**Primer on Reinforcement Learning**

**Foundations**

*Reinforcement Learning* (RL) lies between *supervised* and *unsupervised* learning. In supervised learning, the algorithm has the correct answers (labels or values associated with each data instance) and seeks to learn patterns in the data to reproduce those answers. In unsupervised learning, the algorithm recieves unlabeled data and seeks to, depending on the specific structure of the algorithm, determine its own way of differentiating the data. In RL, the algorithm, known as an *agent*, is placed in an environment with a goal and a reward/penalty associated with its actions. It learns how to maneuver in its environment by the structure of rewards and penalties, seeking rewards and avoiding penalties. So, we do not tell it how to move/interact with the environment (as we might not actually know how to do that ourselves), but we do judge its behavior and give it the ability to learn from its mistakes. In this way, after spending enough time in its environment, the hope is that it has learned how to achieve the goal we set out for it.

For this primer, let be a state, i.e., the realization of the environment at a given time . Let be an action the agent can take in state , and let be a reward associated with that action. We use the following general notation for a *Reward Function*, or the reward received at time : [1,2]

We note that in general, the mathematical set-up of rewards has the reward arriving at , as the agent finds itself in state at time , takes an action , then the environment responds by changing to state and giving a reward. So, we can talk about the step to or to , as both appear in the relevant literature.

Value and Policy Functions:

Our goal is to develop a proper *Policy Function*, a set of rules which tell the agent how to act in each state. It is written: [3]

To evaluate our policy function, we need to know the value of a state under said policy. This is the *Value Function*: [3]

We note that it is a function of , the state our agent finds itself in at time , and a functional of the policy that instructs the agent how to act in that state.

Environment:

For many tasks, it is not necessary to know exactly *how* the environment ended up where it is, but only to have an idea of where it will be next. This simplifies the information we need to feed to our agent. In situations where we *do* need to know what the environment was like several time steps into the past, we can develop our model of the environment to subsume past time-steps’ information into the current time step. By doing so, we are modeling our environment with *Markovian dynamics*, where the probability of transitioning into a next state depends on only a few states in the past. This looks like: [4]

When , we have simply that the probability of the next state is determined entirely by the previous one, giving us a *Markov chain*. [4]

Markov Decision Processes (pg 289)

A *Markov Decision Process* (MDP) is defined by: [5]

In this formulation, is the set of all possible states, is the set of all actions realizable for a given state, is the probability of transitioning to state given and (sometimes written as ), and is the set of all possible rewards.

From this we can being talking about the rewards, given by the *Expected Reward Function*: [6]

This is the *expected* value of a random reward received at time from taking an action in the state and transitioning into the state .

We can also write our total, cumulative reward *actually gained* as: [6]

We note here that in the function above, we are talking about the *expected reward* at a time step given where the agent was, what action they took, and where they ended up, whereas in the equation, we have the realized gains or losses. We introduce the factor for two reasons. First, it allows us to calibrate how the agent should prioritize rewards. If the agent needs to seek better rewards sooner, then will be small. A value of means our agent *only* looks to the next time step’s reward. A value of would weight each time step’s reward equally. The second reason to introduce is in case our time horizon is infinite. In that case, we set to ensure that our sum converges. [6]

In much the same way as we created the expected reward function, looking at the expectation of a reward, we can ask for the expected valuation of a state considering how the system will evolve from that state. At time , we can ask for the cumulative reward from then on, given by the *Return Function*: [7]

Now, to find the expected value of a given state at time , , we take the expectation of the cumulative rewards gained from starting in that state and following policy . This is the *State-Value Function*: [8]

Similarly, we can specify a value of simultaneously being in state and taking an action as a first action while following policy for all following actions, as opposed to always following a policy. This is the *Action-Value Function*: [7]

We see that it is essentially the same as the state-value function, but we have an extra specified first action , after which we follow our policy . This is captured by the following relationship: [7]

A nice visual that showcases how we begin in a state , take an action based on policy , receive a reward , and transition to a new state is called a *Backup Diagram*:

Diagram

Description automatically generated

[9]

Bellman Equations:

We notice that when in either of the action-value or state-value functions, we have only the expected value of the first reward and then the expected value of the other time steps. Similarly, with , we can write it as . This allows us to write the two equations as depending on all future evaluations iteratively: [10,11]

We notice that in our action-value function, the state-value shows up, as well. We will fix this soon. So, our evaluation of state depends on our evaluation of , and so on, a concept known as *bootstrapping*.

To find the *optimal* equations, the *Bellman Optimality Equations*, we look for the best policy: [12,13]

Since the only difference between the action-value and state-value functions is that we have a first action in the action-value formulation, if we pick the best action for our first action, then we will have the best value. This leads to the relationship: [12]

From this, we can rewrite the Bellman optimal action-value function as: [12]

We can also rewrite the optimal state-value function as: [14]

Note, we can also write these as *time-independent* by simply removing the index and thinking of our system as evolving over states instead of linearly through time.

