Asynchronous Methods for Deep Reinforcement learning

The Vanilla DQN uses computationally heavy techniques like Experience Replay which results in higher training time and memory. Improvements to the method have been published since then which include algorithmic changes (Double DQN), modifications to training methods (Prioritized replay) and architectural changes (Duelling networks). This paper proposes a lightweight framework for deep reinforcement learning using Asynchronous gradient descent for optimisation of deep nets. It shows that neural nets can be trained for a variety of problem setups and for different algorithms - be it value/policy based methods, on-policy/off-policy methods.

Training samples are correlated as the successive states are very similar in representation and as a result neural nets trained in this setup tend to diverge. To fix this issue, traditional DQN used Experience replay to reduce correlation between training samples. Another way to reduce correlation is to use asynchronous learners. Multiple agents can learn simultaneously on different instances of the environment. Each of them can explore different parts of the environment with different exploratory policies. This stabilises learning as it decorrelates updates and the distribution tends to be more stationary. Other advantages of asynchronous learners include - reduction in training time and on-policy methods like SARSA, Actor-critic can be used. Gorila framework also uses different agents, each with its own replay memory and environment, and gradient information from each agent is used to update a central model. Here, instead of separate machines and a central parameter server, multiple threads on a single CPU is used. This reduces overheads in parameter/gradient sharing. This paper talks about asynchronous variants of 4 different RL algorithms: one-step Q-learning, one-step SARSA, n-step Q-learning and Advantage Actor-critic.

Async one-step Q-learning is a simple extension from what DQN uses. An older version of the network is used to define targets as is the case with DQN. Gradients are accumulated over multiple timesteps (~ minibatch update) and this reduces overwriting of updates (using the HOGWILD! framework). Each thread uses a different exploratory policy and this results in more robust policies. Similar techniques are used for async one-step SARSA, the difference being Q(s', a') being used instead of max Q(s', a') which is used in Q-learning.

Async n-step Q-learning uses n-step returns and it operates in forward view i.e the algorithm selects actions till t_{max} time-steps into the future or until a terminal state is seen and then gradients are computed for all (s, a) since last update. As expected, the last state uses a one-step return, the penultimate state uses a two-step return and so on - total of t_{max} updates.

Async Advantage Actor-Critic (A3C) is the trademark algorithm of this paper as it performs the best among the 4 methods discussed. Here an estimate of policy (actor) as well as value function (critic) is maintained. It uses n-step returns as in the previous case and the policy and value function parameters are updated after t_{max} steps. Update rule reduces to $grad_{\theta'}[\ log(\pi(a_t, s_t; \theta'))\ A(s_t, a_t; \theta, \theta_v)\ ]$ where $A(s_t, a_t; \theta, \theta_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta_v) - V(s_t; \theta_v)$ is the advantage function which represents the advantage of taking action a_t from state s_t. Also k is upper bounded by t_{max}. In practical cases, the parameters of policy and value functions are shared even though they can hold separate parameters. Also it was observed that adding entropy of the policy π to the objective function improved exploration and resulted in avoiding sub-optimal solutions. Totally, 3 different optimization techniques - SGD with momentum, RMSProp (no stats) and RMSProp (with shared stats) were tried in this paper. RMSProp (with shared stats) was more robust than the other two.

We can infer a number of things based on the experiments conducted. First, neural networks were successfully trained using the asynchronous variants mentioned above. They learnt faster than DQNs with reduced memory and optimal policies were obtained by training it on 16 CPU cores. It was observed that for some games, n-step methods learnt faster than one-step methods - maybe because bootstrapping works better if targets span over a large temporal range. Policy-based A3C is better than the other 3 value-function based methods - maybe the policy parameters are easily learnt from the sequence of observations whereas in some games defining value functions adds unnecessary complications to the solution. A3C can easily be extended to continuous action space and when tried on such problems, it performed well and found good solutions in less time.

Altering the number of agents resulted in increase in speed - pointing to the fact that it scales well with the number of parallel workers. Asynchronous one-step methods showed an unexpected behaviour by showing superlinear speedups with increase in number of agents. This maybe because of the fact that - when multiple agents are used, the bias in estimating value function parameters reduces. Due to reduced bias, learning process is speeded up and thus results in better solutions. Analysing asynchronous methods in terms of robustness gives an interesting result. A range of learning rates were valid for each { method, game } and resulted in solutions with similar performances. This means asynchronous methods are quite robust to the choice of learning rates and random initialisations.

