 times, and each call will consist of 1,000 plays. We'll also assume that the actual rewards are being drawn from a standard normal distribution (zero mean).

We will set the number of bandits to 10, and change the value of ϵ from 0 to 0.01 to 0.1, and see what the average reward per play is over the 1000 plays. To get a good average, we'll run the algorithm 2000 times. The best that is possible would be the average of the maximum values of the vector `mg` below (about 1.55).

Here's a script file that we'll use to call the `banditE` routine:

```
Ravg=zeros(1000,1);
mg=zeros(2000,1);
E=0.1;
for j=1:2000
    m=randn(10,1);
    [As,Q,R]=banditE(1000,m,E);
    Ravg=Ravg+R;
    if mod(j,10)==0
        fprintf('On iterate %d\n',j);
    end
    mg(j)=max(m);
end
Ravg=Ravg./2000;
plot(1:200,Ravg,1:2000,mean(mg)*ones(1,2000));
```

The output of the algorithms are shown in Figure 1.1. What's not shown is what happens if ϵ is increased too far- We'll explore that in the homework.

The Softmax Action Selection

In the "softmax" action selection algorithm, the idea is to construct a set of probabilities from the current estimated payoffs- the idea being that the machine with the best payoff should be selected most often, and

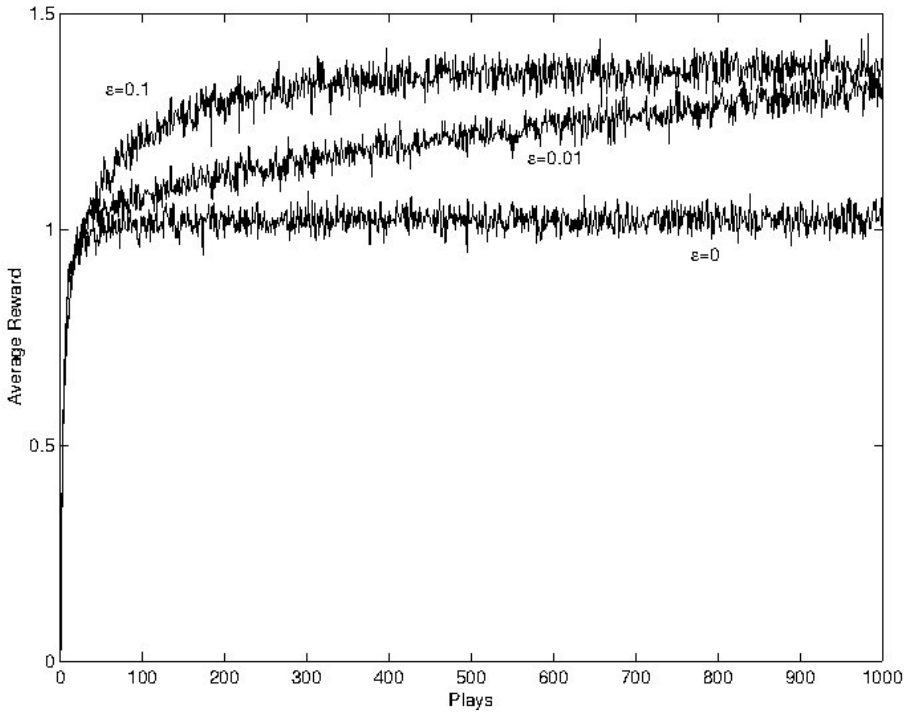

Figure 1.1: Results of the testbed on the 10-armed bandit. Shown are the rewards given per play, averaged over 2000 trials.

the machine with the worst payoff, least often- With other machines selected every once in a while as well. This has the property that, while the best machine now is selected often, we're still exploring parameter space just in case we don't actually have the best machine.

How shall we construct the probabilities? Let's review again what we want:

- The machine (or arm) giving the highest estimated payoff will have the highest probability.
- We will choose a machine using the probabilities. For example, if the probabilities are $0.5, 0.3, 0.2$ for machines $1, 2, 3$ respectively, then machine 1 would be chosen 50% of the time, machine 2 would be chosen 30% of the time, and the last machine 20% of the time.

Now if we have n machines with estimated payoffs (some could be positive, some negative) recorded as:

$$Q = [Q_t(1), Q_t(2), \ldots, Q_t(n)]$$

we want to construct n probabilities,

$$P = [P_t(1), P_t(2), \ldots, P_t(n)]$$

The requirements for this transformation are:

1. $P_t(k) \geq 0$ for $k = 1, 2, \ldots$ (because all probabilities are positive). Another way to say this is to say that the range of the transformation is nonnegative.
2. If $Q_t(a) < Q_t(b)$, then $P_t(a) < P_t(b)$. That is, the transformation must be strictly increasing for all domain values.
3. Finally, the sum of the probabilities must be 1.

