ACCELERATING EXPERIENCE REPLAY FOR DEEP Q-NETWORKS WITH REDUCED TARGET COMPUTATION

Bob Zigon[1] and Fengguang Song[2]

1 Beckman Coulter, Indianapolis, IN 46268
2 Department of Computer Science, Indiana University-Purdue University Indianapolis, Indianapolis, IN 46202

ABSTRACT

Mnih's seminal deep reinforcement learning paper that applied a Deep Q-network to Atari video games demonstrated the importance of a replay buffer and a target network. Though the pair were required for convergence, the use of the replay buffer came at a significant computational cost. With each new sample generated by the system, the targets in the mini batch buffer were continually recomputed. We propose an alternative that eliminates the target recomputation called TAO-DQN (Target Accelerated Optimization-DQN). Our approach focuses on a new replay buffer algorithm that lowers the computational burden. We implemented this new approach on three experiments involving environments from the OpenAI gym. This resulted in convergence to better policies in fewer episodes and less time. Furthermore, we offer a mathematical justification for our improved convergence rate.

KEYWORD

DQN, Experience Replay, Replay Buffer, Target Network

1. INTRODUCTION

Deep Q-networks (DQN) are a fundamental component of reinforcement learning that utilize Q-learning and deep neural networks. DQNs are applied to areas as diverse as game playing [1], portfolio management [2], scheduling [3], industrial control [4], robotics [5] and intrusion detection [6]. If a DQN is trained with samples from a problem space, they can leverage Q-learning theory to learn by trial and error. This makes DQNs widely applicable to many domains. By carefully crafting a goal (called the objective), the algorithm can generate a function that will work towards optimizing the objective without any user feedback. For example, if a DQN is applied to a game like checkers, it can then learn to beat the game in as few steps as possible. A DQN could also be given a graph that describes the paths between cities in the U.S. and the cost of traveling between any two adjacent cities (i.e. the traveling salesman problem [7]). If a given graph has N cities, the computational complexity of this NP-complete problem is O(N 22N). The DQN is tasked with learning how to generate good solutions to millions of travel problems and may run for hundreds or thousands of hours while performing this learning process. The point here is that the DQN did not need to be explicitly told what a good solution looks like. It simply optimizes the objective function to minimize travel time and learns by trial and error.

The downside to this unassisted behavior is that learning can happen very, very slowly. It can take nearly 24 hours on a GPU to train a DQN to consistently beat the Atari Pong game. In this

David C. Wyld et al. (Eds): CSML, NET, BDHI, SIPO, SOEA- 2023
pp. 01-13, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.130101

case, there is a very large solution space. Q-learning theory alone is not good enough to generate a robust solution. The algorithm needs a better representation of the DQN so that it can do a better job of generalizing to problems the DQN has not seen. By adding more "neurons", more layers, or more nonlinear activation functions to the network, the DQN generates a richer solution space.

The next important improvement to a DQN comes from adding a replay buffer and a target network to the overall structure. The replay buffer is used to store samples, as they are generated, for reuse later. The target network is a clone of the prediction network. The target network is updated by copying the prediction network on top of it at a low frequency. This low frequency update reduces parameter correlations with the prediction network that inhibits convergence.

This is where our research begins. We started with this approach involving the replay buffer and target network, and then asked the question "Is there a principled way to execute less logic and get better results?" In our approach we accepted the benefits of the replay buffer. What we rejected was the necessary information stored in the replay buffer. The replay buffer is supposed to make sample generation less expensive. The repeated fitting of the network to those samples is a necessity if you do not want your network to forget what it has learned. However, the classical DQN has a max() operator that is supposed to selfishly choose the next best action. This is where we focused our attention and then generated the following hypothesis: A DQN will converge to a better policy in less time when the number of max() operators is minimized.

A summary of our approach follows. We begin with the classical DQN algorithm that uses a replay buffer and a target network. After each new sample is generated, we immedi- ately compute the target value for the current state, action, and next state. We then save current state, action and target value to the replay buffer. This will eliminate any repet- itive computation of the target value and reduce the number of max() operator calls to one for each sample. The result is a better policy in fewer episodes and less time. Finally, we present a mathematical justification for our approach, as well as the results from three sets of experiments that demonstrate our improvement over Mnih.

Our contributions now include:

1. a new type of DQN that converges to an optimal policy faster than Mnih's approach,
2. an implementation that utilizes a new replay buffer format resulting in lower computational burden,
3. convergence in 21% fewer episodes and 35% less time,
4. and a mathematical argument that justifies the accelerated convergence.

