Unit 4
Reinforcement Learning

Context:

The techniques we have applied up to this point have relied on variations of search – to to things that range from path finding and scheduling to playing competitive adversarial games. We will now turn to a different class of AI techniques. These are concerned with training agents to perform specific tasks by repetition coupled with a system that rewards the agent for carrying out the right actions to solve the task, and penalties when the agent's actions move the agent away from the goal or are in some way detrimental to the agent's performance.

This class of methods is known as reinforcement learning, and they comprise a wide variety of methods within AI. Indeed, in a sense, neural networks and their most advanced deep learning versions are at their heart reinforcement learning machines.

It should be noted that in this section we will only be looking at a specific class of reinforcement learning methods, this is by no means the whole of RL, nor does it represent the most advanced methods – but it is a solid introduction that contains many of the relevant ideas that make RL work in its different formulations.

Problem definition

The kind of problems we will be studying in this section is characterized as follows:

- We do not have a precise definition of the goal, no objective function or heuristic function to guide the agent.
- The agent can observe (measure, get the value of) variables that provide information about the ***state*** of the environment.
- The problem evolves in ***discrete time steps***: At each step, the agent(s) choose(s) an action to perform, then the results of this action are observed.
- Based on the outcome, the agent receives a ***reward*** (if the action helped the agent achieve its goal), or a ***penalty*** (if the action was detrimental to the agent in achieving its goal)

The specific goal need never be communicated to the agent, all the agent receives is the ***reinforcement signal*** that indicates whether the outcome of an action was positive or detrimental. The procedure will involve allowing the agent to ***train*** by attempting to solve the given task repeatedly, accumulating statistics that relate specific states and actions to rewards.

Assumptions:

There is a set of ***state variables*** that fully describe the state of the problem being solved. The ***state space*** corresponds to all possible combinations of values for these variables. You will notice that this is only manageable if the variables themselves have a limited set of possible values. ***A specific combination of values for the state variables uniquely defines a specific state for the problem.***

Problem setup: At each time step -

- The agent receives or measures the values of the ***state variables,*** this identifies the state ***s*** for the problem.
- The agent chooses and carries out an action ***a*** that results in a change of the value(s) of state variables. This results in a new state ***s'***.
- The agent receives a ***reinforcement signal r,*** as a result of the combination ***<s,a,s'>*** the ***reinforcement signal*** will be positive if the new state is ***better*** for the agent than the previous one, and negative otherwise. The magnitude of ***r*** can represent the amout of change for the better or for the worse.

Model:

We will model the problem described above as a ***Markov Decision Process (MDP)***. An ***MDP*** consists of:

- A set of states ***S*** that contains all possible states specified by the combinations of values of the state variables.
- A set of actions ***A*** that defines the possible ways in which the agent can effect a change in the state of the problem.
- A reinforcement signal ***r***, with a specified domain.

The agent's job is to find a ***policy*** $\prod$ ***mapping states to optimal actions***. We assume that state transitions are non-deterministic. The last component of an ***MDP*** is a ***transition function T(s,a,s')*** that specifies, for every state ***s*** and action ***a***, the probability that the resulting state will be ***s'***. State transitions are assumed to be non-deterministic due to uncertainty in the environment, real world behaviour (agent choices do not always have the intended affect in real life), and hidden variables that are either not observable, or not included in our model of the problem.

Importantly, ***we will assume the environment is stationary***. This is important because dynamic environments have to be represented by larger sets of state variables and quickly become impossible to handle computationally.

Key principle

The fundamental principle behind this reinforcement learning framework is that the agent's ultimate goal is to ***maximize the expected sum of rewards over time*** that result from the actions chosen by the agent at each time step. Specifically, our goal will be to maximize:

$$E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right], \gamma \in (0,1)$$

This is an important formula – it summarizes the goal of the agent in the context of MDPs. The constant γ serves to discount the rewards the agent receives over time – future rewards (for larger values of *t*) are worth less – so in effect the infinite sum above becomes manageable.

The expectation ***E[x]*** is required because the state transitions are non-deterministic, so the reward we need to accumulate is the ***expected*** or ***average*** reward for each state/action combination.

