ANTICIPATORY ASYNCHRONOUS ADVANTAGE ACTOR-CRITIC (A4C): THE POWER OF ANTICIPATION IN DEEP REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

We propose to extend existing deep reinforcement learning (Deep RL) algorithms by allowing them to additionally choose sequences of actions as a part of their policy. This modification forces the network to anticipate the reward of action sequences, which, as we show, improves the exploration leading to better convergence. Our proposal is simple, flexible, and can be easily incorporated into any Deep RL framework. We show the power of our scheme by consistently outperforming the state-of-the-art GA3C algorithm on several popular Atari Games.

1 INTRODUCTION

Basic reinforcement learning has an environment and an agent. The agent interacts with the environment by taking some actions and observing some states and rewards. At each time step t, the agent observes a state s_t and performs an action a_t based on a policy $\pi(a_t|s_t;\theta)$. In return to the action, the environment provides a reward r_t and the next state s_{t+1}. This process goes on until the agent reaches a terminal state. The learning goal is to find a policy that gives the best overall reward. The main challenges here are that the agent does not have information about the reward and the next state until the action is performed. Also, a certain action may yield low instant reward, but it may pave the way for a good reward in the future.

Deep Reinforcement Learning (Mnih et al., 2016)has taken the success of deep supervised learning a step further. Prior work on reinforcement learning suffered from myopic handcrafted designs. The introduction of Deep Q-Learning Networks (DQN) was the major advancement in showing that Deep Neural Networks (DNNs) can approximate value and policy functions. By storing the agent's data in an experience replay memory, the data can be batched (Riedmiller; Schulman et al., 2015) or randomly sampled (Mnih et al., 2013; 2015; Van Hasselt et al., 2016) from different time-steps and learning the deep network becomes a standard supervised learning task with several input-output pairs to train the parameters. As a consequence, several video games could be played by directly observing raw image pixels (Bellemare et al., 2016) and demonstrating super-human performance on the ancient board game Go (Silver et al., 2016).

In order to solve the problem of heavy computational requirements in training DQN, several follow-ups have emerged leading to useful changes in training formulations and DNN architectures. Methods that increase parallelism while decreasing the computational cost and memory footprint were also proposed (Nair et al., 2015; Mnih et al., 2016), which showed impressive performance.

A breakthrough was shown in (Mnih et al., 2016), where the authors propose a novel lightweight and parallel method called Asynchronous Advantage Actor-Critic (A3C). A3C achieves the state-of-the-art results on many gaming tasks. When the proper learning rate is used, A3C learns to play an Atari game from raw screen inputs more quickly and efficiently than previous methods. In a remarkable followup to A3C, (Babaeizadeh et al., 2016) proposed a careful implementation of A3C on GPUs(called GA3C) and showed the A3C can accelerated significantly over GPUs, leading to the best publicly available Deep RL implementation, known till date.

Slow Progress with Deep RL: However, even for very simple Atari games, existing methods take several hours to reach good performance. There is still a major fundamental barrier in the current Deep RL algorithms, which is slow progress due to poor exploration. During the early phases, when the network is just initialized, the policy is nearly random. Thus, the initial experience are primarily several random sequences of actions with very low rewards. Once, we observe sequences which gives high rewards, the network starts to observe actions and associate them with positive rewards and starts learning. Unfortunately, finding a good sequence via network exploration can take a significantly long time, especially when the network is far from convergence and the taken actions are near random. The problem becomes more severe if there are only very rare sequence of actions which gives high rewards, while most others give on low or zero rewards. The exploration can take a significantly long time to hit on those rare combinations of good moves.

In this work, we show that there is an unusual, and surprising, opportunity of improving the convergence of deep reinforcement learning. In particular, we show that instead of learning to map the reward over a basic action space $\mathcal{A}$ for each state, we should force the network to *anticipate* the rewards over an enlarged action space $\mathcal{A}^+ = \bigcup_{k=1}^{K} \mathcal{A}^k$ which contains sequential actions like $(a_1, a_2, ..., a_k)$. Our proposal is a strict generalization of existing Deep RL framework where we allow to take a premeditated sequence of action at a given state s_t, rather than only taking a single action and re-deciding the next action based on the outcome of the first action and so on. Thus the algorithm can pre-decide on a sequence of actions, instead of just the next best action, if the anticipated reward of the sequence is good enough.

