A Survey On Model-Free Deep Reinforcement Learning

Akhil Devarashetti
akhil.ai
akhilez.ai@gmail.com

Sunday 24th January, 2021

***Abstract*—Deep Reinforcement Learning is a branch of machine learning techniques that is used to find out the best possible path given a situation. It is an interesting domain of algorithms ranging from basic multi-arm bandit problems to playing complex games like Dota 2. This paper surveys the research work on model-free approaches to deep reinforcement learning like Deep Q Learning, Policy Gradients, Actor-Critic methods and other recent advancements.**

I. Introduction

Reinforcement Learning (RL) is one of the core branches of Artificial Intelligence, first popularized in 1961 by Marwin Minsky's popular paper titled "Steps Towards Artificial Intelligence" [1]. Any algorithm that learns to achieve a task by trial and error can be considered as Reinforcement Learning.

Most RL applications are Game players. It might seem as if RL is not particularly useful to the real world. However RL can solve real world problems, for example, Google applied a RL algorithm to cool down their data center by 40% [2]. Also RL is used in drug development [3]. And it is used in a myriad of control tasks like moving a robotic arm [4]. In a sense, RL algorithms encapsulate a primitive type of general intelligence as it learns to play in complex environments. Our world can also be seen as a highly complex RL environment and we are all agents trying to maximize our personal rewards. So RL algorithms when scaled up could unlock the potential of Artificial General Intelligence [5].

The aim of this study is to outline the significant research works done on different optimizations methods used in model-free RL. We will see how Q Learning algorithm learns to play Atari games in Section 2. Then we will see how Policy Gradient method targets the modeling and optimizing the policy directly in Section 3. Section 4 dives into the combination of Q Learning and Policy Gradients called Actor-Critic Method. In Section 5, we will see further developments in the three ideas discussed in the first three sections and discuss how the improvements were made and which algorithm suits what kind of learning task. Finally, in Section 6, we will briefly discuss other approaches and recent advancements in Deep Reinforcement Learning.

A. The Reinforcement Learning Framework

The main characters in reinforcement learning are the agent and the environment, wherein an agent learns by interacting with the environment it lives in. Here, the environment is the observable world. A state of the environment is a snapshot of the observable world at a given time-step. An agent is the player or the RL algorithm itself. The agent takes an action from the action space in the environment at each time-step, which results in a change in the environment. This action is decided by our algorithm. A reward is received by the agent, which indicates how good or bad the action was. The goal of the agent is to execute the optimal sequence of actions in order to gain maximum rewards.

For example, in Chess, the chessboard and all the pieces are the environment and the player is the agent. Winning a chess game would give a positive reward, and losing a pawn would give a negative reward.

B. The Deep in Deep Reinforcement Learning

The very first RL programs were under the branch of Dynamic Programming [6] [7]. Interestingly however, the early research on RL was actually loosely based on Neural Networks back in 1954 by the Godfather of AI research, Marwin Minsky, along with Farley and Clark [8]. Their work involved building an analog machine with components called SNARCs (Stochastic Neural-Analog Reinforcement Calculators) which learns by trial-and-error. Even Rosenblatt, the pioneer of neural networks used terms like "rewards" and "punishments" indicating that he was inspired by RL. Today, the most powerful Reinforcement Learning algorithms rely on Neural Networks. In fact, Deep Learning has revolutionized many fields since 2011. After DeepMind's Atari paper in 2013 [9], numerous Reinforcement Learning approaches were published.

A neural network is a large-scale differentiable mathematical function with many coefficients or weights that can be tuned in many iterations by a differentiation algorithm called back-propagation. A typical neural network performs a series of operations segmented in "layers" where each layer typically takes an n dimensional matrix as input, applies matrix multiplication, addition and a non-linearity function to the input and outputs a matrix of m dimensions. Each layer has its own parameters or weights which are matrices with which the input gets multiplied. These weights are set to random values initially and the network learns optimal weights over the training period. They learn their weights by calculating a

loss value for each input-output and updates its parameters to minimize this loss value. The learning happens by calculating the partial derivative of the loss value with respect to each of the parameters in the network. Then the derivative is multiplied with a small value between 0-1 called the learning rate and is subtracted from the weight. This process is called gradient descent and the idea of updating the weights layer by layer is called back-propagation [10] All of its parameter tuning happens only by passing in the right data. So these neural networks act as complex function approximators. They learn sophisticated patterns in data which may not be feasible to learn by traditional machine learning algorithms. This is the reason why Deep Learning brought a revolution in the field of RL.

