State-of-the-Art Reinforcement Learning Algorithms

Deepanshu Mehta
B. Eng (Information Technology)
Panjab University (UIET)
Chandigarh, India

***Abstract*—This research paper brings together many different aspects of the current research on several fields associated to Reinforcement Learning which has been growing rapidly, providing a wide variety of learning algorithms like Markov Decision Processes (MDPs), Temporal Difference (TD) Learning, Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C), Deep Q Networks (DQNs), Deep Deterministic Policy Gradient (DDPG) and Evolution Strategies (ES) for different applications. In this paper, the computations and procedures involved in Reinforcement Learning algorithms are briefly discussed. Reinforcement Learning can be used is almost every field for its automation and advancement. Nowadays, Meta-Learning, Automated Machine Learning and Self-Learning Systems have become very popular. Meta-learning which is an application of evolution strategies is an exciting area of research that tackles the problem of learning to learn faster with being generalizable to many tasks. Automated machine learning is the process of automating end-to-end the process of applying machine learning to real-world problems.**

***Keywords*— *Markov's Decision Processes, Q Learning, Temporal Difference Learning, Actor-Critic Algorithms, Deep Deterministic Policy Gradients, Evolution Strategies Algorithm*.**

I. INTRODUCTION

Reinforcement Learning (RL) is an area of Machine Learning which is very dynamic in terms of theory and its application. Reinforcement Learning algorithms study the behavior of subjects in environments and learn to optimize their behavior[1]. RL algorithms can be classified as shown in Fig.1.

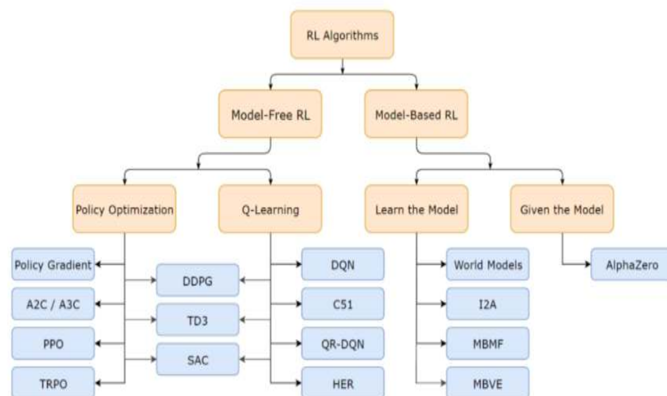

Fig. 1. Reinforcement Learning classification.

RL algorithms can be categorized mainly into Value-based or Value Optimization(Q-Learning) RL, Policy-based or Policy Optimization RL and Evolution Strategies which is a completely different Reinforcement Learning approach. [1] There are a little Terminologies like Agent: The learner and decision-maker, Environment: where the agent learns and decides what actions to perform, Action(A): set of actions 'a' which can be performed in an environment, State(S): the set of states of an agent in the environment, Reward(R): each action performed by the agent provides a positive or a negative reward. Expected Return(G): It is the cumulative sum of rewards which the agent tries to maximize as shown in "(1)". [2]

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} + \text{- - - - - - -} + R_T \quad (1)$$

where 'T' is the final time step.

Discounted Return: In this return, discount rate 'γ' ε [0,1] is used to discount the future rewards and determine the present value of future rewards so that more immediate rewards are given more importance. Hence, expression of Discounted Return becomes as shown in "(2)".

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \text{- - - - -} = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (2)$$

Policy(π): the decision-making function that maps a given state to probabilities of each possible action from that state. Value function: These are functions of states that evaluates how adequate it is for an agent to be in given state (State-value function) which is denoted by "V_π" or these are functions of state-action pairs that estimate how good it is for an agent to perform a given action in a given state (Action-value function) which is denoted by "Q_π". Both of these functions are given in terms of Expected Return "E_π" as shown in "(3)" and "(4)". [2][3]

$$V_\pi(s) = E_\pi [G_t \mid S_t = s] = E_\pi [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \quad (3)$$

$$Q_\pi(s,a) = E_\pi [G_t \mid S_t = s, A_t = a] = E_\pi [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a] \quad (4)$$

II. COMPUTATIONS AND PROCESSES INVOLVED IN RL ALGORITHMS

A. *Markov's Decision Processes*

It is the framework we use to describe RL problems. In MDP, Agent and Environment interact continually and learns simultaneously as shown in Fig.2. The environment

