Reinforcement Learning

Reinforcement learning is an unsupervised learning model based on a reward function that rewards desired behaviors and punishes negative ones. The incentives from rewards hopefully makes progress on the desired outcome, which is all done automatically. The model works by iterating over “state”, in which a decision must be made.

Rewards are typically applied at “terminal state”, which is a desired final condition, usually marking the end of a learning outcome. No rewards are given after terminal state is reached. In the meantime, states in between the desired states have relatively low (or zero) rewards, to incentivize the algorithm to continue moving. Combining these low rewards with the time steps, the algorithm has a way to determine if it is in a suboptimal path (or endless loop).

Each step is a function of the state s, action a, reward R, and next state s',

$$algorithm \rightarrow (s, a, R(s), s')$$

This approach is formally known as the Markov Decision Process (MDP). The future only depends on current state and not how the algorithm got there.

▼ Defining the Model

The Return

The return is a way of comparing rewards and determining which ones are better or worse. It is able to account for time steps and weigh according to the application. Sometimes the time steps are incredibly important, and other times they are not.

The return works by summing the rewards R from each step and and weighing them with a discount factor γ, until the terminal state is reached,

$$Return = R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots \text{(until terminal state)}$$

By decreasing the gamma value in each subsequent step, the algorithm becomes a little impatient and is incentivized to finish as fast as possible. The lower the gamma value, the more the algorithm is penalized for taking another time step, because the discount factor perpetually decreases.

This affect has interesting behavior for negative reward systems. The discount factor actually incentivizes the system to push out the negative rewards as far into the future as possible.

Making Decisions with Policies

In reinforcement learning, the first goal is to come up with a policy π that takes input state s and map it to some action a,

$$\pi(s) = a$$

The final goal is to find a policy that gives an action for every possible state, to maximize the return.

▼ Defining the Model Algorithm

The State-action Value Function

The state-action value function $Q(s, a)$ or Q^* gives the return if starting in state s and taking action a once. Then after taking the action once, behave optimally after. So, take whatever actions result in the highest possible return.

How does the algorithm know the optimal behavior to take after, and if it knows, why couldn't it have done this originally? Even though this definition is slightly circular, it is possible to come up with a Q-function before the optimal policy has been defined. This will be discussed in more detail further down.

When creating the policy, the Q-function can be computed from every possible action at all given states. Since states depend on future states, these Q-functions are recursively calculated to cover all possible scenarios. The best possible return is then selected from each state,

$$\max_a Q(s, a)$$

Thus, the best possible action is the one that maximizes the return at that state. This defines the overall policy of the model.

The Bellman Equation

The final piece to putting this model together is to define the equation that compute the Q-function, which is the Bellman equation. Given the current state and action, future state and action, discount factor, and reward for current state,

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a')$$

This equation defines a recursive function that iterates through all states and uses the maximum reward from the following state. From a high level view, it requires the current return and the optimal return from the next step. Once the algorithm reaches a terminal reward, the state-action value function simplifies down to $Q(s,a) = R(s)$, and the rewards can funnel back up the recursive tree.

▼ Random (Stochastic) Environments

Actions do not work in absolutes and are in fact probabilities. In other words, a policy does not always dictate that a specific action should be chosen over the other 100% of the time. Actions are represented as probabilities, much like choices in other machine learning models. A stochastic environment refers to the variable process, where there is some randomness or uncertainty. This randomness is usually due to external factors. For example, say a robot is learning how to walk, and it slips and falls on the ground. It may go back to a previous state or even go to an unintended state.

When working with stochastic environments, the algorithm is more concerned with maximizing the average value E of rewards across many attempts. So instead of running the algorithm once and taking its highest rewards, the algorithm may get hundreds, thousands, millions, etc. of attempts.

$$Return = E\left[R_1 + \gamma R_2 + \gamma^2 R_3 + \ldots (\text{until terminal state})\right]$$

Stochastic environments also modify the Bellman equation. Since future states and actions are random, the average of those is taken too.

$$Q(s,a) = R(s) + \gamma E\left[\max_{a'} Q(s',a')\right]$$

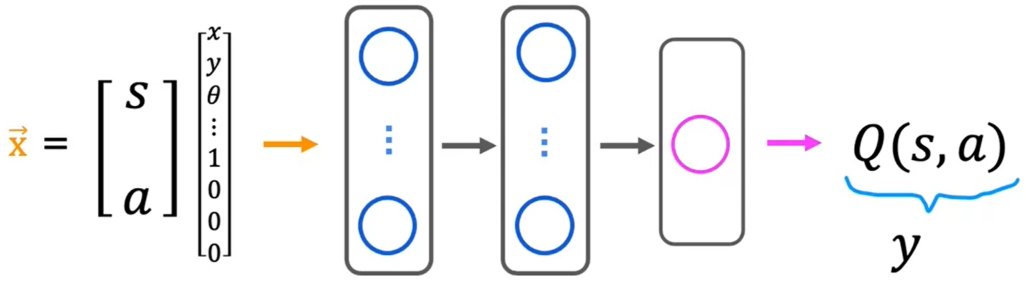
Since randomness can cause non-optimal reward runs, the Q-values returned will not be perfect. Judging how well the rewards returned, relative to an ideal run, can be a good gauge of how efficient the model is.

