Deep Reinforcement Learning Algorithms to Play Atari Games and its Applications

Rohit R.R
M.E S.S.A
SR No: 12321
Indian Institute of Science, Bangalore
Advisor : Prof.Shalabh Bhatnagar

***Abstract*—In this project we explore the various state of the art algorithms which use deep neural networks with Reinforcement Learning. Using deep neural networks in reinforcement learning algorithms enables us to successfully learn complex tasks directly from high dimensional sensory inputs. We implement and analyze how they work, feasibility of their usage and how effective they are when compared to existing reinforcement learning algorithms. We try to improve upon the existing algorithms with experience replay, batch and weight normalization. The concept of weight normalization is shown to improve the performance of the algorithm. We try to apply these new algorithms on some real life applications.**

I. Introduction

Reinforcement Learning is a field of machine learning, concerned with how agents ought to take actions in an environment so as to maximize some notion of cumulative reward. But in order to use reinforcement learning in most real-life cases with high complexity we mostly need to deal with high dimensional state spaces and we must derive efficient representations from the high dimensional sensory inputs that we get. So while reinforcement learning has been successful in many cases, their applicability has been restricted to cases where useful features can be handcrafted or in cases with fully observed, low dimensional state spaces. But with the advent of Deep Neural Networks and with their power of learning these constraints are being overcome. Combining reinforcement learning algorithms with Deep Learning concepts give rise to "Deep Reinforcement Algorithms".

We try to make some improvements to some existing algorithms and will look into applications where these can be applied. We start with explaining some of the reinforcement learning concepts used for the algorithms. Then we explain the main deep reinforcement algorithms that have been introduced recently mainly for discrete action spaces. They are accompanied with the implementation details and results for the different papers followed by the details about our work.

II. Reinforcement Learning Background

We consider the standard reinforcement learning setting where an agent interacts with an environment E over number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action at from some set of possible actions A according to its policy π, where π is a mapping from states s_t to actions a_t . In return, the environment moves to the next state s_{t+1} and gives the agent a scalar reward r_t . The process continues until the agent reaches a terminal state after which the process restarts. The return $R_t = \sum_{k=0} \gamma^k r_{t+k}$ is the total accumulated return from time step t with discount factor $\gamma \in (0, 1)$. The goal of the agent is to maximize the expected return from each state s.
The action value $Q^\pi(s, a) = \mathbb{E}[R_t|s_t = s, a]$ is the expected return for selecting action a in state s and following policy π. The optimal value function $Q(s, a) = max_\pi Q^\pi(s, a)$ gives the maximum action value for state s and action a achievable by any policy. Similarly, the value of state s under policy π is defined as $V^\pi(s) = E[R_t|s_t = s]$ and

is simply the expected return for following policy π from state s.

There are model based and model free methods in reinforcement learning but since most popular effective algorithms are based on model free techniques we stick to that.

In value-based model-free reinforcement learning, the methods involve solving the Bellman equation iteratively to arrive at the solution. In these methods the action value function is represented using a function approximator, such as a neural network. Let $Q(s, a; \theta)$ be an approximate action-value function with parameters θ. The updates to θ can be derived from a variety of reinforcement learning algorithms. One example of such an algorithm is Q-learning, which aims to directly approximate the optimal action value function:$Q(s, a) \approx Q(s, a; \theta)$. In one-step Q-learning, the parameters θ of the action value function $Q(s, a; \theta)$ are learned by iteratively minimizing a sequence of loss functions, where the i^{th} loss function defined as

$$L_i(\theta_i) = \mathbb{E}(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i))^2$$

where s' is the state encountered after state s. We refer to this method as one-step Q-learning because it updates the action value $Q(s, a)$ toward the one-step return $r + \gamma \max_a Q(s, a; \theta)$.