C. Taxonomy of RL Algorithms

There are two types of RL algorithms. One is model-based RL and the other is model-free RL [11]. A model-based algorithm tries to "understand" how the world works. It often does this by predicting the next states and rewards and then takes optimal actions. Although model-based algorithms are mostly better than model-free, they tend to easily inject bias in its understanding of the world model.

However, it is not necessary to learn a model to get good enough policy. The agent can instead learn a sequence of actions directly using algorithms like Policy Gradients, Q-learning and Actor-Critic. These algorithms are called model-free RL algorithms and they rely on real samples from the environment and don't try to create a model of the world to find the optimal policy.

There are other types of RL algorithms that don't quite fit into a traditional tree like taxonomy like Hierarchical RL, Exploration and Curiosity based RL and Combination of Monte Carlo Tree Search with Q Learning.

II. Deep Q Learning

The first paper that took off with the Deep Learning revolution is the DeepMind's "Playing Atari with Deep Reinforcement Learning" which learned to play multiple Atari games at a super-human level performance [9]. DeepMind used an algorithm called Deep Q Learning with some smart tricks to learn multiple games with the same model. The model takes in the last 4 frames of the game as raw pixel values and outputs a "value" for each action the agent can take which says how values each action is. This value is also known as the Q value.

$$Q(s) = f(A|s) \tag{1}$$

Where f is a function of set of all actions A given a state s. The model architecture consisted of a series of Convolutional Neural Networks (CNNs) which are type of layers designed to work well with images. Then the output from CNNs is flattened and passed through a series of Fully Connected Layers. The last layer would have 4 or so outputs depending on the number of actions. Each hidden layer has ReLU as the activation or non-linearity function and the last layer uses tanh non-linearity.

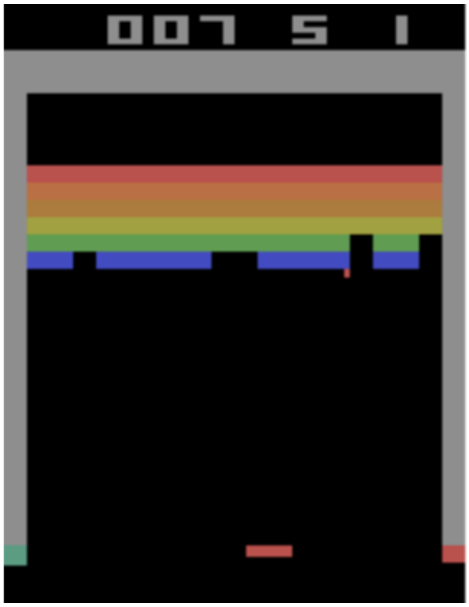

Fig. 1. A screenshot from a famous Atari game called Breakout. The agent moves the paddle left or right and bounce the ball at different angles to smash all the bricks at the top.

To train a network to learn the Q value, we need the correct Q values. The correct Q values are nothing but discounted sum of rewards.

$$Q_t = \sum_{i=t}^{n} \gamma^{i-t} r_i = r_t + \gamma^1 r_{t+1} + \ldots + \gamma^{n-t} r_n \tag{2}$$

Where γ is a small quantity close to 1 like 0.99, r_t is the reward observed at time-step t and n is the length of the episode. These terms are just recursions of itself. Luckily, we don't need to compute the recursion every time. The terms after the first term can be replaced with Q value of the next state. However, we will use the maximum predicted Q value of the next state from out neural network and don't interact with the environment for future rewards. This means that we are using the predicted value of next state to improve the prediction of the value of the current state. This prediction on prediction is called bootstrapping which we will use again in Section 4.

$$Q_t = r_t + \gamma . Q_{t+1} \tag{3}$$

The neural network in the beginning of its training will be initialized randomly. So, it predicts random Q values for each action and learns to predict accurately over the training period. If we decide to only take the action for the highest predicted Q value, we could get stuck in learning. This is because the network might randomly give highest Q value to some sub-optimal action in the beginning. If the agent wins the episode, then the sub-optimal action gets positively reinforced and makes it the highest value for the next episode. This continues and the optimal action might never get a chance to be selected. This is a famous problem of choosing Exploration vs Exploitation.