Continuous State Spaces

So far, assumptions have been made that the system has been in a discrete state. In other words, state was always within a finite number of known positions, which held single values. However, it is possible for state to be continuous. Instead of a single value representing one state, it could be a range of values or a spectrum. This often includes a combination of variables meshed together. For example, if mapping an autonomous car, its position (x and y coordinates), turning radius, velocity, and acceleration could all represent one single state, $s = \begin{bmatrix} x & y & \theta & \dot{x} & \dot{y} & \ddot{x} & \ddot{y} \end{bmatrix}$.

▼ The State-value Function

There is one piece left to complete the model, which is defining an algorithm to pick the right actions. The key to this is the state-value function, which is a deep reinforcement learning technique involving a neural network. The state and actions are fed into the neural network as inputs, and the return (Q-function) is outputted.


$\vec{x} = \begin{bmatrix} s \\ a \end{bmatrix}$
$\begin{bmatrix} x \\ y \\ \theta \\ \vdots \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$
$\underbrace{Q(s,a)}_{y}$

The neural network computes the return for every possible action in a given state and picks the action maximizing the return. So, how does is a neural network trained to output the Q-function? The approach is to use the Bellman equation to create a training set with lots of examples (x, y), then use supervised learning (neural networks) to create the mappings from x to y.

Taking the Bellman equation defined earlier, the input x and output y can be described as,

$$Q(\underbrace{s,a}_{x}) = \underbrace{R(s) + \gamma \max_{a'} Q(s', a')}_{y}$$

And the goal of a neural network is to find some function that (ideally) finds the desired output,

$$f_{w,b}(x) \approx y$$

So, how is a training set created with values x and y for a neural network to learn from? Well, the model can start by trying all various types of states and actions; sometimes randomly, sometimes blindly, sometimes deliberately directed, etc. Each time an action is taken, the relevant data is added to the training set. The state and action are used to compute the input x, and the reward and are used to compute the output y

$$(\underbrace{s^{(i)}, a^{(i)}}_{x^{(i)}}, \underbrace{R(s^{(i)}), {s'}^{(i)}}_{y^{(i)}})$$

The input is pretty straightforward, since state is typically a vector of given values and the actions are one-hot encodings (0 or 1) of whether or not each action was taken,

$$x^{(i)} = (s^{(i)}, a^{(i)})$$

Regarding the output, which is the Bellman equation, the reward is known too and is fed into the equation.

$$y^{(i)} = R(s^{(i)}) + \gamma \max_{a'} Q({s'}^{(i)}, a')$$

Though, one may wonder about the Q-function, as optimal future actions a' are not known yet. Well, initially the result of this Q-function is not known. But it turns out, it is possible to take a totally random guess as a starting point, and the algorithm will work nonetheless.

Now that the training set has been created, the neural network can try to predict y as a function of x. Just like other neural networks discussed, it can use mean squared error or any other type of loss function.

▼ Defining the Learning Algorithm

The first step to putting all of these equations and steps together is to initialize the neural network parameters completely randomly. Initially, pick totally random values for the weights, resulting in a random guess for $Q(s, a)$. This is kind of like initializing parameters w, b completely randomly in linear regression, then using gradient descent to optimize them. What is important here, is if the algorithm can slowly improve upon these initial guesses.

Next, many actions are taken like discussed above. Take actions that are good, bad, random, deliberate, etc. and store them as training data tuples $(s, a, R(s), s')$. The goal is to store the most recent tuples, but the model should be training much more than that. For example, one could store 10,000 examples while having run 500,000 time steps. The reason for this is to avoid excessive memory load, and the values stored are referred to as the “replay buffer”.

The neural network is then trained from these most recent examples in the replay buffer, by converting the tuples to training set data via the state-value function. Like mentioned earlier, the neural network tries to find some function that fits the desired output: $Q_{new}(s, a) = f_{w,b}(x) \approx y$, where Q_{new} simply represents the new Q-function produced by the neural network.

At this point, the model has been trained, but the learning is not finished. It is rare for a reinforcement learning algorithm to get the best return on the first training run, so this whole process is repeated with one caveat - the new return produced by the neural network becomes the starting return for the next one, $Q = Q_{new}$, which eliminates the need for guessing randomly on future iterations.