Even though these methods show better or at least comparable performances with other state-of-the-art methods, there are a lot of things which can be modified or tried out. The paper eliminated the option of Experience replay on grounds of computational cost. But it can be used to increase data efficiency as we can reuse old data and off-policy methods can be used. Some architectural improvements can be suggested just like the case with DQN. Duelling networks tend to obtain more accurate estimates of Q-values and hence can be used to improve performances of value-function based methods.

Learning to play in a Day: Faster Reinforcement learning using Optimality Tightening

The traditional DQN takes a significant amount of time and memory in order to be trained (Mnih et al. 2015). While more novel methods like the Asynchronous methods for Deep learning(described in above section) have arised, this paper aims to address this problem using the notion of *optimality tightening.* In the initial stages of training, the progress is very slow since the only signals for training are the rewards which can become a problem if they are sparse in nature. The core idea behind this paper is to propagate these rewards faster while also using a 1-step Q-learning based update.

If faster reward propagation is indeed the main motive, then one could have possibly tried to use n-step Q-learning or TD(λ) based Q-learning update. The asynchronous methods for deep learning paper(Mnih et al. 2016) in fact tried n-step Q learning and it performed slower than the 1-step Q-learning update(see Table 2 of paper). The reasoning for the same could be possibly because the n-step Q-learning update has a higher variance and as a result was probably slower. One possibility to try out would be vary(reduce) the size of *n* during the updates but a parameter decay schedule for the same may be non-trivial. Hence, the paper comes up with an optimality tightening mechanism whereby they constrain the Q function, to lie within a certain interval. With a modified loss function for the Deep network to optimize, the learning progress is shown to be faster.

The original equation to optimize is the follows : $min \sum_{s_j,a_j} Q_\theta(s_j,a_j) - y_j)^2$, where $y_j = r_j + \gamma \max_a Q_{\theta^{-1}}(s_{j+1},a)$

Two conditions are noted in the paper, which are then used to modify the above update rule.:

$$Q^*(s_j,a_j) = r_j + \gamma \max_a Q^*(S_{j+1},a) \geq r_j + r_{j+1} + \gamma^2 \max_a Q^*(S_{j+2},a)$$
$$\geq r_j + r_{j+1} + r_{j+2} + \gamma^3 \max_a Q^*(S_{j+3},a)$$

In general we can say that : $Q^*(s_j,a_j) \geq \sum_{i=0}^{k_1} \gamma_i r_j + i + \max \gamma^{k+1} Q^*(s_{j+k+1},a) = L^*_{j,k}$. Since this is valid, for all values of *k,* we apply this constraint for a range of values from [1,.....K]. Although it is clear to see that $L^*_{j,1} \geq L^*_{j,k} \quad \forall k$, and only 1 inequality suffices(and accounts for all other inequalities), we will apply all the inequalities with all $L^*_{j,k}$(all k) since the mentioned inequality need not hold for all values of *k* since we are using bootstrap estimates during training.

Similarly we can obtain an upper bound for $Q^*(s_j,a_j)$ using the same equation and the paper shows that we can obtain the inequality : $Q^*(s_j,a_j) \leq \gamma^{-k-1} Q^*(s_{j-k-1},a_{j-k-1}) - \sum_{i=0}^{k} \gamma^{i-k-1} r_{j-k-1+i} = U^*_{j,k}$

We can similarly observe that $U^*_{j,1} \leq U^*_{j,k} \quad \forall k$, and hence for the same reason, we will use all inequalities for k in [1,.....K] even though k=1 implies that the other inequalities are satisfied.

However, although all the inequalities are used, the authors of the paper note that smaller *k's* become more relevant in the later stages of optimality tightening, when our bootstrap estimates are more accurate. Hence it makes sense to use a larger *K* at the start when the bootstrap estimates of *Q(s,a)* are very inaccurate and later reduce the value of *K*

when our estimates become more accurate. We next define two terms we will consequently use which are : $L_j^{max} = \max_k L_{j,k}$, and $U_j^{min} = \min_k U_{j,k}$. These are the upper and lower bounds that we will look to introduce on *Q(s,a)*.Instead of enforcing the inequalities, to hold, the paper uses a loss function with two additional terms enforcing the two constraints. Hence the final loss function to optimize turns out as follows : $\min \sum \left[(Q_\theta(s_j, a_j) - y_j)^2 + \lambda (L_j^{max} - Q_\theta(s_j, a_j))_+^2 + \lambda (Q_\theta(s_j, a_j) - U_j^{min}))_+^2 \right]$

If we have a reward in the future, then the L_j^{max} term ensures that this reward propagates faster. This is because the derivative at s_j, a_j depends on terms in the future. Similarly, the U_j^{min} considers terms in the past which could possibly stabilize the learning. This constraints show a significant improvement in learning whereby the Q-learning updates still hold while we manage to speed up the rate at which reward signals propagate.