This paper is organized as follows. Section 2 begins with background on reinforcement learning and section 3 presents related work. Our new approach, called TAO-DQN (Target Accelerated Optimization-DQN), is presented in section 4 along with a mathematical justification for its behavior. Section 5 consists of three experiments with results. Finally, section 7 presents our conclusions and describes our future work.

2. BACKGROUND

Reinforcement learning (RL) is a machine learning technique that allows an agent to interact with and learn from an environment to maximize the cumulative return. The goal is to learn good policies for sequential decision problems [8]. We can describe this with a Markov Decision Process (MDP) that is specified as a tuple (S, A, π, r, γ). At each time step t, the agent begins in state $s_t \in S$. After selecting an action a_t from a set of actions $A(s_t)$ according to the policy

$\pi t(st) \rightarrow at$, the environment advances to state st+1 with a reward signal of rt+1 and returns them to the agent. This process continues until the agent reaches a terminal state, all the while seeking to maximize the action value function of expected discounted return in equation 1

$$Q^{\pi}(s, a) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s, a] \quad (1)$$

for some discount factor $0 \leq \gamma \leq 1$.

The taxonomy of RL techniques [9] include Q-learning [10], temporal difference learning [11], Deep Q-networks [12,13] and Double Q-learning [14,15]. Q-learning maintains an estimate $Q : S \times A \rightarrow R$ of the optimal value function. Given a sequence of transition tuples (s_t, a_t, r_t, s_{t+1}), it updates $Q(s_t, a_t)$ towards the target y_t^J of

$$y_t^J = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a), \quad (2)$$

for each $t \geq 0$. With most problems being too large to learn all action-state pairs, we can instead learn a parameterized value function $Q(s, a; \theta_t)$, in which case the target y_t^{JJ} is

$$y_t^{JJ} = r_t + \gamma \max_{a \in A} Q(s_{t+1}, a; \theta_t). \quad (3)$$

Here the parameters are updated according to

$$\theta_{t+1} = \theta_t + \alpha(y_t^{JJ} - Q(s_t, a_t; \theta_t)) \nabla_{\theta_t} Q(s_t, a_t; \theta_t) \quad (4)$$

where α is the step size. As a result, Q-learning can identify an optimal action-selection policy for any MDP given infinite time. An optimal policy is a policy for action selectionthat maximizes future rewards

2.1. Deep Q-Networks

DQNs [16,17] combine a neural network function approximation and experience replay to create a scalable RL algorithm. The neural network takes a representation of the state as input, and generates a separate output for each possible action. Each output, predicted by the Q-values of the individual actions, corresponds with a given input state. This optimal action value function behavior obeys an identity known as the Bellman equation. If the optimal value $Q^*(s^J, a^J)$ of the sequence s^J at the next time step t was known for all possible actions a^J, then the optimal strategy is to select the action a^J that maximizes the expected value of $r + \gamma Q^*(s^J, a^J)$.

The neural network function approximation to the optimal value $Q^*(s^J, a^J)$, with weights θ, can be trained by minimizing a loss function $L(\theta_t)$ that changes with each time step t giving

$$L(\theta_t) = E_{s,a \sim \rho(\cdot)}[(y_t^{JJJ} - Q(s, a; \theta_t))^2], \quad (5)$$

where $y_t^{JJJ} = r + \gamma \max_{a'} Q(s_{t+1}, a^J; \theta^-)$ is the target and $\rho(s, a)$ is a probability distribution over sequences and actions. This target function, y_t^{JJJ}, uses target parameters θ^- which are updated every k steps with θ_t. The delayed update was discovered by Mnih, et al. [17] and proved important to convergence, along with the use of experience replay [18,19,20]. Stochastic gradient descent can then be used to optimize the loss function $L(\theta_t)$ with respect

to the parameters.

As previously mentioned, the other critical component of a DQN is experience replay. In many RL algorithms an experience is discarded after it is used to compute the loss function. In experience replay the agent's experiences at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$ are stored in a data set $D = e_1, e_2, \ldots, e_N$ and pooled over many episodes into a fixed size, circular replay buffer. As the DQN advances over time, a random subset of D is drawn and Q-learning updates are applied to these samples. This has the advantage of greater data efficiency because the samples are reused for training. The randomization also helps to lower the variance of the updates because any correlation between samples is broken. When many samples within the random subset are correlated, the overall information content is low which slows down training

3. RELATED WORK

Several researchers have tried to improve or accelerate the convergence rate of reinforcement learning. With memory replay having an important role in RL, Liu and Zou [19] chose to generate a deeper understanding of the underlying mechanism by reformulating it as a dynamical system using ordinary differential equations (ODE). They were able to derive an analytic solution to the ODEs for a simple problem. With that example they showed that the amount of memory allocated to replay can affect the agent's convergence. Zhang and Sutton [20] followed a similar path and introduced a new hyper parameter that they could study. Their resulting empirical study showed how large replay buffers can significantly hurt performance and then proposed a simple method to remedy the negative influence.