One solution to this problem is to use an epsilon ϵ probability exploration which this paper has used. We would define a small number between 0 and 1 called ϵ which defines the probability of exploration. This means that with probability ϵ, we take a random action. We take the action with highest Q values rest of the times. In the Atari paper, DeepMind used a decaying ϵ where ϵ starts off as 1 because all actions are randomly valued and with every epoch, ϵ is decreased. Overtime, the network gets better at predicting the Q values, so we take the highest Q valued action - exploitation.

There is one more problem that the Atari paper has solved. The agent quickly forgets what it has learned. Because the learning is completely on-line i.e. the learning happens right after taking an action, the gradients oscillate and produce too much of noise. For example, lets take an environment where there is a pit in some random direction (left or right) around the agent. The agent has to figure out where this pit is located and move opposite to that. If in the first episode, the pit is located in the left, and the agent moves left, it loses the game. Here the parameters get updated so that going left is penalized and learns to go to the right. However, if in the next episode, the pit is located in the right and if the agent chooses to the right, the agent loses the game again. This time, the gradients get reset back to the original because now going right is penalized and the agent learns to go left. Overall, the agent did not learn anything. It forgot what it has learned. This is generally not how supervised algorithms learn. In supervised algorithms, the learning happens in batches. The gradients add up in the batch and an overall gradient update is made at once. This is taken as motivation and used in this framework. The researchers let the agent run in the environment for some number of episodes and all the transitions are recorded in an experience buffer. Then a random subset of action samples from the experience buffer is taken and trained upon. After training, they let the agent run in the environment again and the experiences are queued into the experience buffer. In practice, as the queue size is limited, old experiences are deleted to accommodate new experiences. This way, a batch learning scheme was applied in the Atari paper. This approach of training on old experiences is termed as experience replay.

III. Policy Gradients

Instead of dealing with the complexity of value functions of state action pairs, Policy Gradient algorithms simply learn to predict the action to take or a probability distribution over the set of actions. One of the first policy gradient algorithms based on neural networks is REINFORCE [12] by Ronald J Williams and it was practically put in terms of deep learning framework by Sutton et all [13]. A policy is a function that maps states to the best actions. It is the plan of action to be taken to gain the most rewards. This algorithm involved a neural network where the input is the state of the environment and the output is a probability distribution of all the actions that the agent could take. The network could also output real valued actions directly like seen in [14] which we'll see in the next section. An action is sampled with the probability distribution ie action with highest probability will be picked the most. Since the weights of the neural network are set to random initially, the agent would take random actions. Occasionally, the agent gains a reward, say by winning the game. Then the actions that lead up to the winning action are considered as "good" actions and we reinforce them. However, not all actions in the episode lead to the win, some of them might be bad, but on overall, most of the actions lead to the winning of the episode. This is the principle behind the REINFORCE algorithm - although there are bad actions, in a long term, good actions get reinforced. However, not all actions are given the same weight. In environments like balancing a pole on a cart, the actions that lead to the fall of the pole are at the end of the episode. These ending actions must be given less credit or ignored because they didn't help us win the episode. In this case, the actions in the beginning of the episode must be given higher credit and each succeeding action gets exponentially lower credit. This might be opposite in the environments like playing tic-tac-toe where the last move is the most significant one in which case, the actions at the end of the episode gets the highest credit. This problem of crediting the actions is called the problem of credit assignment.

To train the weights of the model, we need to compute some loss value out of the probability distribution and then minimize that value. Consider multiplying the probability value with the reward and summing them all for an episode. Now we want to maximize this value. However, neural networks work well with loss that minimizes. So we will multiply this value with -1 and minimize it. Now, out loss function becomes:

$$J = -\sum_{t} \pi(s_t, a_t).r_t.c_t \tag{4}$$

Where J is the loss, $\pi(s_t, a_t)$ is the predicted probability value for the state s and action a at time-step t. r_t is the reward observed and c_t is the credit assigned to that time-step. This loss works, but there it can be improved to train better. Instead of using the immediate reward only, we could use return. A return is a discounted sum of future rewards.