A function that satisfies requirements 1 and 2 is the exponential function. It's range is nonnegative. It maps large negative values (large negative payoffs) to near zero probability, and it is strictly increasing. Up to this point, the transformation is:

$$\hat{P}_t(k) = \mathrm{e}^{Q_t(k)}$$

We need the probabilities to sum to 1, so we normalize the $\hat{P}_t(k)$:

$$P_t(k) = \frac{\hat{P}_t(k)}{\hat{P}_t(1) + \hat{P}_t(2) + \ldots + \hat{P}_t(n)} = \frac{\exp(Q_t(k))}{\sum_{j=1}^{n} \exp(Q_t(j))}$$

Here's a sample computation, with output, in Matlab. Suppose we have 4 machines with payoffs: $\{-2, 0, 1, 3\}$. To change these into probabilities:

```
Qt=[-2 0 1 3]
Ptemp=exp(Qt)
   Ptemp=[0.135, 1, 2.71, 20.08]
P=Ptemp/sum(Ptemp)
   P=[0.0057, 0.0418, 0.1135, 0.839]
```

Therefore, the machine with the highest payout would be selected about 84% of the time, the next machine about 11% of the time, the next about 4% of the time, and the lowest less than 1% of the time.

This conversion from payouts to percents uses a popular technique worth remembering- We have what is called a Gibbs (or Boltzmann) distribution. We could stop at this point, but it is convenient to introduce a control parameter τ (sometimes this is referred to as the temperature of the distribution). Our final version of the transformation is given as:

$$P_t(k) = \frac{\exp\left(\frac{Q_t(k)}{\tau}\right)}{\sum_{j=1}^{n} \exp\left(\frac{Q_t(j)}{\tau}\right)}$$

EXERCISE: Suppose we have two probabilities, $P(1)$ and $P(2)$ (we left off the time index since it won't matter in this problem). Furthermore, suppose $P(1) > P(2)$. Compute the limits of $P(1)$ and $P(2)$ as τ goes to zero. Compute the limits as τ goes to infinity (Hint on this part: Use the definition, and divide numerator and denominator by $\exp(Q(1)/\tau)$ before taking the limit).

What we find from the previous exercise is that the effect of large τ (hot temperatures) makes all the probabilities about the same (so we would choose a machine almost at random). The effect of small τ (cold temperatures) makes the probability of choosing the best machine almost 1 (like the greedy algorithm).

In Matlab, these probabilities are easy to program. Let `Q` be a vector holding the current estimates of the returns, as before, and let `t`$= \tau$, the temperature. Then we construct a vector of probabilities using the softmax algorithm:

```
P=exp(Q./t);
P=P./sum(P);
```

Programming Comments

1. How to select action a with probability $p(a)$?

   We could do what we did before, and create a vector of choices with those probabilities fixed, but our probabilities change. We can also use the uniform distribution, so that if `x=rand`, and $x \leq p(1)$, use action 1. If $p(1) < x \leq p(1) + p(2)$, choose action 2. If $p(1) + p(2) < x \leq p(1) + p(2) + p(3)$, choose action 3, and so on. There is an easy way to do this, but it is not optimal (in terms of speed). We introduce two new Matlab functions, `cumsum` and `histc`.

The function `cumsum`, which means *cumulative sum*, takes a vector x as input, and outputs a vector y so that `y=cumsum(x)` creates:

$$y_k = \sum_{n=1}^{k} x_n = x_1 + x_2 + \ldots + x_k$$

For example, if $x = [1, 2, 3, 4, 5]$, then `cumsum(x)` would output $[1, 3, 6, 10, 15]$

The function `histcounts` (for *histogram count*) has the form: `n=histcounts(x,y)`, where the vector y is monotonically increasing. The elements of y form "bins" so that $n(k)$ counts the number of values in `x` that fall between the elements `y(k)` (inclusive) and `y(k+1)` (exclusive) in the vector `y`. Try a particular example, like:

```
Bins=[0,1,2];
x=[-2, 0.25, 0.75, 1, 1.3, 2];
N=histc(x, Bins);
```

`Bins` sets up the desired intervals as $[0, 1)$ and $[1, 2)$. Since -2 is outside of all the intervals, it is not counted. The next two elements of x are inside the first interval, and the next two elements are inside the second interval. Thus, the output of this code fragment is $N = [2, 2]$.