Our experiments shows that by simply making the network anticipate the reward for a sequence of action, instead of just the next best actions, the network shows significantly better convergence behavior consistently. We even outperform the fastest known implementation, the GPU accelerated version of A3C (GA3C). The most exciting part is that that anticipation can be naturally incorporated in any existing implementation, including Deep Q Network and A3C. We simply have to extend the action set to also include extra sequences of actions and calculate rewards with them for training, which is quite straightforward.

2 Background

Methods for reinforcement learning can be classified into three broad classes of solutions: Value-based, Policy-based and Actor-Critic.

2.1 Value-based Methods

The main idea in Value based methods is to define a function called Q-function (Q stands for Quality) which estimates the future reward for a given state-action pair. One popular way to construct and learn a Q-function is called Deep-Q learning (Mnih et al., 2015). The Q-function is iteratively learned by minimizing the following loss function

$$L(\theta) = (r + \gamma max_{a'} Q(s', a'; \theta) - Q(a, s; \theta))^2$$

Here, s is the current state, a is the action, r is the reward earned for action a and s' is next state that we end up. The recursive definition

$$Q(s, a) = r + \gamma max_{a'} Q(s', a'; \theta)$$

comes from the Bellman equation in Dynamic Programming. This is called 1-step Q-Learning as we only perform one action and observe the reward. If we instead observe a sequence of k actions and the states resulting from those actions, we can define the Q function as follows

$$Q(s, a) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{k-1} r_{t+k-1} + \gamma^k max_{a'} Q(s_{t+k}, a'; \theta)$$

2.2 Policy-based Methods

In policy-based model-free methods, a function approximator such as a neural network computes the policy $\pi(a_t|s_t;)$, where θ is the set of parameters of the function. θ is updated by maximizing the cumulative reward as per Bellman Equation given by

$$R[t] = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

One of the popular approaches in policy-based methods is REINFORCE (Williams, 1992). REINFORCE uses the gradient of $\nabla_\theta log\pi(a_t|s_t;\theta)R[t]$ which is an unbiased estimator of $\nabla_\theta E[R_t]$. But the rewards are highly variant, and we would like to discount them with a baseline which suggests whether the current reward is good or not. The baseline is denoted by b_t and the new loss function is $\nabla_\theta log\pi(a_t|s_t;\theta)(R[t]-b_t)$. An intuitive baseline is the mean of all previous rewards. If the current reward is higher than the mean of all previous rewards, then the current action is 'good'. Otherwise, it is 'bad'. That is encapsulated in the loss function directly.

2.3 Actor-Critic Methods

Baseline b_t being independent of current state s_t is not the beneficial because it has no context of the current state. Hence, we would like to redefine it as $b_t(s_t)$. One such popular function is $b_t(s_t) = V^\pi(s_t) = E[R_t|s_t]$. Here, V^π is the Value function. This approach marks the transition from pure Policy-Based Methods to a blend of Policy-based and Value-based methods. Here, the policy function acts as an $actor$ because it is responsible for taking actions and the Value function is called the $critic$ because it evaluates the actions taken by the $actor$. This approach is called the Actor-Critic Framework (Sutton & Barto). We still solve the parameters for policy function but use a Value function to decide on the 'goodness' of a reward.

2.4 Asynchronous Advantage Actor Critic(A3C)

A3C (Mnih et al., 2016) is currently the state-of-the-art algorithm on several popular games. It uses an Asynchronous framework in which multiple agents access a common policy, called central policy, and play simultaneously. They communicate the gradients after atmost t_{max} actions. All the communicated gradients from multiple agents are then used to update the central policy. Once the policy parameters are updated, they are communicated back to all the agents playing. The framework uses a shared neural network which gives 2 outputs, one is the policy distribution, and the other is the Value function. Policy $\pi(a_t|s_t,\theta)$ is the output of softmax(because it is a distribution) and Value function $V(s_t,\theta)$ is the output of a linear layer.