Fedus et al. [18] built on Zhang's work and introduced two new hyper parameters: the replay capacity and the ratio of learning updates to experience collected. Interestingly enough, their additive and ablative studies partially contradicted Zhang. Fedus found that greater capacity substantially improves the convergence of some algorithms while leaving others unaffected. Schaul et al. [21] took a different approach. They simply acknowledged that prior researchers uniformly sampled the experiences from the replay buffer. The approach replays transitions regardless of their significance to the learning process. Their solution was to develop a framework that replayed important transitions more frequently with the goal of learning and converging more efficiently.

4. OUR APPROACH

We first describe our implementation of the TAO-DQN algorithm. We will then explain the overestimation error that our approach addresses in the context of our implementation.

4.1. TAO-DQN

Our approach begins with the code in algorithm 1, the basic DQN. The outer loop on line2 advances over every episode of training data while the inner loop on line 5 steps throughtime and processes each sample. At a high level, the agent operates on the current state s to generate the action a. The environment then operates on the action to advance tothe next state of the MDP. As each sample is generated by the environment, the tuple (s, a, r, s_{next}) is saved to the replay buffer, where s_{next} is the next state. A collection of these tuples is known as a *trajectory*. Finally, the replay buffer itself is replayed and the prediction network is trained against the sampled subset.

Algorithm 1 Basic DQN algorithm

```
1: function DQN(M,T)
2: for episode 1 to M do
3: s = env.reset()
4: totalReward ← 0
5: for time_tep ← 1 to T do
6: a = agent.action(s)
7: s_next, reward, done = env.step(a)
8: if done then
9: break
10: end if
11: agent.save(s, a, reward, s_next, done)
12: agent.replay_sar()
13: totalReward ← totalReward + reward
14: s ← s_next
15: end for
16: Every C steps reset θ' ← θ
17: end for
```

Our improvement addresses the replay logic. The traditional replay logic is shown in algorithm 2 of Figure 1. Here the replay buffer is sampled and the target network is used to predict the value of the next state. The target values for each sample in the mini batch are generated from the (r, max_q) pair, where max_q is the maximum Q value of the next state across all actions. The prediction network then fits the sample using the neural network. We call this the (s, a, r) approach because the $target = G(s, a, r)$, where G is effectivelythe replay logic.

Our new implementation of the DQN algorithm is shown in algorithm 4. Here we makethe observation that the value of the next state, $next_q$, can be computed once in each time step and then its associated target value is also computed. This is shown on lines 11 and 12

Instead of saving the tuple (s, a, r, s_{next}) to the replay buffer, we save (s, a, t) where t is the target. This new implementation, called (s, a, t), causes the target accelerated replay logic in algorithm 3 to run approximately an order of magnitude faster, although it canbe more given that it is directly related to the size of the mini batch buffer. A line by line comparison of the traditional replay logic in algorithm 2, with the new logic in algorithm3, shows the logic that has been optimized out.

Algorithm 2 Traditional Replay logic

```
1: function replay_sar()
2: mini_batch = sample(replay_buf, batch_size)
3: state, action, reward = mini_batch
4:
5: next_q = target_net(next_state)
6: max_q = amax(next_q, axis = 1)
7:
8: for i in range(state.size) do
9:     if done[i] then
10:        target_q[i] = reward[i]
11:    else
12:        target_q[i] = reward[i] + γ · max_q[i]
13:    end if
14: end for
15: result = predict(state, action, target_q)
16: return result
```

Algorithm 3 Target Accelerated Replay logic

```
1: function replay_sat()
2: mini_batch = sample(replay_buf, batch_size)
3: state, action, target_q = mini_batch
4:
5: Expensive target value is not recomputed.
6: No max() operators are called.
7:
8: Targets are no longer recomputed.
9:
10:
11:
12:
13:
14:
15: result = predict(state, action, target_q)
16: return result
```

Fig. 1: The pseudo code for the traditional replay logic and the target accelerated replaylogic. The two listings are virtually identical except for lines 5 through 14. In the traditionalreplay logic the value of the next state is calculated on line 5, every time the replay functionis called, for every sample in the mini batch. The execution time for line 5 is proportionalto the *batch size* and the complexity of the Q function.

hen the replay sar() function is called in the traditional approach, the target Q function will be called *batch size* times so that the value of the next state can be computed. The computational complexity of algorithm 1 when calling the traditional replay logic is O($M \cdot T \cdot$ *batch size*), where M is the number of episodes to iterate over, and T is the maximum number of time steps per episode. For comparison, our new approach, TAO- DQN, has a complexity of O($M \cdot T$). Again, the difference lies in the fact that our new approach does not recompute the target values before fitting.