In contrast to value-based methods, policy-based model-free methods directly parameterize the policy $\pi(a|s; \theta)$ and update the parameters π by performing, typically approximate, gradient ascent on $\mathbb{E}[R_t]$. One example of such a method is the REINFORCE family of algorithms due to Williams (1992) [10]. Standard REINFORCE updates the policy parameters θ in the direction $\nabla_\theta log\pi(a_t|s_t; \theta)R_t$, which is an unbiased estimate of $\nabla_\theta \mathbb{E}[R_t]$. It is possible to reduce the variance of this estimate while keeping it unbiased by subtracting a learned function of the state $b_t(s_t)$, known as a baseline (Williams, 1992), from the return. The resulting gradient is $\nabla_\theta log\pi(a_t|s_t; \theta)(R_t - b_t(s_t))$. A learned estimate of the value function is commonly used as the baseline $b_t(s_t) \approx V_\pi(s_t)$ leading to a much lower variance estimate of the policy gradient. When an approximate value function is used as the baseline, the quantity $R_t - b_t$ used to scale the policy gradient can be seen as an estimate of the advantage of action a_t in state s_t, or $A(s_t, a_t) = Q(s_t, a_t)V(s_t)$, because R_t is an estimate of $Q_\pi(s_t, a_t)$ and b_t is an estimate of $V_\pi(s_t)$. This approach can be viewed as an actor-critic architecture where the policy π is the actor and the baseline b_t is the critic.

III. Benchmark for Deep RL

Deep Reinforcement Learning algorithms are usually tested out on game environments like Atari games and Physics MuJoCo Simulators (simulations on learning to walk) due to two main reasons. One is that these learning tasks have proved to be difficult for the traditional reinforcement learning algorithms due to the high dimensionality of the state-action spaces and due to the complexity of the learning tasks, so they would prove to be challenging for any learning algorithm. Secondly there are many readily available game engines written in different languages that provide simple clean interfaces which makes it flexible and easy to test out all aspects of an algorithm. For our project we mostly concentrate on the performance of these algorithms on the Atari games as apart from being easy to use in many languages they provide more tractable and feasible learning challenges. Agents learn to play these games mostly by taking the raw screen images/frames as input. That is a set of frames at each step are considered to constitute a state. Figure 1 shows the screenshots of some Atari games.

IV. Recent Work

In deep reinforcement learning, the function approximators used are deep neural networks (convolutional neural networks for games). Now there are two main issues of using deep neural networks as function approximators with the existing RL algorithms like Q-Learning and Policy Gradient algorithms. Note that it has been shown that even with linear function approximators the algorithms like Q-Learning may not converge.

1) The data/experiences generated in a reinforcement learning scenario are correlated. Using

Fig. 1: Screenshots of popular games like Breakout and Space Invader-on the first row

this data to perform stochastic gradient updates makes it difficult for the neural networks to converge making the learning unstable. So when fitting/optimizing a neural network by stochastic gradient methods we preferrably want to use IID data.

2) Reinforcement Learning is generally not a very fast learning method involving iterative updates of an evaluation followed by some improvement. Couple this with the high training time and data required for deep neural networks, we can see that scaling these algorithms becomes an issue. They would take a long time to train and would require huge amounts of data.

In this recent domain of deep reinforcement learning there have been three main approaches/algorithms introduced which have great success for discrete action space games especially, in fully observable MDP scenarios. We have mainly concentrated on them in this project. Continuous action spaces are more difficult to deal with often requiring a lot of computational resources [5][2]. Also another algorithm considering the Atari game as a Partially Observable MDP has also shown good results. A short overview of them is given below.

A. Deep Q-Learning

In this algorithm in [7], a convolutional neural network is used as a function approximator for the Q-Learning algorithm. But in order to tackle the issue of instability due to non IID data used for gradient updates two methods are used.

Experience Replay: The experiences (including the state, action taken in the state and reward obtained) are saved in a replay memory buffer. The buffer is filled and if any new experience arrives the old ones are popped out. Now instead of using the immediate experiences for the gradient updates as in the case of the normal Q-Learning algorithm here a batch of experiences are sampled from the replay memory and then this data is used for the gradient updates. This helps in ensuring that the gradient updates are sufficiently uncorrelated as experiences sampled are mostly far apart and not related to each other.

Target network: In the Q-Learning algorithm the Q function approximator used to choose an action, also generates the TD targets and the updates that we get as a result of this are used to update the same function approximator. This again induces correlation in the updates. When $Q(s_t, a_t)$ is increased, $Q(s_{t+1}, a)$ for all a also would mostly change in the same way hence increasing the TD targets which might lead to divergence of the networks. To combat this two separate networks, one target network - the network that would generate the TD targets for the update and the normal network - network which chooses action, where the gradient updates happen are used for every step. After every few steps the parameters of the updated network are copied on to the target network. This ensures that the updates do not affect the targets immediately ensuring more stability in learning.

$$L_i(\theta_i) = \mathbb{E}\Big[\big(r+\gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\big)^2\Big]$$

On differentiation $\nabla_{\theta_i} L_i(\theta_i) =$

$$\mathbb{E}\Big[\big(r+\gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta_i)\big)\nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

The above equations show the loss function and the corresponding gradients w.r.t parameters. θ^-

represents the parameters of the target network. The batch of gradients are then used for the Q-learning updates.