$$R_t = \sum_{t} \gamma^{t-1} \cdot r_t \tag{5}$$

Here γ is a small value like 0.99 and t is the time-step. We can also change it to use $log(\pi(s_t, a_t))$ instead of $\pi(s_t, a_t)$. When a probability value close to 1, the log value is close to 0 and when the probability value is close to 0, the log probability value is close to negative infinity. This gives a nice scaled value to the network to learn without which we would need a lot of precision points to store small probability values. Now our loss function becomes

$$J = -\sum_{t} log(\pi(s_t, a_t)).R_t.c_t \tag{6}$$

IV. Actor-Critic

Policy Gradient algorithms is usually better or equally as good as Deep Q Learning. However, the main disadvantage of Policy Gradient is that it requires a complete run of an

episode to determine the loss and backpropagate. This is also called Monte Carlo method of Reinforcement Learning. This gets challenging really quick when we use it for long running episodes like in the game of chess. The reward may be gained so far into the future that any action in the beginning makes equal sense. Moreover, the value networks can potentially be highly biased, but with low variance. On the other hand, the advantage of DQL is that it can be trained in a truly online fashion and they can be less biased, but can potentially have high variance. It makes sense to combine the best of Policy Gradients and the Q Learning method.

The Actor-Critic paper by Mnih et al (A3C) does just that [15]. It consists of two networks. A policy network that predicts the policy or the action probabilities. A value network which predicts the state values. Note that in the Deep Q Learning above, we discussed a neural network that predicts state-action values, but here we predict state value. This is because the action is already selected by the policy network, so we don't need values for each action, but only state. We could however use state-action values, but it is usually not used as it takes longer to train.

In our policy gradient loss function, we used the return R to multiply with the log probabilities. We use the same approach here, but instead of using just the return values, we will use use $(R_t - V(s_t))$ where $V(s_t)$ is the value of state s at time step t that is predicted by the value network. This new term is called Advantage. The value network, as usual tries to predict the expected discounted sum of rewards. And the term $(R_t - V(s_t))$ will give us the value of how much (more or less) rewards did we actually observe compared the rewards we expected. This is why the term Advantage comes in the name of A2C - Advantage Actor-Critic. For example, for a state if we predict the state value as +4, that means that on average, we got reward as 4 for that particular state. And if the actual reward on taking a predicted action (from actor network) is, say +10, then we beat our expectation by 10 - 4 = +6. This means that the predicted action led the agent to gain 6 additional rewards than usual. So the action must be reinforced. This is what advantage term does to the training behaviour - learn by beating the expected rewards. At the same time, the value network learns that the value of the state is not 4, but 10. This makes the value function predict accurate rewards, trying to match what the actor would do. This brings an adversarial relationship between the actor and the critic which is the key idea behind the Actor-Critic method.

When it comes to online vs Monte-Carlo training, the A3C paper introduces an n-step online training. They say that in a completely online training, the value predictions by the value network are of high variance. However, given an n time-steps (less than the length of the episode), the Q network can predict better state value for the (n+1)th state. This is the idea behind n-step online training.

For the first n time-steps, the predicted state values, rewards, and predicted actions are stored in a buffer along with one additional value - the predicted state value of (n+1)th state. This (n+1)th state value is used as the discounted sum of rewards for all the states after n time-step when calculating the returns R.

$$R = r_1 + \gamma.r_2 + \gamma^2 r_3 + \ldots + \gamma^n + \hat{V}(s_{n+1}) \tag{7}$$

Here $\hat{V}(s_{n+1})$ is the predicted value of the state s at time-step $n+1$ We are essentially bootstrapping the policy network using the prediction from the value network. This seems to smoothen the training loss and also help train as fast as twice the speed of Monte-Carlo training.

In practice, we do not use two different neural networks for actor and critic. We use a single Neural Network with two heads. The input would be the state and the neural network gets forked into two different heads where one head outputs the policy and the other predicts the state values. Both the actor and critic share the initial layers of the neural network. The critic's head is usually detached from the computational graph so that its gradients do not flow through the shared layers. This is done because we don't want the actor and critic rub against each other and mess up the initial layers. We want only the actor's gradients to flow through the shared layers for better training speed.

