Deep Reinforcement Learning

MInDS @ Mines

Reinforcement learning is the subset of machine learning that focuses on training an agent to act in an environment to maximize the reward it receives from the environment. Deep reinforcement learning is the use of neural networks to train an agent to act in an environment. There are a variety of approaches here and we will discuss an overview of the available methods.

Reinforcement Learning

Reinforcement learning involves the training of an agent to act in a way to maximize its reward in an environment. We discussed the general objective in reinforcement learning as represented by the Bellman equations,

$$V^{\pi}(s_t) = \mathsf{E}\left[r(s_t, a_t) + \gamma V^{\pi}\left(s_{t+1}\right)\right], \tag{1}$$
$$Q^{\pi}(s_t, a_t) = \mathsf{E}\left[r(s_t, a_t) + \gamma \mathsf{E}\left[Q^{\pi}\left(s_{t+1}, a_{t+1}\right)\right]\right]. \tag{2}$$

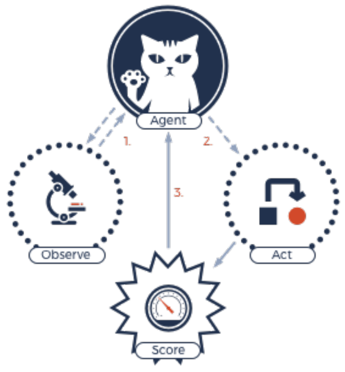

Figure 1: An illustration of the general reinforcement learning process.

where s_t is the state / agent's observations at time t, a_t is the agent's action at time t and γ is the discount rate to determine the relative value of future rewards.

We also discussed the general types of reinforcement learning being,

- Model-free methods: methods that do not utilize a state transition function of the environment.
  - Policy optimization methods: methods that directly optimize the parameters that govern the policy to determine the best action to perform.
  - Q learning methods: methods that approximate the action-value function to determine the best action to perform.
- Model-based methods: methods that model the environment's state transition and utilize that in their policy determination.
  - Learned model: methods that learn the state transition function.
  - Given model: methods that are provided the state transition function.

Deep Q Networks

Deep Q Networks (DQN) are the initial approach to deep reinforcement learning that have catapulted research in deep reinforcement learning. They were developed to play Atari games using the input image from the game and allowing the agent to perform all the actions allowed by the game. Since the network takes input images, it starts off with convolutional layers. The convolutional layers are then followed by fully connected layers which aim to

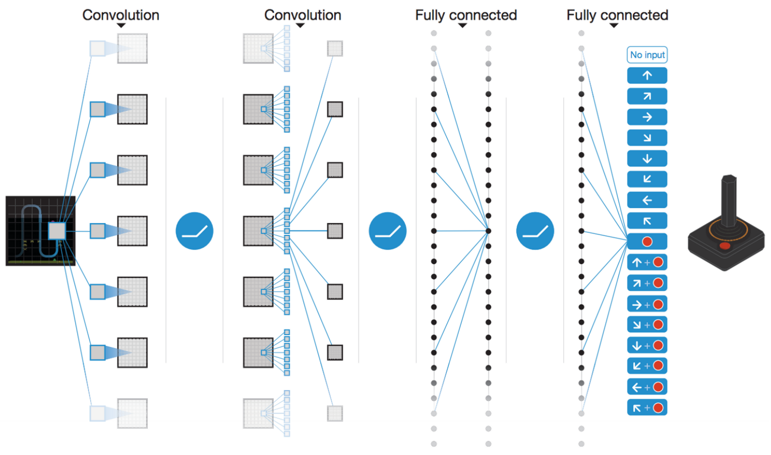

Figure 2: A visualization of Deep Q Networks from the *Nature* paper.

estimate the Q values of all actions performed in that state. You can think of the output of the DQN as being the row in the Q table for that given state.