B. *Asynchronous Actor-Critic*

In this algorithm in [6] multiple agents try to learn simultaneously using Actor Critic updates. Each agent corresponds to a thread and we spawn multiple threads that perform operations asynchronously and parallely. Each thread has their own network (value network and policy network). There's a common shared parameter server which stores a copy of network parameters. Now as the agents run the actor critic algorithm they generate gradients which would be used to change the network parameters using some stochastic gradient updates. But instead of updating their own parameters the threads send their updates to a common parameter server where the parameters are updated.

All these operations are performed asynchronously and parallely meaning the threads are not synchronized while updating the common parameter server. So some updates may collide and we may lose those updates but because of the HogWild property mentioned in [6] in an expected sense we would still be able to learn well. Before performing a set of actions again to get the next updates each thread synchronizes its parameters with the shared parameters .

The parallelism in the algorithm serves two purposes. One is that it increases the training speed as more training steps are done with more agents. Secondly the parallel agents will more likely be seeing different parts of the state space at any point of time as they follow stochastic policies, so the updates they make would be independent of each other thus overcoming the issue of divergence due to non IID updates. Following are the actor-critic gradient accumulation updates performed by each agent.

$$d\theta \leftarrow d\theta + \nabla_{\theta'} log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$$

$$d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$$

For Atari games in [6] and [7], four consecutive screen frames of the game are taken to be a state. This ensures that the state is able to capture the temporal aspects of the game, for example if a ball is moving in a game like breakout we can get a sense of direction and speed with which the ball is moving by using the four frames. So the Atari games are formulated as fully observable MDPs and use the algorithm. A convolutional neural network is used as a function approximator that enables them to learn good representations of the environment.

C. *Trust Region Policy Optimization*

In the usual policy iteration and policy gradient algorithms a lot of learning and unlearning happens meaning every update does not necessarily maximize the reward function. So in [9] they have tried to overcome this by attempting to make every update to actually improve on the rewards we get. Using the identity,

$$\eta(\widetilde{\pi}) = \eta(\pi) + \mathbb{E}_{r\sim\widetilde{\pi}}[A^{\pi}(s_0,a_0) + A^{\pi}(s_1,a_1) + ...]$$

where $\eta(\pi) = \mathbb{E}[R|\pi]$ and $A(s,a)$ is the advantage function. Considering a parameterized policy, making some approximations we get a first order approximation $L(\theta)$ to $\eta(\pi)$ in parameter θ (full theoretical proof in [9]).

$$L_{\theta_{old}} = \mathbb{E}_s\big[\sum_{t=0}^{T-1}\mathbb{E}_{a\sim\theta}\big[\frac{\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta_{old})}A^{\theta}(s_t,a_t)\big]\big]$$

Then using conservative policy iteration in [4] a theorem in [9] shows the following,

$$\eta(\theta) \geq L_{\theta_{old}} - C\max_s D_{KL}[\pi(\cdot|\theta_{old},s)||\pi(\cdot|\theta,s)]$$

where D_{KL} is the Kullback-Leibler divergence between probability distributions. So the basic concept used is to optimize the right hand side which will increase the total reward. But in order to make this algorithm practical many more approximations are made [9].

D. *Deep Recurrent Q-Learning for POMDPs*

Unlike the previous algorithms this algorithm in [1] makes agents learn to play Atari games by formulating them as a POMDP. The Atari games form a POMDP when only one screen frame is used as a input/observation (loses temporal information) and we take actions based on the history of the observations/screen frames. To accomplish this task again the state is input to a convolutional neural network but in this case a recurrent neural network is added (LSTMs here) after the convolutional neural network. They use the deep Q-Learning algorithm seen before but try to reduce the gap between $Q(O, a; \theta)$ in the POMDP case and the $Q(s, a; \theta)$ with the deep recurrent Q-networks. The idea is to select random episodes from the replay memory and use these experiences for the updates while retaining and carrying forward the hidden states in the RNNs. This memory property of RNNs would help in generating Q values related to the history of observations. Figure 2 shows the architecture used.