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

transitions into a new state when an agent takes an action and during that very moment the agent gets a reward based on its action. Transition is represented by a tuple (s, a, r, s'), where s is the previous state, a is the action taken, r is the reward received on taking the action a and s' is the next state which the environment is transitioned into[4]. The transition probability from s state to s' state with reward r and action a is shown in "(5)".

$$P(s', r \mid s, a) = P_r\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \quad (5)$$

While interacting with the environment, the main goal of the agent is to maximize the returns according to which optimal policy, optimal state-value function V_*, optimal action-value function Q_* are chosen[4]. Bellman Optimality equation for calculating Q_* proves to be immensely useful.[4]

$$Q_*(s,a) = E[R_{t+1} + \gamma max_{a'} Q_*(s', a')] \quad (6)$$

OR

$$V_\pi(s) = E[R_{t+1} + \gamma(V_\pi(S_{t+1})) \mid S_t = s] \quad (7)$$

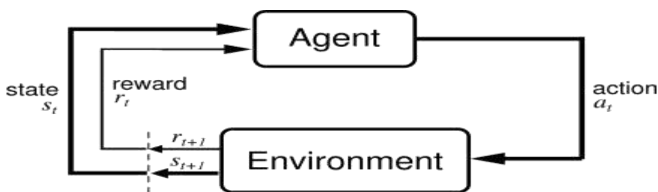

Fig. 2. Agent-Environment Interaction

Once we have Q_*, we can determine the optimal policy as with Q_* for any state s, an RL algorithm can find the action a that maximizes $Q_*(s,a)$.

Also, in MDP, Epsilon greedy Strategy or ε – greedy Strategy is used to get a balance between exploitation and exploration. Exploration is the act of exploring the environment to find out information about it whereas Exploitation is the act of exploiting the information that is already known about the environment in order to maximize the return. The agent will always start with the exploration as it does not know anything at that time. Here "ε" is the exploration rate which ranges from 0-1 where ε=1 means the agent is only exploring and ε=0 means it is only exploiting the information it has. MDPs are used in TD Learning, DQNs,A2C,A3C and DDPG.[5][6]

B. *Temporal Difference Learning*

Temporal difference is an agent learning from an environment through episodes without any preliminary information about the environment. TD Learning is considered as a better algorithm as compared to Monte-Carlo (MC). In TD Learning, the agent learns at every step and update values unlike in MC, where the values are updated at the termination of an episode. TD Learning is an unsupervised learning approach. [7][8]

TD(1) is the same as MC and TD Error can be calculated as shown in "(8)".

$$TD\ Error = G_t - V(S_t) \quad (8)$$

Updation of values: $V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t)) \quad (9)$

In TD(0), TD Error is calculated using "(10)".

$$TD\ Error = R_{t+1} + \gamma\, V(S_{t+1}) - V(S_t) \quad (10)$$

Updation of values: $V(S_t) \leftarrow V(S_t) + \alpha\,(R_{t+1} + \gamma\, V(S_{t+1}) - V(S_t)) \quad (11)$

In TD(0), instead of using G_t, we only look at immediate Reward R_{t+1} plus the discount of the estimated value of only 1 step ahead $V(S_{t+1})$. TD(λ) is used if we want to update values prior to the ending of the episode and use more than one step ahead for our calculation. It has two views in it: Forward and Backward view.

Forward View: Looks at the next n-steps frontwards and λ is essentially operated to decay those future estimates. λ is the credit assignment variable.

Backward View: Updates values at each step. So, after each step in an episode, you make updates to all prior steps[9]. δ_t is the TD Error as shown in "(12)". We also use Eligibility Traces (ET) to assign credit to prior steps appropriately. Basically, ET keeps a record of the occurrence and recency of moving into a given state which can be calculated using "(13)". Credit is assigned to the states that are visited frequently and recently with respect to our final state.[9] The lambda (λ) and gamma (γ) are the terms which discount those traces.

$$\delta_t = R_{t+1} + \gamma\, V(S_{t+1}) - V(S_t) \quad (12)$$

$$ET_t(s) = (\gamma\lambda)\, ET_{t-1}(s) + 1 \quad (13)$$

Updation of values: $V(S_t) \leftarrow V(S_t) + \alpha\, \delta_t\, ET_t(s) \quad (14)$

An environment can have an infinite number of states (i.e. continuous state spaces). If we are using a neural network, then to update its weights θ, we well do,

$$\theta \leftarrow \theta + \alpha(r + \gamma max_{a'} Q_*(s', a') - Q(s, a)) \frac{\partial Q(s,a)}{\partial \theta} \quad (15)$$

where $Q_*(s, a) = E[\, r + \gamma max_{a'} Q_*(s', a') \mid s]$

TD Error is used in A2C, A3C, and DDPG[10].