The architecture of DQN is actually quite simple. The complexity arises in training the model and there are a few important factors about the proposed DQN that make it effective. In general, we train a network by calculating a loss/error function that we want to minimize. In our case, the loss function is a difference between the estimate of the Q value and the true Q value that results from the simulation,

$$L(s,a) = \mathbb{E}\left[\left(\hat{Q}(s,a) - Q(s,a)\right)^2\right], \tag{3}$$

$$= \mathbb{E}\left[\left(r(s_t,a_t) + \gamma \max_{a_{t+1}} Q\left(s_{t+1},a_{t+1}\right) - Q(s,a)\right)^2\right]. \tag{4}$$

In our formulated loss function we use the squared difference but we can use other calculated differences.

To train the model effectively, we need to apply some modifications to the typical training of the network. One key alteration to training this network is called *experience replay*. Experience replay is the process by which the network stores or caches experiences and the resulting values without applying any learning to them then it learns weights for its experiences all at once. The model stores batches of experience that it can learn from all at once instead of using each state and action independently. This helps prevent the model from overfitting.

Another key alteration is using *fixed Q targets*; where the weights are only updated once every f iterations. Since the Q value function of a state and action is affected by the Q value function of the next state, we would be constantly updating the weights that calculate the Q value function. This constance updating results in a highly unstable system that can have some difficulty training the model. By updating the weights after many iterations, the model's training can stabilize and learn from a more holistic experience.

Since the environment can be fairly complex, it would normally have a complex reward structure based on the various states that would exist. A simplified reward structure can allow the network to train more easily. One approach that was applied in the original paper was to clip the rewards to be either 1 or -1. This approach allows the model to be normalized across different game environments so they could use the exact same model structure and hyperparameters across different environments. Despite the fact that this approach doesn't teach the model how to prioritize varying magnitudes of rewards or punishment, the added simplicity in training results in a more effective model.

The final alteration to the typical system is one that focuses on high speed environments with minimal change. Instead of passing in every image frame of the game environment, the game skips k frames and whatever previous action was determined is applied to those frames. This can speed up training and playing drastically.

For an example of how well this neural network can perform, see the original posted video by DeepMind when they published their paper, https://www.youtube.com/watch?v=V1eYniJ0Rnk.

Policy optimization via "Vanilla" Policy Gradient approach

So far we've referred to our policy as simply a lookup of all the estimated values of our actions within a state and a selection of the highest reward. This approach is good when we have a lot of data but with high dimensionality, it can be difficult to obtain values for every single possible state. Instead of just approximating the value, we now try to develop a representation for our policy,

$$\pi_\theta(s,a) = P(a|s,\theta), \tag{5}$$

where θ represents the parameters used to determine our policy. The idea with policy optimization is that we optimize for the approach that we actually care about and ignore other cases. For simplicity, we denote,

$$R(\tau) = \sum_{t=0}^{H} R\left(s_t, u_t\right). \tag{6}$$

Given the optimization function,

$$J(\theta) = \mathsf{E}\left[\sum_{t=0}^{H} R\left(s_t, u_t\right); \pi_\theta\right] = \sum_{\tau} P(\tau;\theta) R(\tau) \tag{7}$$

we can calculate the gradient as,

$$\nabla_\theta J(\theta) = \sum_{\tau} P(\tau;\theta) \nabla_\theta \log P(\tau;\theta) R(\tau) \tag{8}$$

In order to use this in practice we follow these steps:

1. Initialize parameters

2. Run a few test runs with the policy and collect data about the rewards

3. Estimate the policy gradient
4. Update policy using gradient ascent
5. Compute loss function and repeat from step 2

Actor-Critic Algorithm

Another approach to reinforcement learning combines the benefits of both policy optimization and Q learning. The general idea is to train both a policy optimization method and a Q learning method. The policy optimization method is considered the *actor* and the Q learning method is considered the *critic*. The policy selects the actions to take and the critic estimates the Q values for the actions. The Q values can then be used to update the policy directly and we only need to keep track of the values for actions that follow the policy we want to use. This approach combines the advantages from both methods.