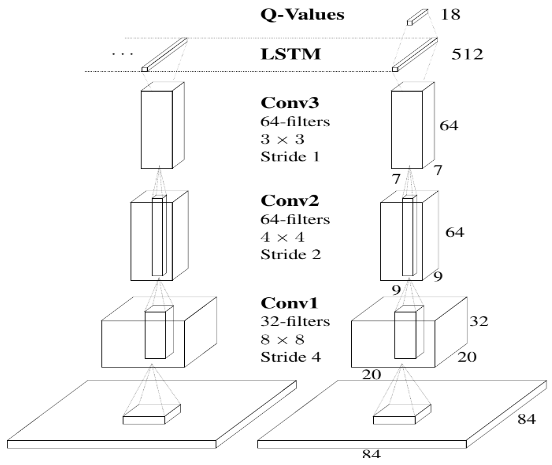


Fig. 2: The neural network architecture used in [1]

V. Implementation and Results

The trust region policy optimization algorithm does not provide better results than the others. For the implementation of the other two algorithms to make agents play Atari games we used Python with Tensorflow-Deep Learning Library. For the algorithms the input to the neural networks-the states were the four most recent screen frames of the games as explained before. For the Atari game simulators OpenAI gym python libraries were used which provide a simple way to manage the games and the interactions with the game.

The Deep Q-Learning algorithm was implemented with the architecture mentioned in [7]. The code was run on a server with a K40 GPU for the neural network training. But results obtained were not comparable to that of the paper's. This might be because the authors have trained the algorithm for a week whereas we were not able to access the GPU for such long periods of time. Even in the 1/2 days we could train continuously the progress in learning to get more rewards was not as expected.

Asynchronous actor critic algorithm implementation : In this implementation, all threads performed updates after every 5 actions. The agents used a network which had a convolutional layer with 16 filters of size 8x8 with stride 4, followed by a convolutional layer with 32 filters of size 4x4 with stride 2, followed by a fully connected layer with 256 hidden units. All three hidden layers were followed by a rectifier nonlinearity. In order to reduce the number of parameters the value network and the policy network shared these set of parameters. The output of this network is fed to a softmax layer corresponding to the number of actions for the policy network and to a linear layer for the value network. The state/the set of 4 images are the inputs to the network. Shared RMS prop was used for the stochastic updates (shared because the g parameter in RMS prop is also shared). A learning rate of 0.0007 was used which was linearly reduced to zero over the course of the training. Python has an issue of not supporting multi-threading meaning that only one thread handles the entire python process. To overcome this we had to use multi-processing where we spawn multiple processes instead of threads. But on doing this there is an issue of sharing parameters that arises as processes normally do not share memory like threads. To combat this we needed to use special multiprocessing shared arrays that only handle 1-D data and hence we stored the shared parameters in a 1-D array. So we had to shape and reshape the thread parameters everytime we needed to update the shared parameters. We used the algorithm on three games-Breakout, Pong and Beamrider. We used around 30 processes for Breakout and Beamrider while around 24 processes for Pong. We trained the network on each of them separately for more than a day each over about 80 million steps. Once fixed except for the number of actions for each game we did not change the hyperparameters between games. Due to the time taken to train these games we restricted to training and analyzing only three games.

Results: The results obtained on training with the Asynchronous Actor Critic algorithm for 80 million training steps were comparable to that in the paper [6]. They are shown in figure 3. The agents trained were able to reach human level of play.

VI. Proposed Modifications and Experimental Results

Many ideas and approaches were tried of which we explain two main approaches.

A. Experience Replay

One of the main concerns with deep reinforcement learning is that the data used to perform gradient updates may get correlated which destabilizes the network. So to avoid this, experience replay was used in [7] and parallelism was necessary in [6]. To improve the stability we combined this experience replay concept in the Asynchronous Actor Critic Algorithm. So a replay memory buffer was given to each agent/thread from which experiences were stored and sampled for the updates. Algorithm 1 with the same architecture as before was used by each actor learner process.

