Reinforcement Learning with Hindsight Experience Replay

Or Rivlin

Reinforcement learning has gained a lot of popularity in recent years due some spectacular successes such as defeating the Go world champion and (very recently) winning matches against top professionals in the popular Real time strategy game StarCraft 2. One of the impressive aspects of achievements such as that of AlphaZero (the latest Go playing agent) is that it learns from sparse binary rewards, it either wins or loses the game. Having no intermediate rewards during the episodes makes learning extremely difficult in most cases, as the agent might never actually win, and therefore have no feedback on how to improve its performance. Apparently, games such as Go and StarCraft 2 (at least the way it was played in the matches) have some unique qualities that make it possible to learn with these binary rewards: they are **symmetric zero-sum games**. I am not going to go further into this right now, but I will probably devote a future article to the algorithm behind AlphaZero.

The thing is, most problems are not symmetric zero-sum games, leaving us again with no feedback and no learning. As an example, we can look at a classic RL problem called Mountain-Car. In this problem a car is trying to reach the flag at the top of the mountain, but since it lacks enough acceleration to drive straight up the mountain, it must swing back and forth to gain speed and eventually reach the goal.

As long as the car does not reach the flag, it gets a reward of -1 for each step until the episode terminates after a fixed number of steps. A practical approach used quite often is to augment the reward using domain knowledge, in what is known as **Reward Engineering** or **Reward Shaping**. In our mountain car example, since we know that the car must gather speed to ascend the mountain, a reasonable approach would be to add the velocity of the car to the reward, encouraging it to gather speed. If we shaped the reward carefully enough our agent will gather speed, eventually stumble upon the flag and avoid some of the negative reward it would otherwise have gotten.

The problem with this approach is that it is not always easy to do. In some cases, we might not know how to shape the reward to assist learning; in other words, **we must know something about how to solve the problem in order to properly shape the reward**. This knowledge is not always available, especially for difficult problems. Another danger is that once we engineered the reward, we are no longer directly optimizing for the metric we are really interested in, but instead optimizing a proxy that we

hope will make the learning process easier. This could cause a compromise in performance relative to the true objective, and sometimes even lead to unexpected and unwanted behavior, that might necessitate a frequent fine tuning of the engineered reward in order to get it right.

Another distinct aspect of many famous RL achievements is that these games are concerned with a very specific objective, such as “score as many points as possible in Breakout”. In these problems, the agent observes a **state**, chooses an **action** and gets a **reward**. Our policy in that case can be expressed as:

$$\pi(a|s)$$

Where ‘a’ is the candidate action, and ‘s’ is the current state.

But many real-world problems are not like that, with a single global task we must perform. In many cases we would like our agent to be able to achieve many different goals, such as “fetch the red ball”, but also “fetch the green cube”. I don’t mean to say the agent needs to perform these goals at once somehow, but rather that we would like it to be able to perform any of these tasks upon request. In that case we can express our policy as:

$$\pi(a|s,g)$$

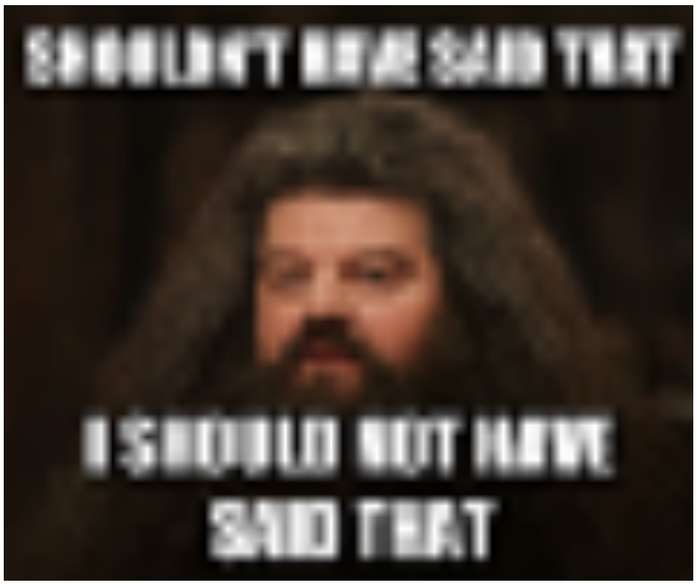
Where ‘g’ is the desired goal. In this article we will treat the goal as a state we would like our agent to reach. This is a multi-goal learning problem. If we combine multi-goal problems with the additional difficulty of sparse binary rewards, we got ourselves into some real difficulty.

In their paper “Hindsight Experience Replay”, researchers from OpenAI give a simple example of such a problem. Suppose we have a state that consists of a vector of binary numbers (0–1) and we wish to train an agent to reach **any** given binary goal vector. The possible actions are to switch a single bit each time, with -1 reward for each time step that the goal has not been reached. obviously if we set the size of the state and goal vectors to be large enough, we would have no hope of solving this problem using conventional methods. The chance of randomly switching bits and somehow stumbling upon the desired goal vector is extremely unlikely, and even using specialized exploration methods (like the one I wrote about in a previous article) we are very likely to fail, since each goal vector is like a completely different problem for these methods. The state-goal space is just too large. The authors demonstrate that with DQN, the maximum solvable vector size is 13, after which the success rate drops sharply to zero.