Learning Framework

Given the set of states ***S –*** estimate a quantity $V^*(s)$ defined as

$V^*(s) = \max_\Pi E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$

this is ***the expected reward of starting at state s and following the optimal policy.*** States with high value are considered good for the agent, because if the agent finds itself at any of those states, the agent can expect to receive a large reward by acting in an optimal way. States with a large negative value are considered bad for the agent – if the agent finds itself there, the future is bleak even if the agent takes the best possible actions.

The value for a state can be rewritten as

$V^*(s) = max_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right), \forall s \in S$

The above is a neat trick – it allows us to replace the infinite sum with two terms, one that is the reward obtained by taking action ***a*** from state ***s***, and one that is the discounted ***expected reward*** due to the ***possible resulting states s'*** that could result from action ***a*** and initial state ***s***.

Example:

Suppose we have the following very tiny problem. ***S*** has 10 states, the values for the states are shown below. The agent has four possible actions, numbered 0-3.

$V^*(0)=15$
$V^*(1)=-4$
$V^*(2)=7$
$V^*(3)=12$
$V^*(4)=-8$
$V^*(5)=3$
$V^*(6)=0$
$V^*(7)=10$
$V^*(8)=-1$
$V^*(9)=5$

Suppose the agent is at state ***s=2***, it takes an action ***a=3*** that results in reward ***R(2,3)=5***, and the resulting states can be 3 (with T(2,3,3)=.25), 4 (with T(2,3,4)=.25) and 7 (with T(2,3,7)=.5). The equation above would result in:

$R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s') = 5 + \gamma * (12 * .25 - 8 * .25 + 10 * .5)$

The definition of the optimal policy follows from the definition of the value of a state. They are complementary – the value of a state is the expected long-term reward of taking the optimal action starting from that state. The policy is simply the ***index*** that identifies the optimal action for that state:

$$\Pi^*(s) = argmax_a \left(R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V^*(s')\right), \forall s \in S$$

Computing optimal state values

The process of estimating $V^*(s)$ is in fact fairly straightforward, though it is computationally intensive.

```
For all s in S,

      set V(s)=0

      for all a in A,

         set Q(s,a)=0

Repeat until policy is good enough

   for all s in S
   {
      for all a in A
      {
         Q(s,a) = R(s,a) + γ Σ_s' T(s,a,s')V*(s')
      }
      V(s) = max_a Q(s,a)
      Π(s) = argmax_a Q(s,a)
   }
```

At the end of this process, V(s) approximates the optimal value for state s, and the policy corresponds to the optimal action that should be taken from each state to maximize rewards. This method is called ***value iteration***, and converges over time to the optimal value and policy for each state. Each iteration has a complexity of $O(|A||S|^2)$ – this is because of the sum over states ***s'*** in the inner loop.

The definition of ***good enough*** for the policy is problem dependent, and involves evaluating the performance of the agent on the given task – when the agent performs at an acceptable level, the process can be stopped.

Despite its simplicity, the process suffers from its computational cost, and from the need to know in advance the transition function ***T(s,a,s')***. The transition function may be unknown or simply too large to represent in memory. There is an alternate learning algorithm that can be used to estimate the value for states, and the optimal policy.

Q-Learning

The Q-Learning algorithm does not require knowledge of the transition table, or the transition probabilities between states. Instead, it relies on the training process to allow the algorithm to *explore* possible outcomes of different actions taken from each state ***s***, and to arrive at a good policy over time.