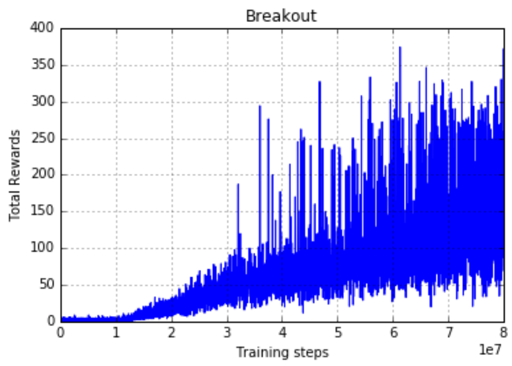
Results : Using this concept did not improve the training and results much. This might be because the parallelism itself made sure that the data is IID enough. Also the experience replay memory used was small due to memory constraints. Plus addition of these operations slows the algorithm further making it not worth the stabilization it provides if any. But we feel that this concept of experience replay would be very helpful on some games (maybe some more complicated games) where the agents are not able to explore the state space well, in which case the data would be correlated and hence this would be useful.

B. Batch and Weight Normalization

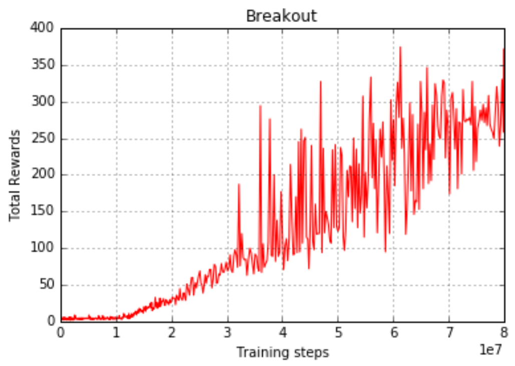
Batch Normalization : Batch Normalization introduced in [3] has shown tremendous emperical

Algorithm 1 Pseudocode for each actor-learner process with experience replay

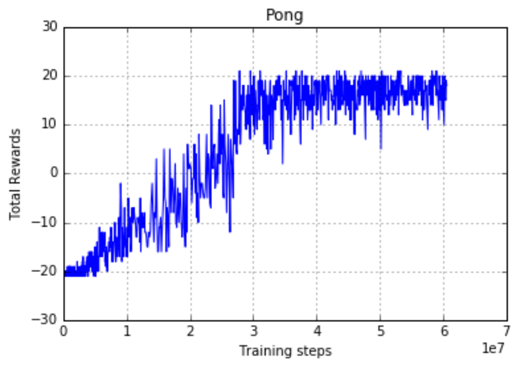
```
//Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$ and load experience replay memory

repeat
    Reset gradients : $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        Pop oldest experience from replay buffer
        Push $[s_t, a_t, r_t, V(s_t;\theta'_v)]$ in replay buffer
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0, & \text{for terminal } s_t, \\ V(s_t, \theta'_v), & \text{for non-terminal state } s_t \end{cases}$
    Choose a random integer $t_e$ from $[0, \text{replay memory size-1} - t_{max}]$
    for $i \in \{t_e, ...., t_e + t_{max}\}$ do
        Take experience i from buffer - $[r_i, s_i, a_i, V(s_i;\theta'_v)]$
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$ :
        $d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$ :
        $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
    end for
    Perform asynchronous update on shared parameters $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$
until $T > T_{max}$
```



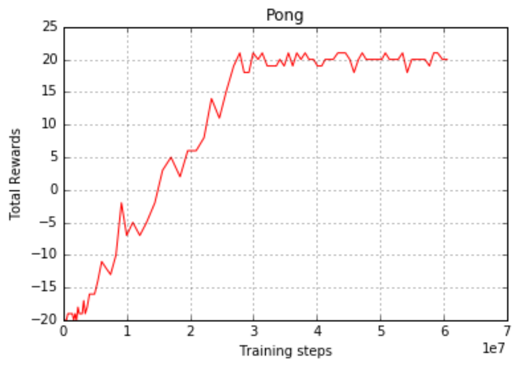
(a) Rewards collected for each episode



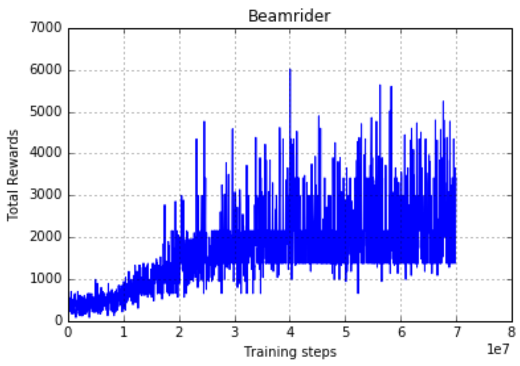
(b) Max of rewards collected over 10 episodes