The objective function for policy update of A3C is as follows(note that we maximize the policy objective)

$$L_\pi(\theta) = log\pi(a_t|s_t;\theta)(R_t - V(s_t;\theta)) + \beta H(\pi(s_t;\theta))$$

Here, the first part is typical actor-critic framework except that the Value function now shares parameters θ. The second part is the entropy over the policy distribution of actions. From information theory, we know that entropy is maximum when all actions are equally likely. Hence, this term favors exploration of new actions by enforcing some probability to unlikely actions. The weight β decides how much priority we give to exploration. Please note that A3C pseudocode in the original paper doesn't mention anything about entropy, but we include it here as it is discussed in various other references. Since, $V(s_t;\theta)$ is also a function of θ, we also get value-function-gradients from V by minimizing the DQN-type loss function

$$f_v(\theta) = (R_t - V(s_t;\theta))^2$$

Both the gradients are calculated and stored by each agent until they terminate or perform t_{max} actions. The collection of gradients is then communicated, and the updated central network is now available for all agents.

The major concern with A3C is that it relies on sequential training. More generally, all Reinforcement Learning paradigms are plagued by the fact that we do not have a pre-decided training and testing data and we have to leverage information while training. That renders GPUs and other parallelizations useless for implementing RL algorithms, particularly A3C.

2.5 GPU Enabled A3C(GA3C)

GA3C (Babaeizadeh et al., 2016) was proposed as a follow-up and an alternative framework for A3C that enables the usage of GPU. The broad idea of GA3C is to use larger batches of input-output(output in our case refers to reward) pairs to facilitate better usage of GPUs like usual supervised learning. Since we need to perform actions and observe rewards, every agent GA3C maintains two queues called $PredictionQueue$ and $TrainingQueue$. Every Agent queues up Policy requests in $PredictionQueue$ and submits a batch of input-reward pairs to the $TrainingQueue$.

Instead of having a central policy that every agent uses to predict, GA3C has a predictor that takes $PredictionQueue$s from as many agents as possible and sends an inference query to GPU(this is where the batch size increases thereby making use of GPU). Predictor then sends updated policy to all agents that sent their $PredictionQueues$. On the other hand, there's a trainer component of GA3C which takes the input-reward batches from as many agents as possible and updates model parameters by sending the batches to a GPU.

GA3C presents new challenges as it has to deal with trade-offs like size of data trasfer vs number of data transfers to GPU, number of predictors N_P vs size of prediction batches etc. While we build our idea on GA3C, we set most of these parameters to their defaults.

3 Our Proposal: A4C

Our proposal is an unusually straightforward extension, and a strict generalization of the existing deep reinforcement learning algorithms. At high level, by anticipation we extend the basic action set $\mathcal{A}$ to an enlarged action space $\mathcal{A}^+ = \bigcup_{k=1}^{K} \mathcal{A}^k$, which also includes sequences of actions up to length K. As an illustration, let us say $\mathcal{A} = \{L, R\}$ and we allow 2-step anticipation, therefore our new action space is $\mathcal{A}^+ = \mathcal{A} \cup \mathcal{A}^2 = \{L, R, LL, LR, RL, RR\}$. Each element a^+ belonging to $\mathcal{A}^+$ is called a meta-action, which could be a single basic action or a sequence of actions. Typical deep reinforcement learning algorithms have a DNN to output the estimated Q values or policy distributions according to basic action set $\mathcal{A}$. In our algorithm, we instead let the DNN output values for each meta-action in the enlarged action set $\mathcal{A}^+$. Overall, we are forcing the network to anticipate the "goodness" of meta-actions a little further, and have a better vision of the possibilities earlier in the exploration phase.