III. STATE OF THE ART REVIEW OF REINFORCEMENT LEARNING ALGORITHMS

A. *Deep Q Learning*

In this algorithm, we use DQNs or Deep Q Networks which consists of deep neural networks. It is a value-based RL algorithm. Each state in the environment would be expressed by a set of pixels and the agent would be capable to take distinct actions from each state. Rather than using value iterations as in MDPs to determine the Q-values and find optimal Q-function, we alternatively use a function approximator to estimate optimal Q-function i.e. using Deep Neural Networks. In Q Learning, the target depends upon the prediction.[11] Q Learning is a semi-gradient off-policy algorithm. We will make use of DQNs as shown in Fig.3 to estimate the Q-values for each state-action pair in a given environment. The objective of this network is to approximate the optimal Q-function which will satisfy the Bellman equation. The loss from the network is determined by comparing the outputted Q-values to the target Q-values from the righthand side of the Bellman equation. After the loss is calculated, the network updates weights via Stochastic Gradient Descent and Backpropagation and this is how loss is minimized[12].

With Deep Q Networks, we often utilize the technique called "Experience Replay" and "Replay Memory" during its training. In it, we store all the agent's experiences e_t at each

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

time step in a dataset called replay memory in a form of tuple represented as $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. The replay memory dataset is randomly sampled and used to train the network and this sampling helps in breaking the correlation between the consecutive steps and avoids inefficient learning. Now, we use two kinds of Networks namely Policy Networks and Target Networks.

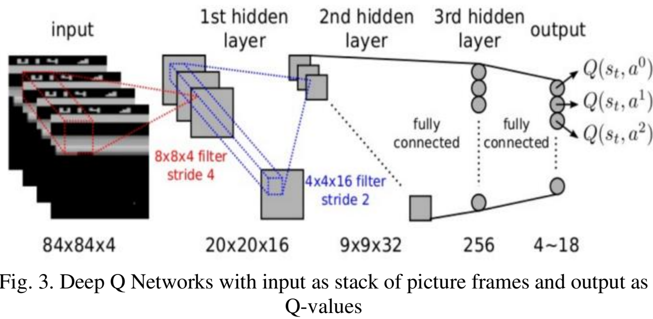

Fig. 3. Deep Q Networks with input as stack of picture frames and output as Q-values

The network which is fed by random sampled e_t for training and outputs Q-values is called the policy network. In this network, the loss which is shown in "(16)" is backpropagated and minimized[13]. A Q table is made which is updated at each time step during training. New Q-value is equal to the weighted sum of old Q-value and the learned value as shown in "(17)".

$$Loss = E\,[R_{t+1} + \gamma max_{a'}\, Q_*(s', a')] - E\,[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}] \quad (16)$$

$$Q^{new}(s,a) = (1-\alpha)\, Q(s,a) + \alpha\, (R_{t+1} + \gamma max_{a'}\, Q_*(s', a')) \quad (17)$$

where α is the learning rate.

$$Target = r + \gamma(1 - done)\, max\, \{Q_*(s', :)\} \quad (18)$$

Prediction = Q(s,a), where done = True if the task is done successfully and done = False in the opposite case.

$$Error = (Target - Prediction)^2 \quad (19)$$

If we use the same Q in Target and Prediction, then Target is always fluctuating along with the prediction so, both will become dependent on each other and thus inefficiency hence, we use a separate Target Network for getting Target values to avoid this. [14]