Summary of Algorithm

- Initialize neural network randomly as a guess of $Q(s, a)$.
- Repeat:
    - Take actions to retrieve training examples $(s, a, R(s), s')$.
        - Store a fraction of these as training data tuples.
    - Train the neural network:
        - Create the training set using the state-value function.
            - $x = (s, a)$ and $y = R(s) + \gamma \max_{a'} Q(s', a')$
        - Train the model to predict desired outputs $Q_{new}(s, a) \approx y$.
    - Set the return produced by model as the initial return for the next model $Q = Q_{new}$.

▼ Refining the Learning Algorithm

Improving the Neural Network Architecture

One way to improve performance is to modify the architecture of the neural network itself. Rather than output a single unit in the output layer (resulting in one single action), it can be more efficient to compute the return for all possible actions; each one being a unit in the output layer. Because all actions need to be computed anyways, it is faster to compute each one and pick the best return, rather than feed them through the neural network.

It turns out it also makes computing parts of the Bellman equation more efficient, specifically $\max_{a'} Q(s', a')$. This is because the return is now being computed for all actions too. So the max value from all actions can be picked here too.

ϵ-greedy Policy

When running reinforcement learning, even though the model is being trained, there still needs to be a mechanism to pick actions while learning. It was mentioned before that this could be some combination of random, direct, good, bad, etc. actions. However, there is a more intentional way to pick actions, which is through the ϵ-greedy policy (epsilon-greedy policy). This is especially important at the beginning of learning, when the model still does not have a decent grasp on what does and doesn’t work.

Taking completely random actions all the time is not a good idea, and also strictly taking the action maximizing the return $Q(s, a)$ is not either. Though, combining these two together actually yields better results:

- **Greedy**: With probability P, pick the action a that maximizes $Q(s, a)$. (e.g. 95%)
- **Exploration**: With probability $(1 - P)$, pick an action a randomly. (e.g. 5%)

Occasionally picking a random action helps the algorithm try out all possibilities, while still being biased towards the best return. Otherwise, depending on how the model is tuned or initialized (remember, the starting return values are chosen at random), it may never try certain actions. Sometimes a model can “believe” that certain actions are always a bad idea, even if they’re not, and randomness can help mitigate this by forcing the algorithm to try those actions.

$$\epsilon = 1 - P$$

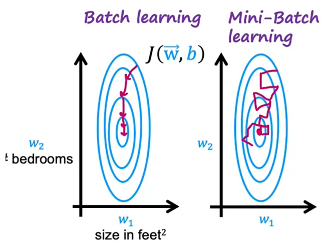
Conventionally, the epsilon value ϵ is referenced at whatever percentage is chosen for exploration. This value usually starts high at the beginning of model training and gradually decreases. Doing so allows the learning algorithm to try out many random actions at first, then gradually become more strict as it has an idea of what actions it should and should not be taking.

One last note - because reinforcement learning algorithms are less mature than supervised learning algorithms, tuning parameters can be increasingly more important. Comparatively, using less than ideal parameters can result in exponential losses in terms of time and computing resources wasted. As a result, hyper-tuning parameters is a crucial step of the process.

Mini-batch

Mini-batches is a process of reducing the size of training examples, when a machine learning model is optimizing its model algorithm (through gradient descent, etc.). As such, this technique can be used for both unsupervised and supervised learning. It's fairly straightforward and involves picking some subset m' of m training examples, with the goal of speeding up computing time. When dealing with massive training sets, it can be a big efficiency boost with negligible performance losses. Each mini-batch takes once unique chunk of the training data, so the entire dataset will be used after all batches have finished. For example, when running gradient descent over 100 million training examples, the subset could have 1,000 examples.

When compared to regular batch learning (using the entire dataset at once), mini-batch tends to be more noisily and not as reliably approach the global minimum of the cost function. However, on average, it tends to do as well and uses less computing resources and time. This has made it preferable for practical applications and a go-to approach in industry.



Going back to reinforcement learning, mini-batch is implemented after the tuples have been stored in the replay buffer. So, the replay buffer is equivalent to the full dataset, and mini-batches are pieces of it.

Soft Updates

In the final step of the reinforcement model training loop, when the Q-function is updated for the next iteration $Q = Q_{new}$, often the update can be abrupt and sometimes even worse than before. Soft updates are implemented to avoid using a potentially worse and noisy Q-function through an unlucky step. They work by fractionally modifying Q_{new} so it still has most of the old Q values. Given parameters w, b in the Q-function and some (small) percentage p,

$$w = (p)w_{new} + (1 - p)w$$
$$b = (p)b_{new} + (1 - p)b$$

This percentage value is typically very small, say 1%, and its weight determines how aggressive the updates are. For example, $w = 1w_{new} + 0w$ is the same as the original function $Q = Q_{new}$, because all weight has been given to the new parameters.

It turns out using the soft update method causes the model to converge on an optimal return more reliably and avoid unwanted properties.