Source

Off-Policy Learning

But how do humans deal with such problems? Sometimes when we fail to perform some task, we recognize that what we have done could be useful in another context, or for another task. It is this intuition that the authors of the paper (which is one my favorite RL papers) used to develop their method.



Source

In order to translate the capability to inspect past actions and infer useful information from them even though we ultimately failed, we will turn to a learning paradigm that is called **Off-Policy Learning**.

In RL, we try to learn a policy that will maximize the expected cumulative reward given a distribution of initial states. We learn the policy by interacting with the environment through trial and error, and use the data gathered in the process to improve our policy. But some RL algorithms can learn a policy from data that has been gathered by **another policy**. This other policy records its interactions with the

environment and our learning algorithm can use that to infer a potentially better policy. From now on, I will refer to the policy we are trying to learn simply as the policy, and to the other one I will refer to as the **exploration policy**. The purpose of the exploration policy is to explore enough of the state-action space so that our learning algorithm could infer from them what actions should be taken at different states. Classic examples of off-policy learning algorithms are DQN (Deep-Q-Network) and DDPG (Deep Deterministic Policy Gradient), These algorithms can learn the value of state-action pairs given data gather by an exploration policy.

In contrast, **On-Policy** learning algorithms must rely only on data gathered by the same policy that is learnt, which means that they cannot use historical data gathered by another policy, including older versions of themselves (before an SGD update to the neural network for example). Typical examples of on-policy learning algorithms are the various Policy-Gradient methods such as REINFORCE and A3C (Asynchronous Advantage Actor-Critic).

Hindsight Experience Replay

In the famous DQN algorithm, a buffer of past experiences is used to stabilize training by decorrelating the training examples in each batch used to update the neural network. This buffer records past states, the actions taken at those states, the reward received and the next state that was observed. If we wish to extend this to our multi-goal setting, we would have to save the goal in addition to the state, and learn the value of state-goal-action triplets.

As we have seen, the data in the experience replay buffer can originate from an exploration policy, which raises an interesting possibility; **what if we could add fictitious data, by imagining what would happen had the circumstance been different**? This is exactly what Hindsight Experience Replay (HER) does in fact.

In HER, the authors suggest the following strategy: suppose our agent performs an episode of trying to reach goal state G from initial state S, but fails to do so and ends up in some state S' at the end of the episode. We cache the trajectory into our replay buffer:

where **r** with subscript **k** is the reward received at step k of the episode, and **a** with subscript **k** is the action taken at step k of the episode. The idea in HER is to **imagine that our goal has actually been S' all along**, and that in this alternative reality our agent has reached the goal successfully and got the positive reward for doing so. So, in addition to caching the real trajectory as seen before, we also cache the following trajectory:

This trajectory is the imagined one, and is motivated by the human ability to learn useful things from failed attempts. It should also be noted that in the imagined trajectory, the reward received at the final step of the episode is now a positive reward gained from reaching the imagined goal.

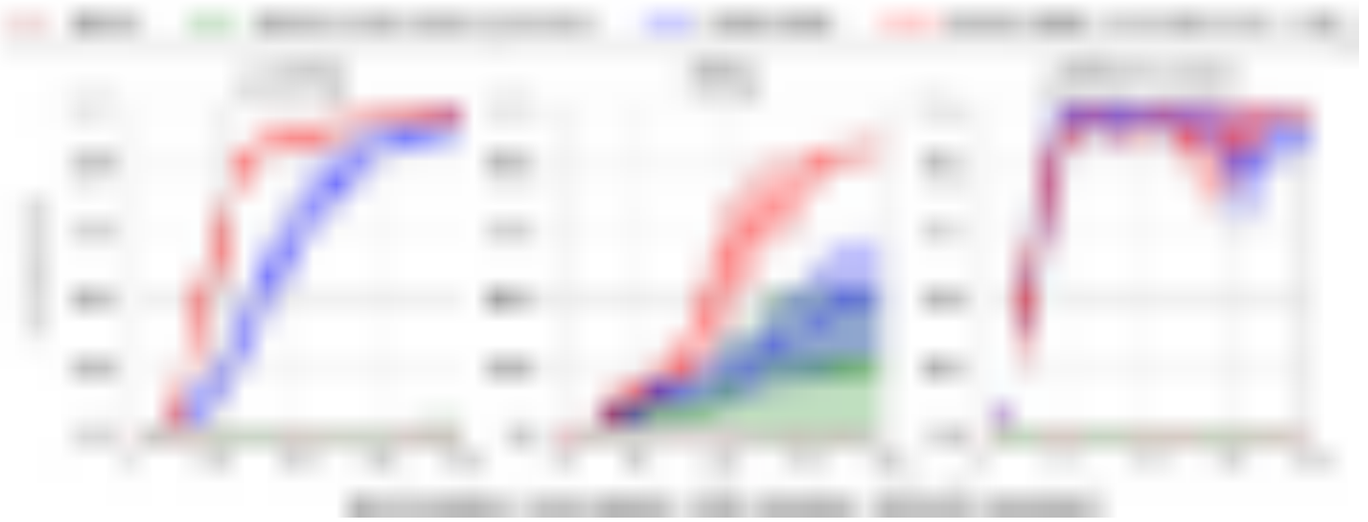
By introducing the imagined trajectories to our replay buffer, we ensure that **no matter how bad our policy is, it will always have some positive rewards to learn from**. At this stage we might ask ourselves what use is there of learning to achieve goals that we have no interest in? clearly in the beginning of training, those imagined goals will be only states that our poor randomly initialized policy could reach, which are of no practical use. However, the magic of function approximation by neural