B. A2C and A3C Algorithms

A2C Stands for Advantage Actor-Critic and A3C Stands for Asynchronous Advantage Actor-Critic Algorithm. Both algorithms are policy-based RL algorithms. Policy-based algorithms output policies rather than the q values and each policy distribution has different exploration estimations. Policy-based methods can handle continuous action spaces easily as it represents parameters of the distribution as output which is finite[12][15]. In training a policy-based algorithm, instead of minimizing error and finding optimal policy, the concept of gradient is used. According to Policy Gradient Theorem,

$$\nabla_\theta J(\theta) = E\,[\,A(s,a)\, \nabla_\theta \log \pi(a|s)] \approx (1/N)\,[\sum_{i=1}^{N} A(s_i,a_i)\, \nabla_\theta \log \pi(a_i|s_i)], \quad (20)$$

where *Advantage*, $A(s,a) = Q(s,a) - V(s)$, (21)

or $A(s,a) = R + \gamma V(s') - V(s)$, (22)

and ∇_θ is the gradient and V(s) is the baseline. J(θ) is the loss function whose gradient with respect to θ is found. Advantage function captures how preferable an action can be as compared to others at a given state, while we know the value function captures how beneficial it is to be at this state. Both A2C and A3C are actor-critic algorithms. In A2C and A3C, take N = 5, collect all (state, action) pairs, calculate the N-Step Reward and Advantage, and after that go in the direction of the gradient and minimize the loss to update weights in the neural network.

In A3C, we have one master network which intermittently copies its weights to the worker networks as shown in Fig.4. The worker nets are responsible for doing the rollouts. This process is Multi-threaded. Every 5 steps, each worker sends its gradients back to the master. Instead of updating its own weight, the worker sends its gradients back to the master net and master net updates its own weights. So, the master has the most up to date policy. A3C implements Parallel training where multiple workers in parallel environments independently update a global value function. These agents one by one interacts with its own copy of the environment and at the same time, the other agents are interacting with their environments. The reason this works better than having a single agent (beyond the speedup of getting more work done), is that the experience of each agent is not reliant on the experience of the others. In this way, the overall experience available for training becomes more divergent.

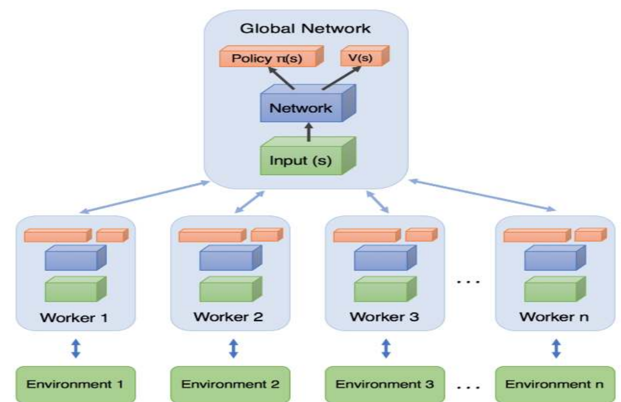

Fig. 4. A3C architecture

In A2C, the steps are performed in each worker synchronously unlike A3C. In A2C, a single-worker variant of A3C is present. A2C is like A3C but without asynchronous part. The critic estimates the value function and actor updates the policy distribution in direction suggested by the critic with policy gradients.[16] In A2C, we simultaneously optimize the value function and the policy. Take N=5 steps of an episode, collect (state, action) pairs, calculate N-step reward and advantage, and go in the direction of the gradient. The regularization here can be thought of as exploration. Equation (23) and (24) determines cost function and loss function respectively.

$$J = \sum_{i=1}^{N} (\, y_i - w^T x_i\,)^2 + \lambda\, |\theta|^2, \quad (23)$$

here λ is called the regularization parameter and is used to penalize the weights. Regularized loss = Policy loss + Penalty.

$$L = -\,E\,[\,A(s,a) \log \pi(s|a)\,] - H(\pi), \quad (24)$$

where $H(\pi) = \sum_{i=1}^{N} \pi_i \log \pi_i$ and H(π) is called the entropy. Entropy is directly proportional to the exploration.[17]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

C. Deep Deterministic Policy Gradient Algorithm

When we take DQNs (which works with only discrete actions) and modify it to work with continuous actions or action spaces, the result comes out to be DDPG as shown in Fig.5. DDPG is the combination of DQN and PG (Policy Gradient). We use semi-gradient descent and all tricks of DQNs are applicable in DDPG. It is only used for environments which can have continuous action spaces i.e. is not used much in video games but is used in our movements or robot's movements which will require continuous control. [18]