The loss function of the actor is similar to Policy Gradient but with Advantage. (loss function here)

$$J_a = -1 * \gamma_t * (R - v\,(s_t)) * \pi(a \mid s) \tag{8}$$

The critic's loss function is simply the Mean Squared Error of the predicted value and the observed discounted sum of rewards.

$$J_c = (R - v)^2 \tag{9}$$

In the Atari paper, the authors used a technique called experience replay which samples random timesteps form the experience buffer to train. We could use the same principle here as well. However, most Actor Critic approaches use a type of neural network called Recurrent Neural Network (RNNs). RNNs work on sequences of data. It does use batched data, but batches of sequences. So randomly sampling timesteps would require storing the hidden weights or cell states of the RNNs in the experience buffer along with their gradients which is not desirable. So the authors here introduced a training scheme where multiple different environments are setup in parallel. The neural network is shared among all the environments. There would be multiple agents each running in their respective environment and producing different experiences. All of these agents produce the gradients asynchronously which will then be summed or averaged and updated. This is why the term Asynchronous comes into the name of the algorithm A3C - Asynchronous Advantage Actor Critic.

V. Further Developments

A. Prioritized Experience Replay

We have seen that in Deep Q Learning, experience replay was introduced to solve the problem of catastrophic forgetting. A naive way of experience replay is to use random sampling from the experience replay buffer. We could make this better by giving more sampling probability to the experiences that

led to a win. However, if we do this, the network might see the same winning experiences again and again and might overfit the policy to those experiences. This leads to poor performance on varied experiences. Schaul et al has proposed a new and complex technique [16] to solve this problem. For every episode the agent plays, they would store loss value associated with that episode along with the experience itself in the experience buffer. When sampling the experiences for training, the team would assign higher sampling probability to the episodes that had higher loss value. This means that the experiences that the network performed poorly will be trained more. After a while of training, the network becomes good at the episodes that it was poor on. Then the whole experience buffer is refreshed with new loss values and the sampling is done again with new loss values. This proved to be an effective approach to catastrophic forgetting.

B. Distributed Q Learning

An important advancement of DQL is Distributed Q Learning. Bellman's equation states that given a state and an action, the next state is always deterministic. However, in some environments, there may be some random variable that injects randomness into the state transitions. The prediction of the value of a state action becomes difficult if the next state is also dependant on randomness. Thus, to solve this problem, Bellemare et al introduced a concept of Distributional Q Learning. In this algorithm, instead of predicting a single Q value for a state action pair, the model predicts a whole distribution. Different parts of the distribution associates to Q values of different state transitions.

C. Trust Region Policy Optimization

The problem with vanilla policy gradient is that with the gradients take a huge step in the parameter update that the parameters go beyond the curvature of the loss function and end up with unusually high loss. This also happens because unlike supervised learning, the loss function is highly noisy and not smooth or monotonic. Consider a loss function which looks like $y = x^2$ where Y-axis represents the loss and the X-axis represents the weight. When the loss is very high, taking a fairly large step in X-axis towards would yield lower loss. However, as x approaches the origin, taking larger steps is highly risky as the step update goes beyond the origin which is the lowest loss value and overshoots into high loss region. So the loss functions could be quite sensitive and the parameter updates need to be made more carefully. This is the idea behind developing Trust Region Policy Optimization (TRPO) [17]. There are two new things here. (1) Instead of using logarithm of action probabilities in the loss function, this approach uses.

$$J = \mathbb{E}_{a\sim q}\left[\frac{\pi_\theta(a|s_n)}{q(a|s_n)}A_{\theta_{old}}(s_n, a)\right] \tag{10}$$

(2) The weight updates are constrained within their KL divergence value.

$$\mathbb{E}_{s\sim\rho_{\theta_{old}}}\left[D_{KL}(\pi_{\theta_{old}}(.|s)||\pi_\theta(.|s))\right] \leq \delta \tag{11}$$

This means that we always need to maintain a copy of what the previous weights were.

D. Proximal Policy Optimization

The problem with TRPO is that the KL divergence constraint can be an overhead to the algorithm and sometimes can lead to unusual training behavior. So the idea behind Proximal Policy Optimization (PPO) [18] is to include this constraint directly into the loss function. From the loss function of TRPO, the log probability is replaced with $r_t(\theta)$ as follows

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{12}$$

We can see that the ratio $r_t(\theta)$ is greater than 1 if the updates are more likely to be newer than the previous weights. If it is between 0-1, then the updates are likely to be close to the previous weights. Instead of constraining the updates, the idea here is to clip the returns to a smaller values so that the updates don't go too far from the previous weights. The value to be clipped is included in the loss function as below.

$$L^{CLIP}(\theta) = \mathbb{E}_t\left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right] \tag{13}$$

Essentially, the loss value gets flattened out when returns R gets too high, hence flattening the gradient updates. When R is negative, the loss flattens out before R approaches 0 without which the gradients would be too small to have a meaningful change in the weights. Finally, the PPO objective function is much simpler to calculate than finding the complex KL Divergence from TRPO. It is shown that this helps train faster and also outperforms TRPO.