3.1 Intuition

From human observations and experiences in both sports and video games, we know the importance of "Combo" actions. Sometimes single actions individually do not have much power, but several of common actions could become very powerful while performed in a sequential order. For example, in the popular game CounterStrike, jump-shoot combo would be a very good action sequence. This kind of observation inspires us to explore the potential of "Combos", i.e. multi-step anticipatory actions in reinforcement learning. Moreover, the advantage of anticipatory actions over the standard ones for improving exploration is analogous to how higher n-grams statistics help in better modeling compared to just unigrams in NLP.

Another subtle advantage of anticipating rewards for sequence of actions is better parameter sharing which is linked with multi-task learning and generalization.

Parameter Sharing: Back in 1997, (Caruana, 1998) showed the advantage of parameter sharing. In particular, it showed that a single representation for several dependent tasks is better for generalization of neural networks than only learning from one task. With the addition of meta-action (or extra actions sequences), we are forcing the network layers to learn a representation which is not only useful in predicting the best actions but also predicts the suitability of meta-actions, which is a related task. A forced multi-task learning is intrinsically happening here. As illustrated in Figure 1, the black box parameters are a shared representation which is simultaneously learned from the gradients of basic actions as well as meta-actions. This additional constraint on the network to predict more observable behaviors regularizes the representation, especially in the early stages.

Anticipatory Deep Q Network: Although our main proposal is A4C which improves the current state-of-the-art A3C algorithm, to illustrate the generality of our idea, we start with a simpler algorithm – Anticipatory Deep Q Network (ADQN). DQN is a value-based algorithm whose network approximates Q values for each action. If we see each gradient update as a training sample sent to the network, DQN generates 1 training sample for each action-reward frame. We believe one frame could provide more information than that. With meta-action, i.e., ADQN algorithm, instead we force the network to output Q values for each meta-action in the enlarged action space. For example, in CartPole game, the basic actions are L, R. In ADQN, we let the output values be over $\mathcal{A}^+ =$

$\{L, R, LL, LR, RL, RR\}$. For an experience sequence $(..., s_i, a_i, r_i, s_{i+1}, a_{i+1}, r_{i+1}, s_{i+2}, ...)$, we will get two updates for state s_i:

$$L_i(\theta_i) = (r_i + \gamma \max_{a' \in \mathcal{A}^+} Q(s_{i+1}, a'|\theta_i) - Q(s_i, a_i|\theta_i))^2$$

$$L_i(\theta_i) = (r_i + \gamma r_{i+1} + \gamma^2 \max_{a' \in \mathcal{A}^+} Q(s_{i+2}, a'|\theta_i) - Q(s_i, a_i|\theta_i))^2$$

In this way, we could abtain two gradient updates for each state. This update improves the intermediate representaion (parameter sharing) aggresively leading to superior convergence. In practice, we could organize them into one single training vector, as illustrated in the Figure 1. This algorithm performs very well on CartPole game (see Section 4.1).

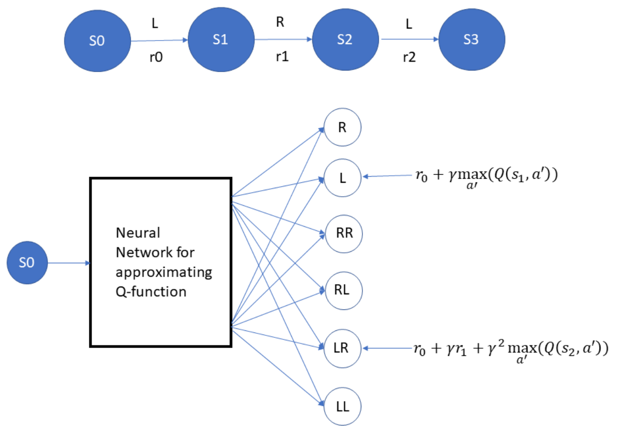
S0
L
r0
S1
R
r1
S2
L
r2
S3
S0
Neural Network for approximating Q-function
R
L
RR
RL
LR
LL
$r_0 + \gamma \max_{a'}(Q(s_1, a'))$
$r_0 + \gamma r_1 + \gamma^2 \max_{a'}(Q(s_2, a'))$

Figure 1: A toy example for ADQN with an enlarged action set $\{L, R, LL, LR, RL, RR\}$. For input s_0, we have 2 gradients, one for action L and other for action LR.