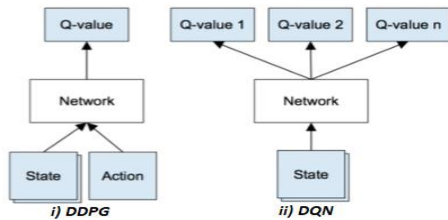

Fig. 5. Inputs and Outputs in DDPG and DQN respectively

DDPG approach is a little different from the DQN approach. The limitation of handling continuous action spaces in DQN is overcome in DDPG. DDPG has two different networks just like GANs. One of those is a Deterministic policy function μ(s) represented by a neural network that outputs the optimal action (scalar or vector) after taking an input 's'. Deterministic policy function, $\mu(s) = \arg\max_a Q(s,a)$. The other network is Q network which gets optimal action from μ(s) and state 's' as its inputs. Policy network μ(s) must pass through the Q network to get the loss and output as an Action-Value from the latter. When updating weights of μ-net 'θ_μ', weights of Q-net 'θ_Q' remain fixed and the output from Q net is maximized by adjusting the weights in μ-net[19]. For optimizing μ(s), the loss function for μ-net and the Gradient of its loss function is shown in "(25)" and "(26)" respectively.

$$J_\mu = E\,[\,Q(\,s, \mu(s)\,)\,] \tag{25}$$

$$\nabla_{\theta^\mu} J_\mu = E\,[\,\nabla_\mu Q(\,s, \mu(s)\,)\,.\,\nabla_{\theta^\mu}\,\mu(s)] \tag{26}$$

We use the suboptimal approach and calculate the gradient of the loss function and try to maximize the sum of future rewards. DDPG updates μ and Q nets alternatively considering two separate losses for each[20],[21]. Loss function for Q-net can also be calculated as shown in "(27)" and we will try to minimize it,

$$J_Q = (1/N) \sum_{i=1}^{N} (\,r_i + \gamma(1 - d)\, Q_{targ}\,(s_i', \mu_{targ}(s_i')) - Q(s_i, \mu(s_i)))^2 \tag{27}$$

In DDPG, we do a soft update for both policy network and Q-network unlike DQN i.e. we copy just a fraction of weights from the main policy network and Q-network to two separate target networks on every step.

$$\theta^\mu{}_{targ} \leftarrow \tau\, \theta^\mu{}_{targ} + (1 - \tau)\, \theta^\mu \tag{28}$$

$$\theta^Q{}_{targ} \leftarrow \tau\, \theta^Q{}_{targ} + (1 - \tau)\, \theta^Q \tag{29}$$

where $0 < \tau << 1$ [22]

D. Evolution Strategies Algorithm

Evolution Strategies (ES) is a black-box optimization method. Both Value-based and Policy-based categories use gradient descent to minimize the loss but ES takes a biologically inspired approach in particular of evolution. Evolution includes the concept of "Natural Selection". As a general nature's rule, the fit or the strong survive and the weak die. The offsprings which survive will produce offsprings for the next generation and that generation will be slightly different from their parents and these beneficial changes will compound and after many generations, the offsprings will be much stronger than their ancestors. Good changes are kept and bad changes are thrown away as those die.[23] Hill Climbing is an optimization technique used to find the local optimum solution to the computational problem. It starts with a solution that is very poor compared to the optimal solution and then iteratively improves from there. It does this by generating "neighbor" solutions which are relatively a step better than the current solution, picks the best and then repeats the process until it arrives at the most optimal solution because it can no longer find any improvements. Adding random noise leads to climbing of the hill and if the fitness of the model is less, then the noise is deleted. We try a new point and if that point is better than our current point, then we make it our current point and if not, then we consider another random point.[24] Also, Gradient descent is a specific kind of "hill climbing" algorithm. Let the learning rate = η; Normal distribution = ε_n; Noise Standard deviation = σ and Initial policy parameters = θ(0), where θ(n)= policy parameters for n^{th} policy.

$$\theta_{try} = \theta(t) + \sigma\, \varepsilon_n \tag{30}$$

θ_{try} is used to calculate reward by checking the fitness or it can also be the accuracy. For calculating reward, $F_n = F(\theta_{try})$, where F_n is called the reward function. [25]

Updation of policy parameters is shown in "(31)".

$$\theta(t+1) = \theta(t) + \eta\, \frac{1}{N\sigma} \sum_{n=0}^{N} F_n\, \varepsilon_n \tag{31}$$

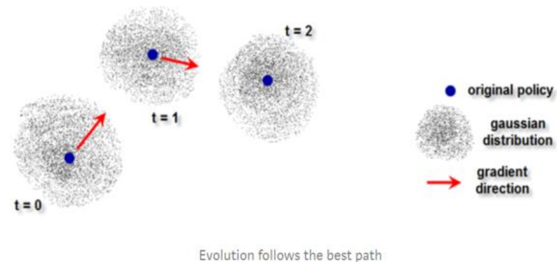

Fig. 6. Applying Gradient Descent to ES for training.