The goal of Q-Learning is to estimate the quantity ***Q*(s,a)***, this is the expected reward that results from taking action ***a*** from state ***s***. The Q-Learning method is summarized below:

```
// Q-Learning - approximates Q*(s,a), V*(s), and the policy for the agent

Initialize the table Q(s,a)=0   for all s and a
Initialize Pi(s)=random action  for all s     // Chosen from all valid actions
Determine initial state s for the agent
Set alpha, the learning rate, to a small value (problem dependent!)

For j in 1 to K      // Perform K rounds of training
{
    p_random = 1-(j/K)       // Starts close to 1.0,
                             // decreases toward 0 as j approaches K
    For i in 1 to M          // Each round consists of M training actions
    {
        Draw a random number c in [0,1]

        if c <= p_random
                Choose a random action a from the set of valid actions at state s
        else
                Choose a to be the current known optimal action in Pi(s)

        // As a result of the chosen action a the agent will
        // - Receive an instantaneous reward value r
        // - The state will change to s' - this is non-deterministic, we don't
        //   need to know what the probability of this happening is, we are
        //   just told (or the agent can measure the state variables and figure
        //   this out) the resulting state s'.
         //
        // This allows the agent to build an experience tuple  <s,a,r,s'>
        // Update Q(s,a) as follows:

        Q(s,a)+= α [r + (γ max_a' Q(s',a')) - Q(s,a)]
    }
    Pi(s)=argmax_a Q(s,a)        // Update policy at the end of the round!
}
```

The algorithm above relies on two key processes:

- Random exploration of the state space, by allowing the agent to choose random actions given their current state, and updating the Q table based on the result of these actions

- Lots of training trials (each trial corresponds to a single action taken by the agent). We will need a large number of training trials to allow the agent to properly explore the state-space. Remember transitions are non-deterministic, and we need the agent to have explored each state many times in

order to have a chance of discovering as many possible outcomes as we can given the available actions at each state.

- The parameter p_random is important, it controls the agent's behaviour. At the start of the training process, this value is close to 1.0, and the agent is choosing random actions all the time (notice we initialized the policy Pi(s) to also contain random actions for each state). This allows the agent to widely explore the state space.

- After each ***round*** of trials, the policy is updated with whatever action has (thus far) proven to yield the highest reward from each state.

- As the round number approached ***K***, the value of p_random decreases toward 0. This means the agent is choosing more and more often the action that the current policy dictates is optimal at each state. This is critical – it allows the agent to ***refine*** a good policy by mostly using the policy but inserting random actions every now and then in the hopes of discovering an even better one.

- The ***learning rate*** is important, and should be tuned carefully for each problem. Roughly speaking, if this is too small the process will take a very long time to converge to a good policy. If it is too large, the process may not converge properly. A reasonable starting point is a value in [.01, .1] – but this is only a guideline.

Given reasonable values for the learning rate ***alpha*** and the future rewards discount rate ***gamma*** the Q-learning process will produce a good policy. Depending on the size of the state space and the number of actions available to the agent, training may take a long time – but this can be done offline. A caveat of the process is that we depending on the problem, we may have to simulate the training process entirely – it is not possible to train a robot to cross the street by having the actual robot explore random actions millions of times (likely, the first attempt will result on the robot being run over – end of training!).

Q-learning allows us to come up with a good policy on reasonably complex problems, and without having to worry about state transition probabilities. Even so, for many problems the state space is so large that it becomes impractical to carry out the amount of training that would allows us to find a good policy – indeed, for some problems the state space is so large we can't even store the table Q(s,a).

An additional problem is that because of the dependence of the process on ***specific state/action*** combinations, the resulting policy ***does not generalize well***.

Suppose you trained a robot to move around a room with certain furniture and objects in it. We could do this using Q-learning under the assumption that the furniture and objects don't change position much – but our policy will be specific to the room and configuration we trained on. Move the robot to a different room, and the policy is not much use.

One option is to add variables to the state space so we can specify the possible configurations for changing elements of the environment, but this makes the state space much, much larger and thus results in the training process becoming impractical and eventually impossible.

A better approach to handling larger problems, and to achieve some amount of generalization, is to get rid of our dependence on specific states.

Feature Based Q-Learning

This is a variation of the Q-learning algorithm intended to help us deal with problems that have a larger state space than can comfortably be handled with standard Q-learning, and also to allow us to build a policy that is more likely to generalize to ***similar*** problems than the one we trained on.

The idea is to try to learn ***abstracted features*** from a set of configurations seen in training, and apply what we learned to never-seen configurations that have ***similar abstracted features.***

The process relies on the extraction of ***useful features*** from the state variables – what exactly is a useful feature is, of course, problem dependent, and the key problem in implementing this type of learning process. As an example, consider the problem of the mouse in the maze full of cats, trying to find and eat a chunk of cheese.