4.2. Addressing Overestimation

Many of the convergence properties for RL are based on empirical results. Both Bradtke [22] and Thrun et al. [23] derived mathematical conditions to suggest when the learning will fail. The key observation is that function approximators realized by DNN introduce

Algorithm 4 The pseudo code for the RT-DQN algorithm. Lines 11 and 12 show how the target is computed once for each item in the trajectory and then stored in the replay buffer. This is in contrast to lines 8 through 14 of the traditional replay logic in algorithm 2 that is constantly recomputing the targets.

```
1: function RTDQN
2: for episode ← 1 to episode_max do
3:     s = env.reset()
4:     totalReward ← 0
5:     for timestep ← 1 to T do
6:         action = agent.action(s)
7:         s_next, reward, done = env.step(action)
8:         if done then
9:             break
10:        end if
11:        next_q = target_net(s_next)
12:        target = reward + (1 − done) · γ · amax(next_q)
13:        agent.save(s, action, target)
14:        agent.replay_sat()
15:        totalReward ← totalReward + reward
16:        s ← s_next
17:    end for
18:    Every C steps reset θ' ← θ
19: end for
```

generalization error into the predictions. Such a generalization error can lead to *overesti- mation* of the action values [23]. The overestimation arises from a positive bias introduced by Q-learning, which approximates the maximum expected action value with the maxi- mum action value. Here, we leverage the work of [22,23] to minimize this overestimation, and present the reasons that the use of a single max() operator per time step of our algorithm is able to find a better policy in less time than Mnih.

In Watkin's original paper on Q-learning [10], the function $Q(s, a)$ was updated ac- cording to

$$Q(s,a) \leftarrow r_{s,a} + \gamma \max_{a'} Q(s', a'). \tag{6}$$

If the values are stored in a data structure like an array, this policy is capable of maximizing the expected cumulative reward precisely. However, when the function approximator approach is used we assume that some form of inaccuracy is introduced. We then have

$$Q^{approx}(s, a) = Q^{exact}(s, a) + \beta_{s,a}, \tag{7}$$

where $\beta_{s,a}$ is a collection of uniformly distributed random variables with $\mu_{\beta S,a} = 0$, $\sigma_{\beta S,a} = \epsilon_{s,a}$ and $Q^{exact}(s, a)$ are the exact target values.

Upon subtracting the approximate and exact forms we get a random variable Z with positive

mean, which was generated from zero mean error $\beta_{s,a}$. This represents the target approximation error.

$$Z = (r_{s,a} + \gamma \max_{a'} Q^{approx}(s', a')) - (r_{s,a} + \gamma \max_{a'} Q^{exact}(s', a')) \quad (8)$$

$$= \gamma(\max_{a'} Q^{approx}(s', a') - \max_{a'} Q^{exact}(s', a')) \quad (9)$$

$$= \gamma \max_{a'} \beta_{s',a'}. \quad (10)$$

The reasoning behind this positive mean for Z follows. Assume a single step of equation 6, and five actions to choose from as shown in Figure 2a. This shows an exact set of Q-valuesand their actions. In Figure 2b we see how the inaccuracies of a function approximator causes the Q-values to fluctuate about their exact values. The application of the max() operator, however, will always pick the largest Q-value. The result of equation 10 is that the max() operator generates overestimation because it does not preserve the zero-mean property, $\mu_{\beta_{S,a}} = 0$.

In order to compare the error between Mnih's approach and ours, we look first at the underlying Markov decision process. Table 1 shows the Q-value for each state and the

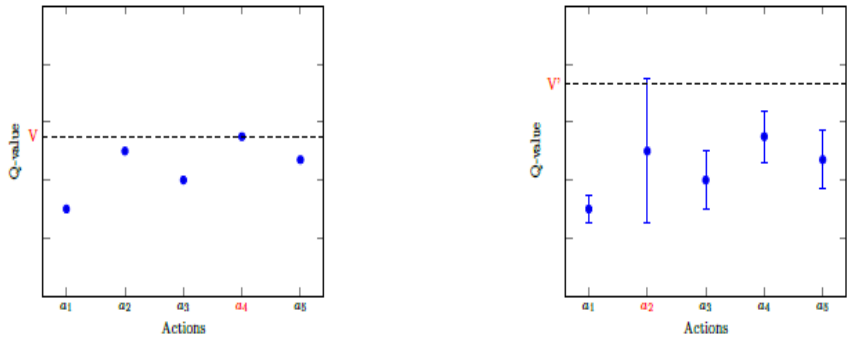

Fig. 2: Figure 2a is an example of actions and their respective Q-values. Since these are error free the max($a_1 \ldots a_5$) will return the correct value V for action a_4. In Figure 2bthere are error bars now present because of the use of the function approximator. We nowsee how the correct value V can be overestimated when performing max ($a_1 \ldots a_5$) wherethe value V' is returned for action a_2.