E. Deterministic Policy Gradients

So far we have seen the actions as discrete set of classes. These actions could be like "up", "down", "jump", "shoot" etc. But some RL tasks involve continuous actions. For example, training a robotic arm to control a rubik's cube will involve a lot of continuous actions like rotating the wrist by 20.4 degrees and index finger by 3.9 degrees etc. We cannot use a Q Learning approach here because it excepts the actions to be limited. Otherwise, the model would have to predict a Q value for infinitely large number of actions. We could discretize the actions, but there is no upper or lower bound to the action value. Moreover, if the model needs to produce multiple action values, then the actions could quickly go into thousands in number. So, Q Learning is clearly not the right approach. This is what this paper [14] tries to solve using Policy Gradients. It is similar to the regular Actor-Critic, except that the critic in this case predicts the Q value for the given state and action rather than just the state. It uses separate models for the actor and critic. The states, actions and rewards are randomly sampled, the actions that actor takes are put into the critic network, then the losses and gradients are determined and then the parameters are updated for both models. The loss on the critic is

$$J = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{14}$$

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{t+1}|Q^{\mu'})|\theta^{Q'}) \tag{15}$$

Its gradient on the loss function is as follows.

$$\nabla^{\mu}_{\theta} J = \mathbb{E}_{s_t \sim \rho^{\beta}} \left[\nabla_{\theta^{\mu}} Q(s, a|\theta^{Q})|_{s=s_t, a=\mu(s_t|\theta^{\mu})} \right] \tag{16}$$

Since these actions are continuous numbers and not individual classes, the gradient directly helps in tuning the action values. This makes the policy deterministic as opposed to stochastic from previous approaches [19].

F. Soft Actor-Critic

The problem with TRPO and PPO is that their sampling efficiency is low. Meaning, every time you want to train the model, you need to play episodes in parallel and create a batch of training data. We could store the batches in memory but it is usually not done because when used with RNNS, random sampling loses the data related to hidden or cell states ie it requires sequential data. DDPG ditches this idea and uses random batch sampling with replay buffer. Although this is efficient, it tends to be unstable compared to TRPO and PPO. Haarnoja et al introduced an approach called Soft Actor Critic (SAC) that combines the best of both worlds [20]. SAC uses stochastic policies and replay buffer and performs more stable than DDPG and has better sampling efficiency than TRPO and PPO. The key idea of SAC is that it gives higher probability to exploration. If we consider the approaches above, as the training proceeds, the probability distribution over the set of actions tends to converge to a single action. This makes the policy deterministic and other actions get lower chances of getting picked. Its loss function is defined in such a way that it tries to maximize the rewards as well as entropy. The policy is defined as

$$\pi^* = argmax_{\pi} \sum_{t} \mathbb{E}_{(s_t, a_t) \sum \rho_{\pi}} \left[r(s_t, a_t) + \alpha H(\pi(.|s_t)) \right] \tag{17}$$

The last term in the equation above is the entropy term which is defined as

$$H(\pi(.|s_t)) = \mathbb{E}_{a \sim \pi(.|s)} \left[-log(\pi(a|s)) \right] \tag{18}$$

Roughly, it combines the expected rewards and negative log probability of all actions and picks the action with maximum value.

VI. Other Approaches

A. Hierarchical Reinforcement Learning

The idea behind hierarchical reinforcement learning is that actions don't need to be fine-grained. Simple actions can be abstracted into more complex actions. For example, when we walk, we don't think about moving individual muscle fibres. We simply consider walking as a single action More fine-grained details can be abstracted. Vezhnevets et al, 2016 [21], 2017 [22] and Nachum et al, 2018 [23] demonstrated the idea of Hierarchical Reinforcement Learning with Deep Learning. Using their model, an agent can plan and take macro actions. For example, in a grid world like environment, instead of taking actions like up, up, up, up, right; the agent could take macro actions like "go all the way up an then take a right". These macro actions are also called options. An option is the combination of an option policy, a termination condition, and an input set. The model would first select an option policy based on the state. Then all further actions are decided by the policy network of the selected option until the termination condition is met. After that, an option policy is selected again and this continues. Here the neural network within an option policy can be a smaller one because it only deals with particular type of states.