3.2 Anticipatory asynchronous advantage actor-critic (A4C)

In the previous section, we have shown that anticipation can be used on value-based reinforcement methods like DQN. However, DQN is not the state-of-art algorithm, and it converges relatively slowly on more complex tasks like Atari games. Due to the simplicity and generality of our method of anticipation, it is also directly applicable to - Asynchronous Advantage Actor-Critic (A3C) algorithm.

As mentioned earlier, A3C uses a single deep neural network with $|\mathcal{A}|$ policy nodes and 1 value node.To enforce anticipation, we can just enlarge the number of policy nodes in the layer without changing other network architecture. Generally, if we want to support up to K steps of action sequences, we need $|\mathcal{A}^+|$ policy nodes for the output layer, where $\mathcal{A}^+ = \bigcup_{i=1}^{K} \mathcal{A}^k$. The new action space $\mathcal{A}^+$ contains both basic single actions and sequences of actions. This improved algorithm is called Anticipatory asynchronous advantage actor-critic (A4C).

In A4C algorithm, the neural network is used for two parts: prediction and training. In the prediction part, A4C lets the neural network output a distribution of actions from $\mathcal{A}^+$. For each state, we choose a meta-action a^+ according to the output distribution. If a^+ contains only one action, this single action will be executed. If a^+ corresponds to an action sequence $(a_1, a_2, ..., a_k)$, these actions will be executed one by one in order.

A4C is a strict generalization of A3C, and it allows for three kinds of gradient updates for given action-reward frame: dependent updating (DU), independent updating (IU), and switching.

3.2.1 Dependent Updating(DU)

A meta-action a^+ can be viewed as a combination of single actions. On the other hand, several basic actions taken sequentially could be viewed as a meta-action. From here comes our intuition of dependent updating, where each meta-action has its dependent basic actions. When we take a meta-action and get rewards, we not only calculate the gradients for this meta-action, but also for its corresponding basic actions. And for a sequence of basic actions, even if they were not taken as a meta-action, we also update the network as it takes the corresponding meta-action. For example, in a 2-step anticipation setting, we get an experience queue of $(s_0, a_0, r_0, s_1, a_1, r_1, s_2, ...)$. No matter (a_0) was taken as a basic action or (a_0, a_1) was taken as a meta-action, we will update both of them for state s_0. In this case, we get 2 times more gradient updates as A3C for the same amount of episodes, resulting in aggressive updates which lead to accelerated convergence, especially during the initial phases of the learning. We call this kind of dependent updating version of A4C as DU-A4C. Our pseudocode for DU-A4C is presented in Algorithm 1.

Algorithm 1 Anticipatory asynchronous advantage actor-critic with Dependent Updating (DU-A4C) - pseudocode for each actor learner thread