Useful features may be:

- Some form of distance to a cat (or cats)
- Some form of distance to the cheese
- Some form of information regarding the configuration of the maze around the mouse.

Notice that none of these are specific to a particular configuration – many similar configurations may have the same features, so a learning process that relies on them instead on the specific locations of agents and walls will likely be able to handle variations and to be more useful in allowing the mouse to deal with a new maze, or with changes in the number of cats and/or cheese from whatever it trained on.

Designing the features to be used for a given problem is the harder task we need to solve. Given the set of features, the training process is straightforward:

The Q-table is replaced by a function

$$Q(s) = \sum_{i=1}^{k} w_i f_i(s) = w_1 f1(s) + w_2 f_2(s) + \ldots + w_k f_k(s)$$

This function evaluates the features f_i at state s, each of them multiplied by a weight w_i, and allows us to estimate whether being at state s is a good thing or a bad thing for the agent.

The goal of the training process is to ***estimate the weights*** for the features so that Q(s) does indeed provide us information about how good or bad a specific state s is – ***importantly, since we don't have a table for all s, we don't care any more about the size of the state-space!*** And our policy will not

depend on specific configurations we trained on, but rather, on the learned weights for features that apply to any state we have encountered, or may encounter in the future.

```
// Feature-based Q-learning

Initialize the table w_i=random in [-.5, .5] with w_i!=0 for all i

For j in 1 to K      // Perform K rounds of training
{
    p_random = 1-(j/K)       // Starts close to 1.0,
                             // decreases toward 0 as j approaches K
    For i in 1 to M          // Each round consists of M training actions
    {
        Draw a random number c in [0,1]

        if c <= p_random
               Choose a random action a from the set of valid actions at state s
        else
               Choose a to be the current known optimal action in Pi(s)

        // As a result of the chosen action a the agent will
        // - Receive an instantaneous reward value r
        // - The state will change to s' - this is non-deterministic, we don't
        //   need to know what the probability of this happening is, we are
        //   just told (or the agent can measure the state variables and figure
        //   this out) the resulting state s'.
         //
        // This allows the agent to build an experience tuple  <s,a,r,s'>
        // Update each w_i as follows:
        w_i+ = α [r + (γ · Q(s')) − Q(s)] f_i(s)

    }

}
```

The process above is a form of gradient descent! It will iteratively approximate the weights that allow the agent to maximize its expected reward. However, notice that the process above does not actually state ***how*** to determine the policy.

In order to figure out the optimal action for a given state ***s*** (and remember this may be a state we never saw before during training) we rely on our features and the function Q(s) with the (now optimized) weights:

```
Given state s
Evaluate each possible action a_i
      Determine the feature values after action a_i has been taken
      Compute Q(s_i) - the resulting state after action a_i
Choose the action a_i which results in the largest Q(s_i)
```

So in effect, the policy is a matter of evaluating how actions will change the feature values, and choosing actions that make those feature values maximize the estimated Q(s) for the resulting state. Note that this may not simple – the change in feature values resulting from a particular action may not be easy to determine. Nevertheless, because the process is intended to generalize well, as long as we

can produce a reasonable estimate of what those features values will be, the agent will do the right thing.

You should be aware of the fact there is a tradeoff in terms of ability to solve very specific problems, or ability to generalize. For problems that can be handled with standard Q-learning, if the task is very well defined, and unchanging, we should expect standard Q-learning to perform better than feature-based. Conversely, as soon as variability is introduced, or the problem size grows, we can expect to see feature-based Q-learning do better than standard Q-learning (assuming the standard method can still be applied and the problem has not grown beyond a manageable size).

This closes our unit on reinforcement learning and Markov Decision Processes. Please bear in mind we have only looked at a small corner of what has been developed to handle MDPs! This is a very wide field of study in A.I. and a huge amount of work has gone into it that we won't be looking at in this course.

For now, keep in mind this is one form of reinforcement learning, we will revisit the idea of training an agent by providing feedback on its performance when we study neural networks.