B. Exploration-based - Curiosity Driven

The Atari model did play beyond human level for few games, but was much worse on a few games. For example, in the game of Montezuma's Revenge, it couldn't even pass the first level. [24] The reason for this poor performance is that the way rewards were given in simple games like Breakout. For every few random actions taken by the model, it was likely that some actions were given positive reward. This reinforces the good actions. And the reward system is frequent enough to keep getting positive rewards ever so often to learn from them. In other words, the reward system is Dense. In games like Montezuma's Revenge, the agent would need to move around and avoid obstacles and obtain a key. It requires long-term planning and the rewards doesn't appear until the key is found. This is called the Sparse reward problem.

The most interesting paper in this area is [25]. The researchers developed a complex learning framework. The rewards can be extrinsic (explicitly given by the environment) or intrinsic (gathered from exploration). The randomness in state transition is also rectified by using scheme where only the parts of the state that get influenced by the action are taken into consideration. The algorithm is capable of learning only from the extrinsic rewards or only from the intrinsic rewards (self-supervised / exploration / curiosity based) and it is also capable of remembering the seen states and exploits using the learned experience.

Benjamin Eysenbach et al with their paper "Diversity Is All You Need" [26] takes an extreme approach where the agent completely learns only from the intrinsic rewards. Previously, Justin Fu et al [27] used a memory-like network where the agent learns to distinguish visited states vs unseen states. One of the earliest paper that uses exploration [28] used bayesian neural networks for the exploration.

C. Model based

We usually refer to our Neural Network as the model. But in context of model-based reinforcement learning, the model is an approximation of the environment. Nicely enough, this model is actually approximated by a neural network. In the Deep Q Learning, Policy Gradient or other model-free methods, all we cared about is how good the model predicts the optimal actions. In model-based method however, we also care about how good the model predicts the behavior of the actual model [29]. The main goal of model-based approaches is to let the agent make necessary long-term planning into the future

before deciding on what action to take. It can look several steps into the future and see the consequences of taking an immediate action. A simple approach to model-based planning is to have a different neural network that can predict the future state. And we can feed this future state s_{t+1} into the same network and get a future state s_{t+2}. This can get increasingly worse as we look further into the future. However, predicting few states into the future can also be helpful in planning the right action.

D. Monte Carlo Tree Search

In games like Chess or Go, there exists an opponent who generally plays just like the agent. In environments like these where there is an opponent agent with whom we want to compete and win, approaches like tree searching are common. In a traditional monte carlo tree search, the algorithm would consider all possible actions, then for each state transition, it computes the all the state transitions and it goes like a tree until the algorithm reaches the first winning state depth-wise [30]. The action that lead towards this winning path is then taken. This can also be complemented with a MiniMax approach where the model computes all the actions of the opponent and makes similarly figures out what the best action for the opponent is. Then we use a method called alpha-beta pruning and pick the best action. IBM had created a world-class chess playing AI that did not use any deep learning [31]. It was solely based on some smart Monte Carlo Tree Search based algorithms. However, as the number of states increase, it gets exponentially difficult to trace all the actions and generate the tree. This is where deep learning comes into play. The AlphaGo algorithm developed by DeepMind had beaten the world-class Go player with a 4:1 win [32]. In this paper, instead of considering all the state transitions, the deep learning model chooses the best actions based on a state action value prediction. Then these actions are considered for the next layer of nodes in the tree. The state-action values are again predicted for each action and this continues until the algorithm finds a path where it wins. These kind of algorithms that combine deep learning and traditional tree search methods are state-of-the-art for the type of games like Chess and Go.

VII. Conclusion

From my brief exploration of Model-Free Deep Reinforcement Learning, it appears that the domain of Reinforcement Learning after involvement of Deep Learning has not only become a vast field, but also a deeper field with many rabbit holes. An interesting find is that most of the advancements in Deep Reinforcement Learning happened just a few years ago. This indicates that the field is still in its infancy and has a lot of potential to improve, especially the Model-Based branch of DRL. In its essence Deep Reinforcement Learning could be one of the first baby steps towards Artificial General Intelligence which is the holy-grail of all of Artificial Intelligence domain.