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared $T = 0$
// Assume thread-specific parameter vector $\theta'$ and $\theta'_v$
// Assume a basic set $\mathcal{A} = \{a_i\}$ and the corresponding enlarged action set $\mathcal{A}^+ = \{a_i^+\}$
// where $\mathcal{A}^+ = \bigcup_{k=1}^{K} \mathcal{A}^k$
Initialize thread step counter $t \leftarrow 1$
**repeat**
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta$
    $t_{start} = t$
    Get state $s_t$
    **repeat**
        Choose $a_t^+$ according to policy $\pi(a_t^+|s_t; \theta')$
        **for** $a_i$ in the basic action sequence $(a_1, a_2, ...)$ corresponding to $a_t^+$ **do**
            Perform $a_i$, receive reward $r_t$ and new state $s_{t+1}$
            $t \leftarrow t + 1$
            $T \leftarrow T + 1$
        **end for**
    **until** terminal $s_t$ or $t - t_{start} >= t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$
    **for** $i \in \{t - 1, ..., t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        **for** $j \in \{i, ..., min(i + K, t - 1)\}$ **do**
            Let $a_{ij}^+$ be the meta-action corresponding to the sequence $(a_i, ..., a_j)$
            Accumulate gradients wrt $\theta'$: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_{ij}^+|s_i; \theta')(R - V(s_i; \theta'_v))$
        **end for**
        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 of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
**until** $T > T_{max}$
```

3.2.2 Independent Updating(IU)

Independent update is a very simple and straightforward updating method that we could just view each meta-action a^+ as a separate action offered by the environment. The reward of a^+ is the sum of rewards of taking all the basic actions in a^+ one by one in order. The next state of a^+ is the state after taking all the actions in the sequence. While updating, we only use the information of reward, and the next state of a^+ without regards to the dependencies and relations between meta-actions. The pseudocode is in Algorithm 2 (in Supplementary Materterials).

Clearly, IU leads to less aggressive updates compared to DU. Even though independent updating makes no use of extra information from the intrinsic relations of meta-actions, it still has superior performance in experiments. The reason is that there exist some patterns of actions that yield high rewards consistently and anticipatory action space enables the network to explore this kind of action patterns.

3.2.3 SWITCHING

Our experiment suggests, DU-A4C converges faster over Atari games for the first few hours of training. DU-A4C shows a big gap over the speed of original A3C. However, after training for a longer time, we observe that aggressive updates cause the network to saturate quickly. This phenomenon is analogous to Stochastic Gradient Descent (SGD) updates where initial updates are aggressive but over time we should decay the learning rate (Bottou, 2010).

Technically, dependent updating makes good use of information from the anticipatory actions and yields fast convergence. Independent updating method offers a less aggressive way of updating but it can sustain the growth for longer durations. Thus, we propose a switching method to combine the advantages of these two updating methods.