Table 1: Table of Q values and target approximation errors

Timestep t	State	$Q(s_t, a; \theta_t)$	Target Approximation Error
0	s_0	$r_0 + \gamma max_{a'} Q(s_1, a'; \theta_0)$	$\gamma \max_{a'} \beta_{s_1,a'}$
⋮	⋮	⋮	⋮
T-1	s_{T-1}	$r_{T-1} + \gamma max_{a'} Q(s_T, a'; \theta_{T-1})$	$\gamma \max_{a'} \beta_{s_T,a'}$

target approximation error as the agent advances to a new state s_t as a result of taking action

a_t.

Thrun et al. [23] showed that the average overestimation can be as large as $\gamma\epsilon\frac{n-1}{n+1}$, and this overestimation bounds the target approximation error as

$$\gamma \max_{a'} \beta_{s',a'} < \gamma\epsilon\frac{n-1}{n+1}, \tag{11}$$

where n is the number of actions to choose from and ϵ is the variance for a time step. In the context of Mnih's et al. [12] DQNs used to train Atari games, the overestimation for one episode of T time steps is shown in equation 12. Here the batch size is a constant.

$$batch\_size \sum_{j=1}^{T} \gamma \max_{a'_j} \beta_{s_j,a'_j} < batch\_size \sum_{j=1}^{T} \gamma\epsilon\frac{n-1}{n+1} \tag{12}$$

$$batch\_size \sum_{j=1}^{T} \gamma \max_{a'_j} \beta_{s_j,a'_j} < K \sum_{j=1}^{T} \gamma\epsilon\frac{n-1}{n+1} \tag{13}$$

The overestimation for our approach shown in equation 13 has introduced the adjustable parameter K, where $1 \leq K \leq batch\ size$. Therefore, our (s, a, t) based approach is mod- eled optimally by equation 13 with $K = 1$, and this is where we derive our reduced computational complexity from. With batch sizes on modern DQNs (in conjunction with the learning rate) getting larger to better control the variance in the results and improve training time, it is easy to see the merit in minimizing the number of max() operators used during training of a DQN that uses experience replay and a target network [24,25,26].

5. EXPERIMENTAL RESULTS

In the previous section we described our (s, a, t) approach, compared it to (s, a, r) and analyzed their overestimation error. In this section, we show how we perform three experiments where our target accelerated approach was compared to the traditional approach of Mnih. The results follow.

5.1. Mountain Car Results

A MountainCar is positioned between two mountains on a one dimensional track. The goalis to drive up the mountain on the right where the flag is located. However, the car's enginedoes not have enough power to achieve this in a single pass. Therefore, the solution is to drive forward and backward, slowly building more momentum, until the goal is achievedor you run out of time.

In Figure 3 we show the results of running the MountainCar-v0 environment 24 times for both (s, a, t) and (s, a, r). Each run takes approximately 5 hours on an Nvidia V100 GPU. When (s, a, t) has reached a score of 100, (s, a, r) is less than half away to its goal ($\rightarrow$).It then takes 32% more episodes for (s, a, r) to generate a solution equivalent to (s, a, t) ($\rightarrow$). The pink and light blue envelopes around the (s,a,r) and (s,a,t) approaches show how minimizing the number of max() operators can affect the variability [27] of the solution.

Figure 4 shows the graph of the run time. Here (s, a, t) executes in 68% of the timeof $(s,$

a, r). By removing the relatively expensive recalculation of the target values in the target network, we train nearly twice as fast.

Figure 5 demonstrates how changing the value of K in equation 13 shifts the graph from right to left. As the number of max() operators are reduced, the policy is generated with less work and in less time.

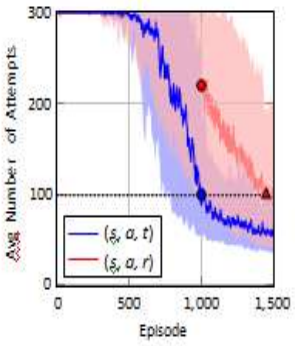

Fig. 3: The results are obtained by run- ning (*s, a, t*) and (*s, a, r*) with 24 differ- ent random seeds for MountainCar.