To summarize, we have seen that Deep Learning plays a huge part in Reinforcement Learning domain. They can predict the state or state-action value functions with Deep Q Learning framework. They can also predict the policy directly using Policy Gradient algorithms like REINFORCE. We have seen that we can combine the Q Learning and Policy Gradient into Actor-Critic networks and develop various kinds of algorithms like TRPO, PPO, DDPG, SAC, etc. Finally, we have seen that there are other more advanced and fascinating ideas being explored like Hierarchical RL, Curiosity and Exploration based RL and Model-Based RL that have a tremendous potential in this domain.

References

[1] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, 1961.

[2] R. Evans and J. Gao, "Deepmind ai reduces google data centre cooling bill by 40% — deepmind." https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40, 2016. (Accessed on 11/29/2020).

[3] M. Popova, O. Isayev, and A. Tropsha, "Deep reinforcement learning for de-novo drug design," *CoRR*, vol. abs/1711.10907, 2017.

[4] S. Amarjyoti, "Deep reinforcement learning for robotic manipulation - the state of the art," *CoRR*, vol. abs/1701.08878, 2017.

[5] F. Rocha, V. Costa, and L. Reis, *From Reinforcement Learning Towards Artificial General Intelligence*, pp. 401–413. 06 2020.

[6] Bellman and Richard, *Dynamic Programming*. Princeton, NJ, USA: Princeton University Press, 1 ed., 1957.

[7] C. Szepesvári, *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2010.

[8] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[10] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning Representations by Back-propagating Errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[11] OpenAI, "Part 2: Kinds of rl algorithms — spinning up documentation." https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html, 2018. (Accessed on 11/29/2020).

[12] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992.

[13] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Advances in Neural Information Processing Systems 12*, vol. 12, pp. 1057–1063, MIT Press, 2000.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning.," in *ICLR* (Y. Bengio and Y. LeCun, eds.), 2016.

[15] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, pp. 1928–1937, 2016.

[16] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2015. cite arxiv:1511.05952Comment: Published at ICLR 2016.

[17] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization.," in *ICML* (F. R. Bach and D. M. Blei, eds.), vol. 37 of *JMLR Workshop and Conference Proceedings*, pp. 1889–1897, JMLR.org, 2015.

[18] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[19] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. A. Riedmiller, "Deterministic policy gradient algorithms.," in *ICML*, vol. 32 of *JMLR Workshop and Conference Proceedings*, pp. 387–395, JMLR.org, 2014.

[20] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *CoRR*, vol. abs/1801.01290, 2018.

[21] A. Vezhnevets, V. Mnih, J. P. Agapiou, S. Osindero, A. Graves, O. Vinyals, and K. Kavukcuoglu, "Strategic attentive writer for learning macro-actions," *CoRR*, vol. abs/1606.04695, 2016.

[22] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," *CoRR*, vol. abs/1703.01161, 2017.

[23] O. Nachum, S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," *CoRR*, vol. abs/1805.08296, 2018.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb 2015.

[25] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," *CoRR*, vol. abs/1705.05363, 2017.

[26] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *CoRR*, vol. abs/1802.06070, 2018.

[27] J. Fu, J. D. Co-Reyes, and S. Levine, "EX2: exploration with exemplar models for deep reinforcement learning," *CoRR*, vol. abs/1703.01260, 2017.

[28] R. Houthooft, X. Chen, Y. Duan, J. Schulman, F. D. Turck, and P. Abbeel, "Curiosity-driven exploration in deep reinforcement learning via bayesian neural networks," *CoRR*, vol. abs/1605.09674, 2016.

[29] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. W. Battaglia, D. Silver, and D. Wierstra, "Imagination-augmented agents for deep reinforcement learning," *CoRR*, vol. abs/1707.06203, 2017.

[30] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Monte-carlo tree search: A new framework for game ai," in *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'08, p. 216–217, AAAI Press, 2008.

[31] F.-h. Hsu, "Ibm's deep blue chess grandmaster chips," *IEEE Micro*, vol. 19, p. 70–81, Mar. 1999.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.