We want to go in directions which are better than where we currently are. The concept of parallelization is used in running multiple offsprings. No backpropagation, MDPs, Bellman eq., value function, etc like previously are used.[26][27] We see better exploration behavior in ES as compared to other policy techniques. There are fewer hyperparameters like learning rate, population size (number of offsprings to create) and noise deviation (how far can offsprings go from the parent). Fig.6 is showing that given an initial policy, we can always generate a population of similar policies around it by applying random changes to its weights. We then evaluate all these new policies and estimate the gradient i.e. we check in what direction things look more promising. Finally, we update weights and policy parameters to move exactly in that direction and start again and loop until we are satisfied with the outcome. [26]

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 8 Issue 12, December-2019

IV. DIFFERENCES BETWEEN THE RL ALGORITHMS

The most valid differences which can be stated between the RL algorithms discussed above are as follows (see Table I).

Table- I: The most valid differences which can be formed in

	DQN	Advantage Actor-Critic	DDPG	ES
1.Classification	It is a value-based RL algorithm[2]	A2C and A3C are policy-based RL algorithms	It is a combination of both value and policy-based RL algorithms[2]	It is different from value and policy-based RL algorithms [24]
2.Training Speed	Slowest	Fast	Slower than actor-critic but faster than DQNs [18]	Fastest
3.Action Spaces	Discrete[11]	Discrete and Continuous both	Only continuous[22]	Discrete and Continuous both[27][25]
4.Memory Consumed	Large replay memory required although it has only one target net[13][14]	Larger memory required as it has many worker nets	Larger replay memory required as it has two separate μ and Q target net [20]	Very Large memory required to store data for training
5.Parallelization Required	No	Yes as many worker nets are working in parallel to update the master net[16]	Does not support parallelization[21]	Yes as many offspring models are running in parallel[26]
6.Backpropagation	Happens[11]	Happens	Happens[21], [22]	Does not happen
7.Sub-networks information	Contains only basic deep neural nets	Actor is π (stochastic) and critic is V (state-value function). Actors are the workers which run in parallel with one critic i.e. master net.[17]	Actor is μ (deterministic) and actor is Q (action-value function). There is just one actor and one critic	Multiple offsprings running in parallel following the concept of Natural Selection by following best gradient direction
8.Weights used	Weights on the main network copied to target nets[12]	Weights of Master net are copied to worker nets	Weight are updated using Soft updates to μ and Q target nets separately[20]	Offspring nets have same weights initially

V. CONCLUSION

This paper has provided an overview of reinforcement learning algorithms which were used in the past and also the algorithms which are in use these days. The comparison between the RL algorithms shows that the Evolution strategies algorithm is much more efficient and faster than other RL algorithms with the only drawback that the data used for its training acquires a lot of memory. Reinforcement Learning is not just limited to the algorithms discussed in this paper. Most recent applications in this particular technology are Neural Scene Representation and Rendering, Brain-Computer Interface, Stock Predictions, Trading, Sports betting, proving complex Mathematical Theorems, Health Care, Astronomy, Business, Manufacturing, Chatbots, Self-driving Car, Astronomy, Playing video games at Superhuman levels and many more. Reinforcement Learning is the future as it has been predicted by researchers and scientists that humanoid robots will be built in the future with superhuman powers that will be much more intelligent and efficient than an average human. It will also be able to do innovations, learn by itself and perform several tasks that a human cannot do at all.