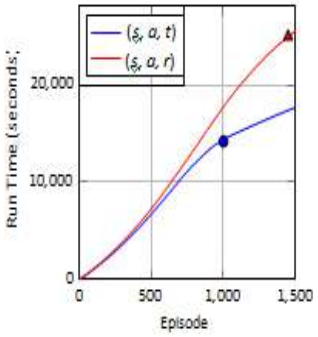

Fig. 4: Average run time across 24 runs for MountainCar. (*s, a, t*) finishes in approximately 68% of the time for (*s, a, r*).

5.2. Acrobot Results

The Acrobot is a planar two link robotic arm where the joint between the two links is actuated. Initially the links are hanging downward. The goal is to swing the end of the lower link up to a given height.

In Figure 6 we show the results of running the Acrobot-v1 environment 20 times for both (*s, a, t*) and (*s, a, r*). Each run takes approximately 1.5 hours on an Nvidia V100

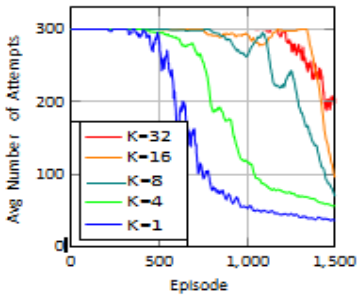

Fig. 5: MountainCar-v0 results for different K values. (*s, a, r*) is K=32 and (*s, a, t*) is K=1.

GPU. When (*s, a, t*) has reached a score of 100, (*s, a, r*) is still 44% from its goal (→). It then takes 21% more episodes for (*s, a, r*) to generate a solution equivalent to (*s, a, t*) (→).Once again, the pink and light blue envelopes around (s,a,r) and (s,a,t) show how the variance in the solutions can be controlled.

Figure 7 shows the graph of the run time. Here (*s, a, t*) executes in 76% of the timeof (*s, a, r*). Figure 8 demonstrates how changing the value of K in equation 13 shifts the graph

from right to left. As the number of max() operators are reduced, the policy is again generated with less work and in less time.

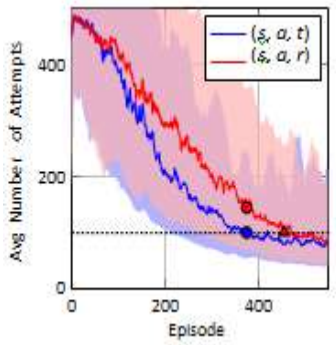

Fig. 6: The results are obtained by run- ning (s,a,t) and (s,a,r) with 20 different random seeds for Acrobot.

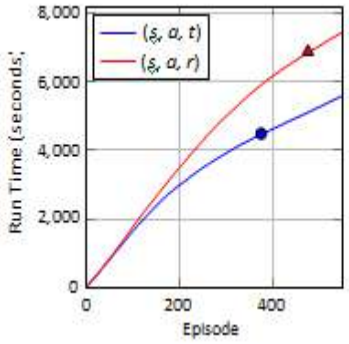

Fig. 7: Average run time across 20 runs for Acrobot. (s, a, t) finishes in approxi- mately 76% of the time for (s, a, r).

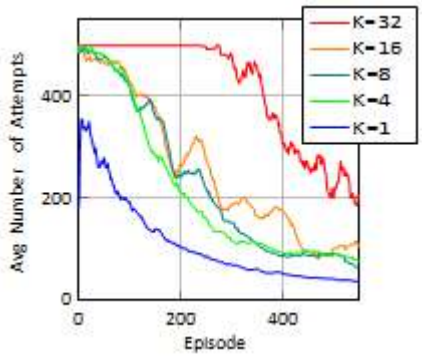

Fig. 8: Acrobot-v1 results for different K values. (s, a, r) is K=32 and (s, a, t) is K=1.

5.3. Cartpole Results

Cartpole is the classic inverted pendulum from control theory. The goal is to balance the pole for as long as possible by moving the base to the right or left.

In Figure 9 we show the results of running the Cartpole-v1 environment 20 times for both (s, a, t) and (s, a, r). Each run takes approximately 20 minutes on an Nvidia V100 GPU. When (s, a, t) has reached a score of 200, (s, a, r) is still 50% away from its goal(●→ ●). Similarly, it takes 35% more episodes for (s, a, r) to generate a solution equivalent to (s, a, t) (▲ →●). Since the goal of the previous two experiments is to minimize the time to achieve the goal, we collected and plotted the run time vs episode. Cartpole, on the other hand, is trying to balance the pole for as long as possible. In this case, it does not make sense to generate the run time graph. Since reporting some form of execution time generates insight into the behavior of TAO-DQN, we included three run times at points (a), (b), and (c) on Figure 9. These time points demonstrate the time improvement of (s, a, t) over (s, a, r). Once again, Figure 10 demonstrates how changing the value of K in equation 13 shifts the graph from right to left, improves the policy and causes cartpole to balance for a longer period of time.

