RL with Function Approximation

In past units, we have talked about supervised learning and regression models, where we have a set of features and try to predict an output from these feature labels. This approach has worked well, but it's approaching a ceiling.

ChatGPT for example, has scraped all the textual data it possibly can. Models typically become smarter the more data you feed into them, but this data is finite - there is an upper bound on human information. Thus, current models are reaching a point where they can't be improved much further.

Reinforcement learning (RL) models, however, do not have the same ceiling. Take the game of Blackjack as an example: we have a simulator to mimic the game. If we want to collect more data, we can simply run the simulation longer. Thus, RL is not limited by the amount of data (although it is still limited by compute resources, which are improving with advances like NVIDIA GPUs).

1 Exploration/Exploitation

This concept was first mentioned in Homework 3: GreedySAT, when we explored the tradeoff between always picking the best variable vs picking a random variable. This idea also applies to RL models.

Once we have our optimal Q-function, finding an optimal policy becomes straightforward. Given an optimal $Q^*(s, a)$, our policy $\pi^*(s) = \text{argmax}_a\, Q^*(s, a)$. In this policy, the argmax selects the action with the highest Q-value, which represents the expected return for a state-action pair—the higher the return, the better the action.

If we are working with a value function approach, we don't actually have the Q-function until the very end. We update Q-values as follows:

$$Q(s, a) \leftarrow Q(s_t, a) + \alpha\big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a)\big]$$

The term inside the square brackets is called the temporal-difference error (TD-error) because it represents the difference between predicted and observed rewards. RL models update until we converge on an optimal Q-value. To update our Q-value using the equation above, we need to know 3 things: the current state (s), an action (a), and the state after taking the action (s_{t+1}).

Given this information, how do we actually decide which actions to take? We could use one of the following approaches:

1. argmax$Q(s, a)$ $\rightarrow$ Look at all the Q-values in the Q-table and select the action corresponding to the maximum Q-value. But this is greedy! If we are always doing something greedy, there may be a large part of the state space that we don't explore.
2. Uniform sampling $\rightarrow$ Take a random action. But this may make our algorithm inefficient - our random selection could yield a suboptimal action that leads to poor Q-values. In general, we should try to focus on learning Q-values that are likely to be in our final policy, since it's important that those are accurate.

3. ϵ-greedy: Choose an action randomly with probability ϵ, or choose the action with the highest Q-value with probability $1 - \epsilon$. This balances exploration and exploitation by occasionally choosing random actions while favoring optimal actions to build an optimal policy.

An aside on the Q update equation

We can equivalently rewrite our Q-value update as a convex combination of our current Q-value and the expected new Q-value based on observed returns:

$$Q(s,a) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha\big[r_t + \gamma \max_{a'} Q(s_{t+1},a')\big]$$

This is a simple rearranging of terms in the previous Q-update equation, so it is, of course, an equivalent way of updating Q-values. We could understand the original Q-value update equation as adding a small update in the direction of the temporal difference error. This convex combination equation, we can interpret as a weighted average between the existing $Q(s,a)$ and our new estimate $r_t + \gamma \max_{a'} Q(s_{t+1},a')$. The value of α controls how much we weight towards our new estimate and how much we weight our original estimate $Q(s,a)$.

2 RL + Function Approximation

In the Machine Learning unit, we talked about many approaches for modeling data that utilized function approximation: regression models, neural networks, and decision trees. Function approximation is necessary in RL when dealing with large or continuous state spaces.

For example, in Blackjack, we used Tabular RL, creating a table of all state-action pairs with associated Q-values. This approach works for smaller problems but does not scale well to large state spaces, such as in chess (which has more possible states than atoms in the universe!). This amount of data can not be stored in a typical list or dictionary - instead, we can approximate Q-values using functions when we encounter a state-action pair. This logic relies on the idea that similar states should have similar Q-values. Even if Q-values are slightly off, the model should perform fairly well.

Instead of of maintaining a table of values for each possible (state, action) pair, we can instead approximate Q-values with linear regression. For each action, we generate a weight vector that maps each state to a set of features. We can denote this function as $\phi(s)$. Thus, to approximate the Q-value of some (s,a) pair, we use: $\hat{Q}(s,a) = w_a^T \phi(s)$.

In a linear regression, we were able to assess accuracy using Mean-Square Error (or the sum of the squared values of $y_{pred} - \hat{y}$). In RL, we can use the TD-Error. Remember that:

$$Q(s,a) \leftarrow (1-\alpha)Q(s_t,a_t) + \alpha[\text{TD-Error}]$$

According to the Bellman equation, the TD-error should ideally be 0 upon convergence:

$$0 = r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s,a)$$

When this equation is zero for all possible (state, action) pairs, the algorithm has converged to the optimal Q-values. However, during training, the TD-error is usually not exactly equal to 0. To account for this, we can denote the above TD-Error equation as $\delta(s,a)$, a term we aim to minimize. This gives:

$$\delta(s,a) = r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s,a)$$

The above equation mirrors $y_{pred} - \hat{y}$, where the first term represents our target Q-value, which we can denote as $\hat{Q}$, while the right term is our current Q value. Rewriting the above yields:

$$\delta(s,a) = \hat{Q} - Q \quad \text{where} \quad \hat{Q} = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

However, both terms above rely on Q(s,a) - and when we update $\hat{Q}$, our δ term on the left also starts to update. So, in order for the Q term to match our $\hat{Q}$ term, we will take $\delta(s,a)$ as a constant and try to match it to our $\hat{Q}$ value. Thus, we can use the expectation of $\delta(s,a)$ as our loss function:

$$L_a = \mathbb{E}[\delta_{s,a}^2]$$

We then use gradient descent to minimize loss, or the difference between $Q(s,a)$ and its approximation $\hat{Q}$:

$$\begin{aligned}
\nabla_w \, \delta(s,a)^2 &= \nabla_{w_a} (\hat{Q} - Q(s,a))^2 && (1)\\
&= \nabla_{w_a} (\hat{Q} - w_a^T \phi(s))^2 && (2)\\
&= 2 * \nabla_{w_a} (\hat{Q} - w_a^T \phi(s)) * (-\phi(s)) && (3)\\
&= -2\delta_{s,a}\phi(s) && (4)
\end{aligned}$$

To eliminate the coefficient of 2, we can add a $\frac{1}{2}$ term to our loss function such that:

$$L_a = \frac{1}{2}\mathbb{E}[\delta_{s,a}^2]$$
$$\nabla_{w_a} L_a = -\delta_{s,a}\phi(s)$$

Finally, if we choose a learning, α, we can update our weights as follows:

$$w_a \leftarrow w_a + \alpha(\delta_{s,a} \cdot \phi(s))$$