Switching is simple: we first use dependent updating method to train the network for a while, then switch over to independent updating method from there on. Since the network is the same for both updating methods, the switching process is quite trivial to implement. We notice that this approach consistently stabilizes training on many scenarios(explained in the Section 4). As mentioned, switching is analogous to decaying learning rate with epochs in a typical Neural Network training, the difference being our approach is a hard reduction while learning rate decay is a soft reduction.

The tricky part is when should we switch. Currently, it is more of a heuristic way: for each game, we typically switch half-way. The reason we choose half is that we want to utilize DU to converge quickly in first half and IU to stabilize and continuously increase in the second half. In our experiments, we realize that switching seems to have robust performance in experiments with regards to different choice of switching points.

4 EVALUATIONS

4.1 STUDY OF EXPLORATION USING CARTPOLE GAME

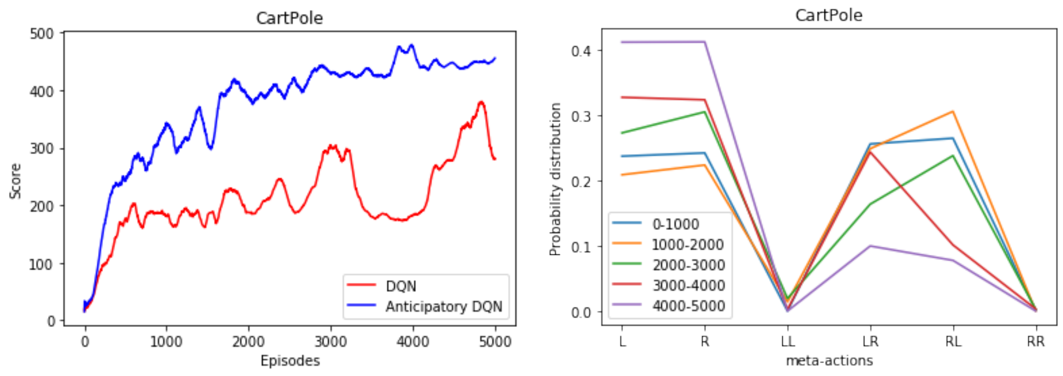

Figure 2: Results and analysis of CartPole game. **Left**: ADQN vs DQN on CartPole-v0; **Right**: The performed action distributions at different training stages. We divide the total 5000 episodes into 5 stages, and plot the distribution at each stage.

To understand the dynamics of the Anticipatory network, we use a simple, classic control game Cartpole. Cartpole game has only 2 basic actions Left and Right, and its state space is $\mathbb{R}^4$. We perform a 2-step Anticipatory DQN(mentioned in section 3.1) with Dependent Updates(DU) and compare against regular DQN. Owing to the simplicity of CartPole, we do not compare A4C vs A3C here, which we reserve for Atari games. We notice a significant jump in the score by using meta-actions space given by $\{L, R, LL, LR, RL, RR\}$, instead of just $\{L,\ R\}$. Although CartPole game

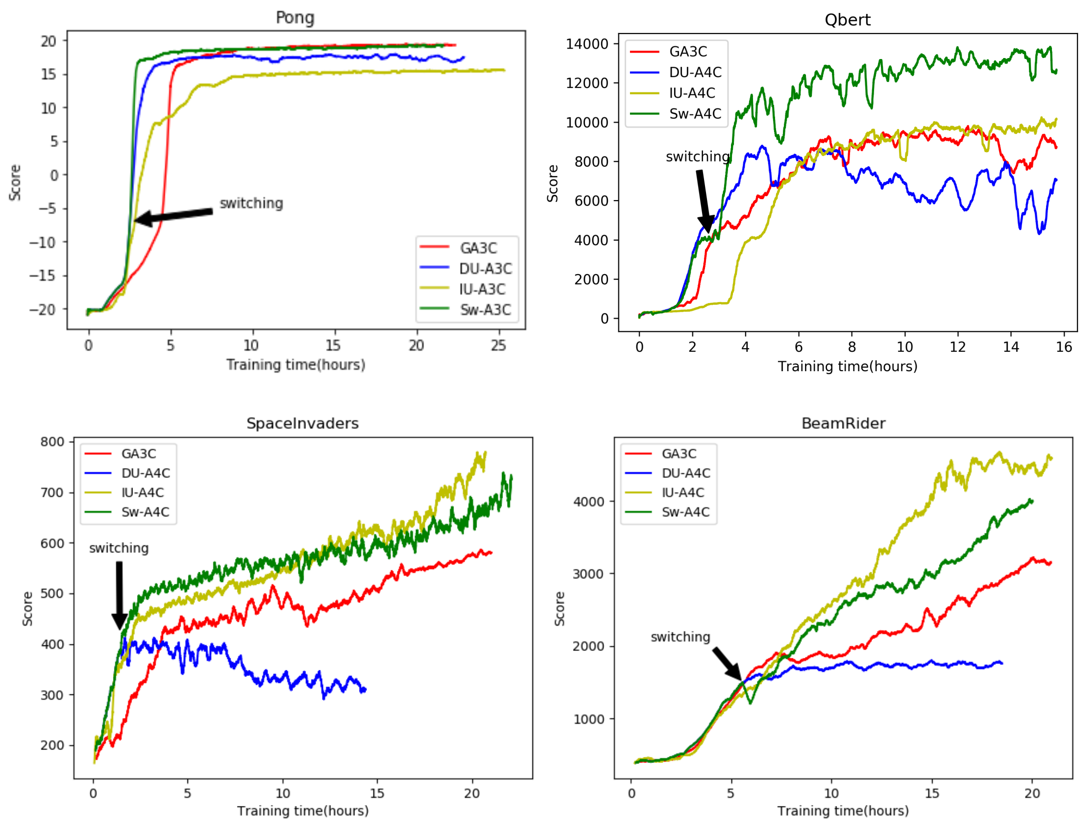

Figure 3: Comparison of three variants of A4C against GA3C. The baseline GA3C is shown in red, Dependent Updates(DU) in blue, Independent Updates(IU) in yellow and the Switching (Sw) in green. Time instant where switching begins is shown by the black arrow. Note that switching time differs with each run and we show a rough indicative switching time.

is simple, the results in the Figure 2(a) reveal the power of anticipation on value-based reinforcement learning.

In the right plot (Figure 2(b)), we also show the probability (frequency) distributions of 6 meta-actions in different learning periods. It is clear from the plots that as learning goes on, the probability of basic actions increases and the probability of multi-step action drops. This trend shows that multi-step actions help the agent to better explore initially, with the anticipated vision of the future, obtaining better rewarding actions. Once the network has seen enough good actions, it figures our the right policy and seems to select basic actions only.

4.2 Atari games