A high value of λ is set to effectively ensure that both the constraints are strictly respected. The authors also make note of how different λ could have been used and even different cost functions could be used and these variations are unexplored. One interesting design choice is to also use the return of every episode R_j, while estimating L_j^{max}. Using a high value of K(like using entire episode return) is impractical due increased memory and time complexities, and also because it may also not drastically improve the performance (since only lower value of k may be significant). However, by using the return R_j at every stage, we may achieve faster reward propagation in the initial stages of learning which is generally a critical task.

One more design choice the authors use is to set the hyper-parameter K=4, mainly motivated by factors of computational time and space. The authors suggest also sampling a subset from [1,.....K] in order to avoid this problem. An another possible alternative that could be attempted is to reduce the value of K with time, since the reward propagation is important particularly in the early stages.

The results have also been shown to work very well from an empirical standpoint also. The experiments are similar in many respects to the original DQN paper, with respect to the experiment setup(Experience replay, Delayed targets) and also the network architecture and algorithms used. While they manage to train the results drastically *faster* than the original DQN paper, what is surprising is that the nature of the *results are also significantly better* than the DQN's trained in the Nature paper in a number of games. One more important point to note is that while the theoretical results are justified for cases where the actions lead to deterministic states, the authors claim that the results will also hold if this is not the case too. The justification for the same lies in the fact that the math remains the same except that an expectation is applied during appropriate steps. Since we deal with random sample, the authors claim that the same procedure will approximate this expectation via random samples.

Dynamic Action Repetition for Deep Reinforcement Learning

This paper focuses on *ARR*(which is the number of times a particular action must be repeated after it is chosen . While the original vanilla DQN(Mnih et al. 2015) chooses 4 frames as the hyper-parameter, Braylan et al. , showed that games like Seaquest achieved a drastic improvement in performance, when this hyper parameter was altered(although the only expectation was to train more episodes). While at first, this seems non-trivial(since higher granularity of frames can achieve the same repetitive action by repeatedly applying the action), the *Frame Skip is a powerful Parameter,* paper reasoned that this modification to the hyperparameter allowed the model to learn more temporally distant causal effects of actions, which was crucial in games like Seaquest. However, the Dynamic Action Repetition paper conducts an analysis by not fixing the frame skip rate which was done in the other paper.

This dynamic actions papers builds on the idea that the ARR, can be a parameter of the model itself and need not be a fixed quantity. The advantage of this is twofold. Firstly, the model can make use of the higher granularity and superhuman abilities when required. Apart from this, the model can also learn relations from temporally distant states which was not possible when there existed a higher granularity. The paper then proceeds to look at a suite of On-policy and off-policy models which is made possible by the asynchronous methods for deep learning paper(Mnih et al. 2016).

The model setup had double the number of output layers whereby the one can choose to pick either a smaller or larger(ARR) at every move(each action has two output units, one for a small ARR and other for a larger ARR). While it is possible to triple or quadruple the number of possible choices of ARR, the time required for training increases due to a increased number of model parameters. However, even just two choices in the ARR showed a significant improvement in performance compared to a fixed action repetition rate. A critical design choice to note here is that the number of hidden units in the fully connected layer are also doubled(for only the Augmented DQN). This is because, the model requires more representational power since we are in effect doubling the number of possible actions we can choose. Hence the Neural Network may be required to learn better abstract representation of the value function or the policy representation.

In the experiments conducted, we see that this Augmentation has a clear performance benefit. In fact for the A3C algorithm, the performance benefit is observed without even doubling the number of units in the fully connected layer(hence having lesser representation power but more actions compared to the vanilla DQN). Something interesting to note is that the games that required long term strategies(Seaquest and Alien) reported a larger improvement in performance as a result of this augmentation(although this is only the case for Augmented DQN's). Also the table depicting the % of actions that used a higher ARR justified that the games used a judicious mix of the two ARR's.

However, a more concrete reason as to why exactly this works may be more insightful. It may be beneficial to look at the scenarios in which one ARR is prefered over the other. While performance benefits are clearly seen, it seems non-trivial as to why a smaller ARR cannot capture the same results as the larger ARR value. This gives us further motivation as to why we should look at stopping actions in between when we have a high ARR value. Possible explanation for this could be that the 1st order MDP assumption fails or alternately, the Q(s,a) function approximator does not have enough representation power.

Other areas to explore could be to look at reducing training time via dynamic action. While the number of parameters increase, we could possibly obtain a speedup(train more games) by obtaining an increased ARR on certain actions. Furthermore, we can also visually compare gameplays and look at the exact situation during which a higher ARR is chosen over sequence of lower ARR and see if this reveals anything new.