networks will ensure that our policy could also reach states similar to those it has seen before; this is the generalization property that is the hallmark of successful deep learning. At first, the agent will be able to reach states in a relatively small area around the initial state, but gradually it expands this reachable area of the state space until finally it learns to reach those goal states we are actually interested in.

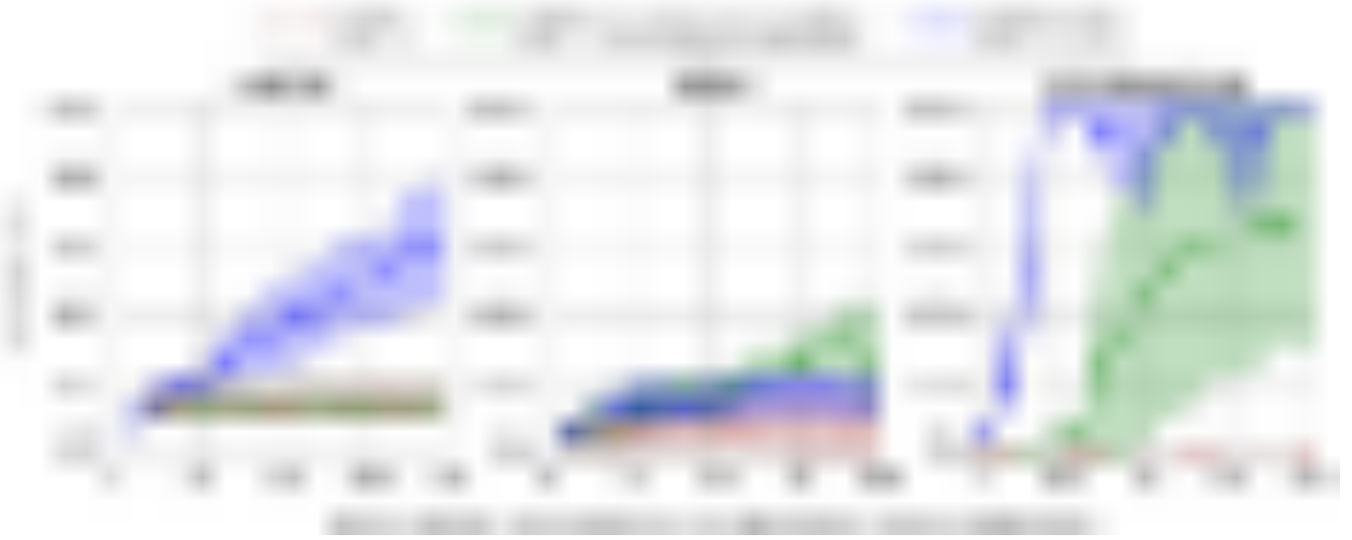
This process bears a lot of resemblance to another common practice in deep learning; curriculum learning. In curriculum learning we want our neural network to learn something difficult, but if we let it train on the real task it will most likely fail. What we can do is let it start training on smaller instances of the problem, that are much easier, and gradually increase the difficulty of the instances until our model learns to perform well on the task we set out to solve. Curriculum learning often works well in practice, but requires the designer to manually engineer the curriculum, and produce easier instances of the task. This is not always easy to do, as sometimes we might not be able to produce an easier version of the problem, and successfully engineering the curriculum might be difficult and time consuming.

In contrast to that, HER gives us a very similar outcome without requiring us to adapt the problem or design a curriculum. We can think of HER as an implicit curriculum learning process, in which we always supply our agent with problems that it is indeed capable of solving, and gradually increase the spectrum of those problems.

The authors tested HER on several robotic manipulation tasks in which the agent must achieve different goals from initial states, such as picking up objects or sliding them to a certain goal position. In these tasks the agent receives a reward if it has completed the task in time, and no reward if it has not. The authors tested HER using DDPG as the base algorithm, and showed that HER succeeds in learning to complete these tasks, where other algorithms fail to learn.



The authors also demonstrated that HER gives improved performance even on tasks where we really care about a specific goal, so long as we can provide other goals for the purpose of training.



This is a very elegant and simple method to tackle a difficult and important problem in many RL applications. Check out the paper. I have implemented the bit flipping experiment using HER, and in addition used HER to train an agent to navigate in a 2D Gridworld environment, feel free to take a look.

Disclaimer: The views expressed in this article are those of the author and do not reflect those of IBM.