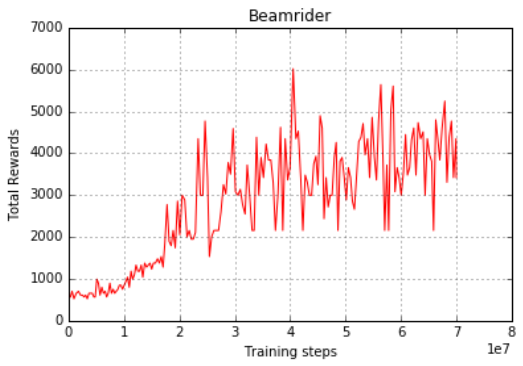
(c) Rewards collected for each episode



(d) Max of rewards collected over 10 episodes



(e) Rewards collected for each episode



(f) Max of rewards collected over 10 episodes

Fig. 3: Results obtained on training using the Asynchronous actor critic algorithm for 3 different games. It shows the scores at different training steps for the games.

results in improving the training speeds of Deep Neural Networks. An internal covariance shift (the change in the distribution of network activations due to the change in network parameters during training) occurs for each subnetwork, which makes training difficult as parameters need to change to considering these shifts. To handle this the input to each activation is normalized as follows:
For a layer with d-dimensional input $x = (x^{(1)}, ..., x^{(d)})$, we normalize each dimension,

$$\widehat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

For instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the nonlinearity. To address this, it is made sure that the transformation inserted in the network can represent the identity transform. To accomplish this, for each activation x^k , a pair of parameters are introduced γ^k, β^k, which scale and shift the normalized value: $y^k = \gamma^k x^k + \beta^k$. These parameters are learned along with the original model parameters, and restore the representation power of the network. This is applied for each mini-batch during training to make it practical.

Issues : But on applying this to networks in Deep RL the algorithms seem to actually worsen the training rather than improve it. The networks most often seem to diverge after some time else do not seem to learn well at all. This might be because in [3] it has been tried mainly on supervised learning scenarios where the training inputs are usually IID. But with reinforcement learning this is usually not the case. So the dependent correlated inputs probably are causing this instability.

Weight Normalization : The weight normalization concept in [8] was seen to improve the training speed in deep neural networks that deals with noisy data. So it could be used on the deep RL networks when considering the non iid behaviour of the data in the RL scenario to be noisy.

For this weight normalization the weight vectors that deal with inputs to each activation function/node are taken to be the product of a magnitude and a unit vector which are controlled separately.

$$w = \frac{g}{||v||} v$$

where g is a scalar and v is a vector and so $||w|| = g$.

Here [8] proposes to explicitly reparameterize the model and to perform stochastic gradient descent in the new parameters v, g directly. Doing so improves the conditioning of the gradient and leads to improved convergence of the optimization procedure. By decoupling the norm of the weight vector (g) from the direction of the weight vector ($v/||v||$), the convergence of the stochastic gradient descent optimization speeds up.

Reparameterizing the network increases the number of variables. But despite this we can get direct equations relating the gradients of weight vector w with those of v and g as shown below. Hence it does not require any extra computation for getting the gradients of the added variables.

$$\nabla_g L = \frac{\nabla_w L.v}{||v||}, \quad \nabla_v L = \frac{g}{||v||} \nabla_w L - \frac{g \nabla_g L}{||v||^2} v$$

The algorithm can be seen in Algorithm 2 where θ_v and θ are merged to get w. The same network architecture used for asynchronous actor critic implementation is used by each process.

Results :
With this concept of weight normalization we were able to improve upon the Asynchronous Actor Critic algorithm. It improved the performance of the agents, mainly for the game of Breakout. The results we got on Breakout after training the games with this modified algorithm can be seen in Figure 4. In fact the new changed algorithm is able to get rewards of more than 100 while the usual algorithm did not even reach rewards of 50. So the training speed is almost twice as fast after normalization.

VII. Conclusion

The recent algorithms in Deep Reinforcement Learning to make agents learn to play Atari Games from raw screen images were explored. Implementation and results of the algorithms gave insights regarding their advantages, disadvantages, feasibility and applicability in practical situations.