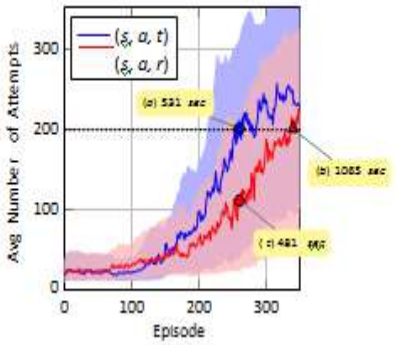

Fig. 9: Cartpole-v1 results with 20 random seeds.

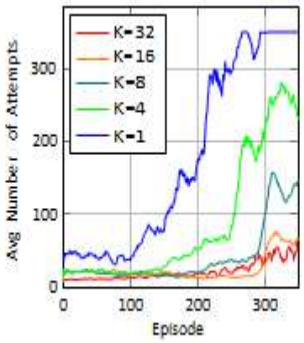

Fig. 10: Cartpole-v1 results for different K values. (*s, a, r*) is K=32 and (*s, a, t*) is K=1.

Table 2: Results for the MountainCar, Acrobot and Cartpole experiments. The episode ratio and time ratio columns demonstrate how inexpensive our (*s, a, t*) approach is when compared to the (*s, a, r*). Note: Smaller ratios are better.

Experiment	(s,a,t) episodes to goal	(s,a,r) episodes to goal	episode ratio	(s,a,t) time to goal (sec)	(s,a,r) time to goal (sec)	time ratio
MountainCar	1,000	1,450	0.68	14,200	25,000	0.57
Acrobot	375	475	0.79	4,500	6,900	0.65
Cartpole	260	340	0.76	531	1065	0.50

6. DISCUSSION

The behavior of (s,a,t) versus (s,a,r) is summarized in table 2. We reduced the number of episodes required by at least 21% (up to 32%). We also reduced execution time by at least 35% (up to 50%). These reductions are consistent with our hypothesis. It is gratifying to demonstrate that these improvements follow the prediction of the K parameter in equation [13] As K gets smaller, the overestimation get smaller and the algorithm executes faster. Figures 3, 6 and 9 further demonstrate how the pink envelope that describes the variance in the (s,a,r) approach shrinks to the tighter blue variance envelope of the (s,a,t) approach. Again, this is attributed to the K parameter that governs the number of max() operators applied during replay.

7. CONCLUSIONS AND FUTURE WORK

This paper introduced a new target accelerated approach to the implementation of DQNs for reinforcement learning. We demonstrated that our approach is faster than the classical approach of Mnih where the state, action and reward are stored in the replay buffer. Our TAO-DQN algorithm is based on the observation that the state, action and target can be saved to the replay buffer, thereby minimizing the overestimation error.

Our experiments and our mathematical justification confirm our hypothesis. A DQN will converge to a better policy in less time when the number of max() operators is minimized.

Our future work will involve graph neural networks. We are interested in applying our new TAO-DQN algorithm to NP-hard graph based optimization problems that lie at the

intersection of reinforcement learning and combinatorial optimization. The literature is already suggesting that these NP-hard problems can be solved with reinforcement learning. We plan to show we can solve those problems in less time with fewer resources.