REFERENCES

[1] N. R. Ravishankar and M. V. Vijayakumar, “Reinforcement Learning Algorithms: Survey and Classification,” Indian J. Sci. Technol., vol. 10, no. 1, pp. 1–8, 2017.
[2] A. Gosavi, “Reinforcement learning: A tutorial survey and recent advances,” INFORMS J. Comput., vol. 21, no. 2, pp. 178–192, 2009.
[3] A. K. Chattopadhyay and T. Chattopadhyay, “Monte Carlo simulation,” Springer Ser. Astrostatistics, vol. 3, no. March, pp. 241–275, 2014.
[4] J. Patrick and M. A. Begen, “Markov decision processes and its applications in healthcare,” Handb. Healthc. Deliv. Syst., no. January 2011, p. 17, 2016.
[5] M. Van Otterlo, “Markov Decision Processes: Concepts and Algorithms,” Course ’Learning Reason., no. May, pp. 1–23, 2009. the algorithms discussed.
[6] S. Thrun, “Monte Carlo POMDPs,” Adv. Neural Inf. Process. Syst. 12, pp. 1064–1070, 2000.
[7] H. Penedones, D. Vincent, H. Maennel, S. Gelly, T. Mann, and A. Barreto, “Temporal Difference Learning with Neural Networks - Study of the Leakage Propagation Problem,” no. July, 2018.
[8] A. Amiranashvili, A. Dosovitskiy, V. Koltun, and T. Brox, “TD or not TD: Analyzing the Role of Temporal Differencing in Deep Reinforcement Learning,” no. 2016, pp. 1–15, 2018.
[9] G. Tesauro, “Practical Issues in Temporal Difference Learning,” Mach. Learn., vol. 8, no. 3, pp. 257–277, 1992.
[10] F. Kunz, “An Introduction to Temporal Difference Learning,” Citesee, pp. 1–8, 2000.
[11] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518, no. 7540, pp. 529–533, 2015.
[12] A. Stooke and P. Abbeel, “Accelerated Methods for Deep Reinforcement Learning,” 2018.
[13] V. François-lavet et al., “An Introduction to Deep Reinforcement Learning. (arXiv:1811.12560v1 [cs.LG]) http://arxiv.org/abs/1811.12560,” Found. trends Mach. Learn., vol. II, no. 3–4, pp. 1–140, 2018.
[14] V. Mnih, D. Silver, and M. Riedmiller, “Deep Q Network (Google),” pp. 1–9, 2015.
[15] P.-H. Su, P. Budzianowski, S. Ultes, M. Gasic, and S. Young, “Sample-efficient Actor-Critic Reinforcement Learning with Supervised Data for Dialogue Management,” pp. 147–157, 2018.
[16] M. A. F. Birck, U. B. Correa, P. Ballester, V. O. Andersson, and R. M. Araujo, “Multi-Task Reinforcement Learning : An Hybrid A3C Domain Approach,” Eniac, no. February 2018, 2017.
[17] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu, “A2C: Self Destructing Exploit Executions via Input Perturbation,” no. March, 2017.
[18] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic policy gradient algorithms,” 31st Int. Conf. Mach. Learn. ICML 2014, vol. 1, pp. 605–619, 2014.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)

[19] S. Choi, T. Le, Q. Nguyen, M. Layek, S. Lee, and T. Chung, “Toward Self-Driving Bicycles Using State-of-the-Art Deep Reinforcement Learning Algorithms,” Symmetry (Basel)., vol. 11, no. 2, p. 290, 2019.

[20] Y. Hou and Y. Zhang, “Improving DDPG via Prioritized Experience Replay,” no. May, 2019.

[21] X. Wu, S. Liu, T. Zhang, L. Yang, Y. Li, and T. Wang, “Motion Control for Biped Robot via DDPG-based Deep Reinforcement Learning,” 2018 WRC Symp. Adv. Robot. Autom. WRC SARA 2018 - Proceeding, pp. 40–45, 2018.

[22] T. P. Lillicrap et al., “Continuous control with deep reinforcement learning,” 2015.

[23] T. Bäck, F. Hoffmeister, and H.-P. Schwefel, “A survey of evolution strategies,” Proc. Fourth Int. Conf. Genet. Algorithms, vol. 9, no. 3, p. 8, 1991.

[24] M. Emmerich, O. M. Shir, and H. Wang, “Evolution strategies,” Handb. Heuristics, vol. 1–2, pp. 89–119, 2018.

[25] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, “Evolution Strategies as a Scalable Alternative to Reinforcement Learning,” pp. 1–13, 2017.

[26] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, “Natural evolution strategies,” J. Mach. Learn. Res., vol. 15, pp. 949–980, 2014.

[27] E. Conti, V. Madhavan, F. P. Such, J. Lehman, K. O. Stanley, and J. Clune, “Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents,” Adv. Neural Inf. Process. Syst., vol. 2018-Decem, no. NeurIPS, pp. 5027–5038, 2018.

(This work is licensed under a Creative Commons Attribution 4.0 International License.)