Algorithm 2 Pseudocode for each actor-learner process with weight normalization

```
//Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared counter $T = 0$
// Assume thread-specific parameter vectors $\theta'$ and $\theta'_v$
Initialize thread step counter $t \leftarrow 1$ and load experience replay memory

repeat
    Reset gradients : $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    Convert to network weights with $w = \frac{g}{||v||}v$
    $t_{start} = t$
    Get state $s_t$
    repeat
        Perform $a_t$ according to policy $\pi(a_t|s_t;\theta')$
        Receive reward $r_t$ and new state $s_{t+1}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal $s_t$ or $t - t_{start} == t_{max}$
    $R = \begin{cases} 0, & \text{for terminal } s_t, \\ V(s_t, \theta'_v), & \text{for non-terminal state } s_t \end{cases}$
    for $i \in \{t - 1, ...., t_{start}\}$ do
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients wrt $\theta'$ :
        $d\theta \leftarrow d\theta + \nabla_{\theta'} log\pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients wrt $\theta'_v$ :
        $d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i;\theta'_v))^2/\partial\theta'_v$
    end for
    Convert to gradients wrt $g, v$ $\nabla_g L = \frac{\nabla_w L.v}{||v||}$, $\nabla_v L = \frac{g}{||v||}\nabla_w L - \frac{g\nabla_g L}{||v||^2}v$
    Perform asynchronous update on shared parameters using $\nabla_g L$ and $\nabla_v L$
until $T > T_{max}$
```

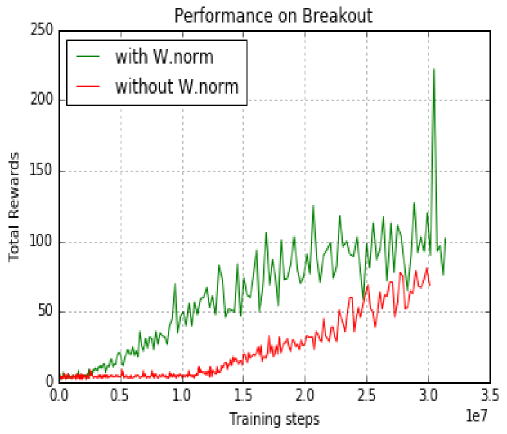


Fig. 4: Comparing the performance with weight normalization

It was clearly seen from the way we are able to learn to play Atari games from just the raw high dimensional sensory inputs, that these algorithms are much more capable of learning complex high dimensional tasks than the previous known algorithms. Some new modified algorithms were proposed, of which the usage of weight normalization has been shown to have improved upon the existing algorithm.

VIII. Future Work

For our subsequent work we would be focusing in two different directions. One direction is to improve the algorithms further preferably by decreasing training time/number of parameters used. For this the following ideas are to be checked out:

- As it can be seen in Figure 3 the initial part of training shows the slowest learning rate (less rewards in more number of time steps). So in order to improve this we will use transfer learning or supervised learning on existing data to make better network initializations. Also we also would try to use better exploration techniques like Thomson's sampling and UCB to improve the training speed.
- Next in most games and practical scenarios there's a background which is usually static over the course of few actions and there

are constant moving components in the foreground. So we would like to separate the background and the moving foreground from the screen frames and capture the temporal aspects of the moving parts in such a way that we can reduce the number of parameters used.

Secondly, we would like to apply these algorithms in a real-life practical application. For this we are looking at the problem of allotting cloud computing resources dynamically to clients. Efficient resource allocation in real time would save a lot of money and help in better utilization of resources. This problem has a big state space and is a good candidate for applying deep reinforcement learning because of the availability of a cloud computing simulator known as CloudSim which would enable us to train the data hungry deep neural networks.

References

[1] Matthew Hausknecht and Peter Stone. "Deep recurrent q-learning for partially observable mdps". In: *arXiv preprint arXiv:1507.06527* (2015).

[2] Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. "Memory-based control with recurrent neural networks". In: *arXiv preprint arXiv:1512.04455* (2015).

[3] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[4] Sham Kakade and John Langford. "Approximately optimal approximate reinforcement learning". In: *ICML*. Vol. 2. 2002, pp. 267–274.

[5] Guy Lever. "Deterministic policy gradient algorithms". In: (2014).

[6] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[8] Tim Salimans and Diederik P. Kingma. "Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks". In: *CoRR* abs/1602.07868 (2016).

[9] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. "Trust Region Policy Optimization". In: *CoRR* abs/1502.05477 (2015).

[10] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.