REFERENCES

1. G. N. Yannakakis and J. Togelius, ArtiFIcial //gameaibook.org. Intelligence and Games.Springer, 2018. http://gameaibook.org.
2. Z. Gao, Y. Gao, Y. Hu, Z. Jiang, and J. Su, "Application of Deep Q-Network in Portfolio Management," arXiv:2003.06365 [cs, q-FIn, stat], Mar. 2020.
3. D. Shi, J. Ding, S. M. Errapotu, H. Yue, W. Xu, X. Zhou, and M. Pan, "Deep q-network based route scheduling for transportation network company vehicles," in 2018 IEEE Global Communications Conference (GLOBECOM), pp. 1–7, 2018.
4. T. Ao, J. Shen, and X. Liu, "The Application of DQN in Thermal Process Control," in 2019 Chinese Control Conference (CCC), pp. 2840–2845, 2019.
5. T. Zhang and H. Mo, "Reinforcement learning for robot research: A comprehensive review and open issues," International Journal of Advanced Robotic Systems, vol. 18, no. 3, p. 17298814211007305, 2021.
6. M. Lopez-Martin, B. Carro, and A. Sanchez-Esguevillas, "Application of deep reinforcement learning to intrusion detection for supervised problems," Expert Systems with Applications, vol. 141, p. 112963, 09 2019.
7. A. Stohy, H.-T. Abdelhakam, S. Ali, M. Elhenawy, A. A. Hassan, M. Masoud, S. Glaser, and A. Rako- tonirainy, "Hybrid Pointer Networks for Traveling Salesman Problems Optimization," PLOS ONE, vol. 16, p. e0260995, Dec. 2021.
8. R. S. Sutton and A. G. Barto, Reinforcement learning: an introduction. Adaptive computation and machine learning series, Cambridge, Massachusetts: The MIT Press, second edition ed., 2018.
9. H. Zhang and T. Yu, "Taxonomy of Reinforcement Learning Algorithms," in Deep Reinforcement Learning: Fundamentals, Research and Applications (H. Dong, Z. Ding, and S. Zhang, eds.), pp. 125–133, Singapore: Springer, 2020.
10. C. J. C. H. Watkins and P. Dayan, "Q-learning," Machine Learning, vol. 8, pp. 279–292, May 1992.
11. R. S. Sutton, "Learning to predict by the methods of temporal differences," Mach. Learn., vol. 3, p. 9–44, aug 1988.
12. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," arXiv:1312.5602 [cs], Dec. 2013.
13. 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.
14. H. van Hasselt, "Double Q-learning," in Proceedings of the 23rd International Conference on Neural Information Processing Systems - Volume 2, NIPS'10, (Red Hook, NY, USA), p. 2613–2621, Curran Associates Inc., 2010.
15. H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," CoRR, vol. abs/1509.06461, 2015.
16. V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning," arXiv:1602.01783 [cs], June 2016. arXiv: 1602.01783.
17. 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.
18. W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting Fundamentals of Experience Replay," arXiv:2007.06700 [cs, stat], July 2020.
19. R. Liu and J. Zou, "The Effects of Memory Replay in Reinforcement Learning," in 2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton), (Monticello, IL, USA), pp. 478–485, IEEE, Oct. 2018.
20. S. Zhang and R. S. Sutton, "A Deeper Look at Experience Replay," arXiv:1712.01275 [cs], Apr.

2018.
21. T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” arXiv:1511.05952 [cs], Feb. 2016.
22. S. J. Bradtke, “Reinforcement learning applied to linear quadratic regulation,” in Proceedings of the 5th International Conference on Neural Information Processing Systems, NIPS’92, (San Francisco, CA, USA), p. 295–302, Morgan Kaufmann Publishers Inc., 1992.
23. S. Thrun and A. Schwartz, “Issues in using function approximation for reinforcement learning,” in Proceedings of the 1993 Connectionist Models Summer School (M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, eds.), pp. 255–263, Lawrence Erlbaum, 1993.
24. P. Goyal, P. Doll´ar, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour,” arXiv:1706.02677 [cs], Apr. 2018.
25 .E. Hoffer, I. Hubara, and D. Soudry, “Train longer, generalize better: closing the generalization gap in large batch training of neural networks,” arXiv:1705.08741 [cs, stat], Jan. 2018.
26. S. L. Smith, P. Kindermans, and Q. V. Le, “Don’t decay the learning rate, increase the batch size,” CoRR, vol. abs/1711.00489, 2017.
27. O. Anschel, N. Baram, and N. Shimkin, “Averaged-DQN: Variance Reduction and Stabilization for Deep Reinforcement Learning,” arXiv:1611.01929 [cs, stat], Mar. 2017

AUTHORS

Bob Zigon is a Principal Research Engineer within the Global Research Organization of Beckman Coulter. Bob earned his Bachelor degrees in Computer Science and Applied Mathematics from Purdue University in 1983. From 2013 to 2015 he pursued and earned his Masters degree in Computer Science from Purdue University. He is currently pursuing his PhD in Computer Science from Purdue University. His research interests include machine learning, high performance computing, parallel algorithms and numerical linear algebra.

Fengguang Song is an Associate Professor in the Department of Computer Science at the Indiana University–Purdue University Indianapolis (IUPUI). He earned his Ph.D. in computer Science from the University of Tennessee at Knoxville in 2009. After receiving his Ph.D., he worked as a Post-doctoral Research Associate in the Innovative Computing Laboratory (ICL) between 2010 and 2012, and worked as a Senior Research Scientisttill 2013 in Samsung Research America–Silicon Valley. Since 2013, Dr. Song has beenworking as a professor of computer science at IUPUI. His research interests include high performance computing, advanced parallel algorithms, parallel and distributed systems, and automated performance analysis and optimization.