Now in our particular case, we set up the bin edges (intervals) so that they are the cumulative sums. We'll then choose a number between 0 and 1 using the (uniformly) random number $x =$ `rand`, and determine what interval it is in. This will be our action choice:

```
P=[0.3, 0.1, 0.2, 0.4];
BinEdges=[0, cumsum(P)];
x=rand;
Counts=histcounts(x,BinEdges);
ActionChoice=find(Counts==1);
```

2. We have to change our parameter τ from some initial value τ_{init} (big, so that machines are chosen almost at random) to some small final value, τ_{fin}. There are an infinite number of ways of doing this. For example, a linear change from a value a to a value b in N steps would be the equation of the line going from the point $(1, a)$ to the point (N, b).

   Exercise: Give a formula for the parameter update, τ in terms of the initial value, τ_{init} and the final value, τ_{fin} if we use a linear decrease as t ranges from 1 to N.

   A more popular technique is to use the following formula, which we'll use to update many parameters. Let the initial value of the parameter be given as a, and the final value be given as b. Then the parameter p is computed as:

$$p = a \cdot \left(\frac{b}{a}\right)^{t/N} \tag{1.2}$$

   Note that when $t = 0$, $p = a$ and when $t = N$, $p = b$[1]

"Win-Stay, Lose-Shift" Strategy

In this experiment, we interpret the strategy as: If I'm winning, make the probability of choosing that action stronger. If I'm losing, make the probability of choosing that action weaker. This brings us to the class of *pursuit* methods.

Define a^* to be the winning machine at the moment, i.e.,

$$a^* = \max_a Q_t(a)$$

[1] In the C/C++ programming language, indices always start with zero, and this is leftover in this update rule. This is not a big issue, and the reader can make the appropriate change to starting with $t = 1$ if desired.

The idea now is straightforward- Slightly increase the probability of choosing this winning machine, and correspondingly decrease the probability of choosing the others.

Define the probability of choosing machine a as $P(a)$ (or, if you want to explicitly include the time index, $P_t(a)$. Then given the winning machine index as a^*, we update the current probabilities by using a parameter $\beta \in [0, 1]$:

$$P_{t+1}(a^*) = P_t(a^*) + \beta\,[1 - P_t(a^*)]$$

and the rest of the probabilities decrease towards zero:

$$P_{t+1}(a) = P_t(a) + \beta\,[0 - P_t(a)]$$

Exercises with the Pursuit Strategy

1. Suppose we have three probabilities, P_1, P_2, P_3, and P_1 is the unique maximum. Show that, for any $\beta > 0$, the updated values still sum to 1.

2. Using the same values as before, show that, for any $\beta > 0$, the updated values will stay between 0 and 1- that is, If $0 \leq P_i \leq 1$ for all i before the update, then after the update, $0 \leq P_i \leq 1$.

3. Here is one way to deal with a tie (show that the updated values still sum to 1): If there are k machines with a maximum, update each via:

$$P_{t+1} = (1 - \beta)P_t + \beta/k$$

4. Suppose that for some fixed j, P_j is always a loser (never a max). Show that the update rule guarantees that $P_j \to 0$ as $t \to \infty$. HINT: Show that $P_j(t) = (1 - \beta)^t P_j(0)$

5. Suppose that for some fixed j, P_j is always a winner (with no ties). Show that the update rule guarantees that $P_j \to 1$ as $t \to \infty$.

Matlab Functions `softmax` and `winstay`

Here are functions that will yield the softmax and win-stay, lose-shift strategies. Below each is a driver. Read through them carefully so that you understand what each does. We'll then ask you to put these into Matlab and comment on what you see.

```
function a=softmax(EstQ,tau)
% FUNCTION a=softmax(EstQ, tau)
%   Input:  Estimated payoff values in EstQ (size 1 x N,
%           where N is the number of machines
%           tau - "temperature":  High values- the probs are all
%           close to equal; Low values, becomes "greedy"
%  Output:  The machine that we should play (a number between 1 and N)

if tau==0
    fprintf('Error in the SoftMax program-\n');
    fprintf('Tau must be greater than zero\n');
    a=0;
    return
end

Temp=exp(EstQ./tau);
S1=sum(Temp);
Probs=Temp./S1;  %These are the probabilities we'll use
```

```
%Select a machine using the probabilities we just computed.
x=rand;
TotalBins=histc(x,[0,cumsum(Probs)']);
a=find(TotalBins==1);
```