Reinforcement Learning is thus primarily focused on finding or approximating the optimal policy, , to maximize our reward (in the form of maximizing our or functions). To do this, we can broadly divide our strategies into *Dynamic Programming*, *Monte Carlo Methods*, or *Temporal Difference Learning*, depending on how much information we have about the environment, or a *model*. [15]

A nice map follows: [16]

Diagram

Description automatically generated

**Model-Based Algorithms**

Dynamic Programming

*Dynamic Programming* (DP) refers to a set of algorithms used to compute optimal policies given a *perfect* model of the environment as a finite *Markov Decision Process* (MDP). Recall the form of an MDP:

In DP, we first begin with *Policy Evaluation*, or evaluating how well each policy does, given our actions and states. Given the finite nature of the MDP, this can, in principle, be done. In the literature, this is often shown as a tabular method, where a value under a policy is assigned to each state-action pair. Thus, we run through and see the consequences for for each policy. [17]

From here, we ask if, though we know the value of an action for a given state, can we switch policies as we move through the states, instead of relying on one policy for all states? We are thus asking if we can choose in state such that . This is *Policy Improvement*.To check this, we look to see if the following action-value and state-value inequality is true: [18]

If this is so, then it is better to switch to policy once we are in state . This leads us to an important point. We can ask this at *each* state, giving us the following equation: [19]

Essentially, at *each step, we are looking for the best policy in that state*. This is termed *greedy*, because we are not interested in subsequent steps, only the most current one. However, it can be shown that by choosing greedily at each step, we will arrive at the best policy possible (unless our original was optimal).

Finally, we can combine both to arrive at *Policy Iteration*, wherein at each step, we evaluate that policy with , then improve it to a better policy , then evaluate that with , and so on. [20]

**Model Free Methods**

Monte Carlo Methods

To begin, a significant departure from DP and *Monte Carlo* (MC) methods of solving for the optimal policy is that in MC, we *do not assume we perfectly know the environment*. We are missing the from our original MDP formulation. MC methods only require *experience*, or samples of sequences of states, actions, and rewards, often from simulations (hence the moniker *Monte Carlo*). Another difference is that in using MC methods, we talk about *episodes*, or finite sequences of states which must terminate, and updates occur after each episode based on the *average* as opposed to *expected* returns. The types of evaluations and updating described in the DP sections have their counterparts here, so we will not delve into detail about them. [21] Two backup diagrams for simple MC methods of state-value and action-value sampling follows: [22]

Diagram

Description automatically generated

However, there is one important aspect of MC methods that *does* need to be introduced, as it will be of importance for the models used in this paper. Recall the policy improvement algorithm introduced in the DP section. To choose a better policy, we act greedily, choosing the best action in that state under a given policy. However, if we do not have all the state-action-value pairs, we might miss a good action. Another issue is that to generate a sample, the agent follows a policy, but there is no guarantee that all state-action pairs will be visited to evaluate their value. We need a method for our agent to *explore*, to visit state-action pairs that it isn’t likely to, as they might end up being better in the long run. For this, we introduce *-greedy policy*: [23]

With an *-greedy policy*, the agent takes the action with the best -value with probability and any other action with probability . As the agent learns over time, the value of can be reduced to reduce the amount of exploring the agent does.

Temporal Difference Learning:

*Temporal Difference* (TD) learning can be seen as integrating both earlier methods, able to learn from experience without a well-defined model of the environment through sampling, as in MC methods, as well as being able to update estimates using bootstrapping, as in DP. TD is a large and active area of research, with many more methods than can fit in this paper, and so we will move quickly to the model outline that we will use in this paper. [15]

We begin with the *Update Equation for TD*: [24]

This formulation is called . We can also define the *Difference* as: [25]

Then, our update equation above becomes:

Now, we need to mention two kinds of TD algorithms, *On-policy­* and *Off-policy*. On-policy algorithms assume that the policy being used to produce a dataset (while exploring, for example), is also the optimal policy, and the task is therefore to learn the optimal policy function from the generated data. However, off-policy does not assume this, but merely seeks to learn the optimal policy when data is collected from a different (possibly random) policy. [26]

We introduce an on-policy algorithm, *SARSA*, with the objective of contrasting it to the algorithm we will use from now on: [27]

We see that this is the same equation as our TD update equation, but now focusing on actions, as we want to be able to maximize the return by choosing the right actions. In this model, we need and . Hence the name SARSA. With this model, the agent explores with an -greedy policy, producing an action , and then uses it *again* to produce the next action, . [28]

Now, we move to the last part of this primer, *Q-Learning*: [29]

We see the only difference between SARSA and Q-Learning is the inclusion of the operator when choosing the action to take at the next state. However, this means that, though our agent might explore, if that exploration is not the optimal action to take in that state, the agent will choose something different. From this, the agent can learn an optimal policy from a non-optimal data generation or exploratory policy.