Here is a driver for the softmax algorithm. Note the implementation details (e.g., how the "actual" payoffs are calculated, and what the initial and final parameter values are):

```
%Script file to run the N-armed bandit using the softmax strategy

%Initializations are Here:
NumMachines=10;
ActQ=randn(NumMachines,1);  %10 machines
NumPlay=1000;               %Play 100 times
Initialtau=10;              %Initial tau ("High in beginning")
Endingtau=0.5;
tau=10;
NumPlayed=zeros(NumMachines,1);  %Keep a running sum of the number
                                 %  of times each action is selected
ValPlayed=zeros(NumMachines,1);  %Keep a running sum of the total
                                 %  reward for each action
EstQ=zeros(NumMachines,1);
PayoffHistory=zeros(NumPlay,1);  %Keep a record of our payoffs

for i=1:NumPlay

    %Pick a machine to play:
    a=softmax(EstQ,tau);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    PayoffHistory(i)=Payoff;
    tau=Initialtau*(Endingtau/Initialtau)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Here is the function implementing the pursuit strategy (or "Win-Stay, Lose-Shift").

```
function [a, P]=winstay(EstQ,P,beta)
% function [a,P]=winstay(EstQ,P,beta)
%  Input:  EstQ, Estimated values of the payoffs
%          P = Probabilities of playing each machine
%          beta= parameter to adjust the probabilities, between 0 and 1
%  Output:  a = Which machine to play
%           P = Probabilities for each machine

[vals,idx]=max(EstQ);
```

```
winner=idx(1);  %Index of our "winning" machine

%Update the probabilities.  We need to do P(winner) separately.
NumMachines=length(P);
P(winner)=P(winner)+beta*(1-P(winner));

Temp=1:NumMachines;
Temp(winner)=[];  %Temp now holds the indices of all "losers"
P(Temp)=(1-beta)*P(Temp);

%Probabilities are all updated- Choose machine a w/prob P(a)
x=rand;
TotalBins=histc(x,[0,cumsum(P)']);
a=find(TotalBins==1);
```

And its corresponding driver is below. Again, be sure to read and understand what each line of the code does:

```
%Script file to run the N-armed bandit using pursuit strategy

%Initializations
NumMachines=10;
ActQ=randn(NumMachines,1);
NumPlay=2000;
Initialbeta=0.01;
Endingbeta=0.001;
beta=Initialbeta;
NumPlayed=zeros(NumMachines,1);
ValPlayed=zeros(NumMachines,1);
EstQ=zeros(NumMachines,1);
Probs=(1/NumMachines)*ones(10,1);

for i=1:NumPlay

    %Pick a machine to play:
    [a,Probs]=winstay(EstQ,Probs,beta);

    %Play the machine and update EstQ, tau
    Payoff=randn+ActQ(a);
    NumPlayed(a)=NumPlayed(a)+1;
    ValPlayed(a)=ValPlayed(a)+Payoff;
    EstQ(a)=ValPlayed(a)/NumPlayed(a);
    beta=Initialbeta*(Endingbeta/Initialbeta)^(i/NumPlay);
end
[v,winningmachine]=max(ActQ);
winningmachine
NumPlayed
plot(1:10,ActQ,'k',1:10,EstQ,'r')
```

Homework: Implement these 4 pieces of code into Matlab, and comment on the performance of each. You might try changing the initial and final values of the parameters to see if the algorithms are *stable* to these changes. As you form your comments, recall our two competing goals for these algorithms:

- Estimate the values of the actual payoffs (more accurately, the mean payout for each machine).

- Maximize our rewards!

1.2.6 A Summary of Reinforcement Learning

We looked in depth at a basic problem of unsupervised learning- That of trying to find the best winning slot machine in a bank of many. This problem was unsupervised because, although we got rewards or punishments by winning or losing money, we did not know at the beginning of the problem what those payoffs would be. That is, there was no expert available to tell us if we were doing something correctly or not, *we had to infer correct behavior from directly playing the machines.*

We also saw that to solve this problem, we had to do a lot of *trial and error* learning- that's typical in unsupervised learning. Because an expert is not there to tell us the operating parameters, we have to spend time exploring the possibilities.

We learned some techniques for translating learning theory into mathematics, and in the process, we learned some commands in Matlab. We don't expect you to be an expert programmer - this should be a fairly gentle introduction to programming. At this stage, you should be able to read some Matlab code and interpret the output of an algorithm. Later on, we'll give you more opportunities to produce your own pieces of code.

In summary, we looked at the greedy algorithm, the $\epsilon-$greedy algorithm, the softmax strategy, and the pursuit strategy. You might consider how closely (if at all) these algorithms would reproduce human or animal behavior if given the same task.

There are many more topics in Reinforcement Learning to consider, we presented only a short introduction to the topic.