Next, we demonstrate out A4C experiments on 4 popular Atari-2600 games namely Pong, Qbert, BeamRider, and SpaceInvaders. We use the environments provided by OPENAI GYM for both these classes of games. Atari-2600 games are the standard benchmarks for Reinforcement Learning Algorithms. We compare our results against the state-of-the-art GPU based Asynchronous Actor-Critic (GA3C) framework from NVIDIA whose code is publicly available(at `https://github.com/NVlabs/GA3C`)). In order to have uniform playing fields for both A4C and GA3C, we ran the baseline GA3C code on various games on our machine with a single P-100 GPU. We ran each experiment for 3 times on each game and plotted the average scores. To test the robustness of approach, we experimented with various hyperparameter values like minimum training batch size, max norm of gradient (whether to clip gradients; if so MaxNorm=40 by default), learning rate and even the time instant where switching begins. We noticed that our approach is better than baseline on all the settings. Nevertheless, the plots we present are for the optimal setting (MinTrainingBatchSize=40, GradientClipping=False, LearningRate=0.003). These values are also suggested to be optimal in the GA3C code. Note that we compare results using the same setting for all 3 variants of A4C and also for the baseline GA3C.

Figure 3 shows the comparison of three variants of A4C updates against GA3C for four games. Note that the baseline GA3C plots(in red) are very similar to the ones reported in the original paper. We notice that the Independent Updates(IU) performs significantly better than GA3C on all occasions except Qbert, where it is very similar GA3C. In particular, IU achieves a score of 4300 on BeamRider game which is way better than the best result mentioned in GA3C paper. IU crosses 3000 score in just 12.5 hrs while it takes 21 hrs for GA3C to achieve the same score. IU also achieves a score of > 750 on SpaceInvaders game where the best result in GA3C paper achieves < 600. At the same time, the Dependent Updates(DU) method (in blue) starts to rise faster than GA3C but doesn't sustain the growth after sometime owing to reasons mentioned in Section 3.2.3. The only case where DU maintains the growth is Pong. The hybrid switching method(Sw) performs remarkably well consistently on all the games, achieving higher scores than the best of GA3C. For example, on Qbert game, the hybrid Sw method achieves a score of 12000 in just 7 hrs. The best result mentioned in original GA3C paper achieves similar score in 20 hrs. The other re-runs of Qbert in GA3C paper stall at a score of 8000. Sw outperforms GA3C on other games as well, but it is still behind IU on BeamRider and SpaceInvaders games. In all, we notice that Switching from DU to IU after few hours is the most robust method while IU alone is good on 2 games.

5 Conclusion and Future Work

We propose a simple yet effective technique of adding anticipatory actions to the state-of-the-art GA3C method for reinforcement learning and achieve significant improvements in convergence and overall scores on several popular Atari-2600 games. We also identify issues that challenge the sustainability of our approach and propose simple workarounds to leverage most of the information from higher-order action space.

There is scope for even higher order actions. However, the action space grows exponentially with the order of anticipation. Addressing large action space, therefore, remains a pressing concern for future work. We believe human behavior information will help us select the best higher order actions.

References

Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. GA3C: gpu-based A3C for deep reinforcement learning. *CoRR*, abs/1611.06256, 2016.

Marc G Bellemare, Georg Ostrovski, Arthur Guez, Philip S Thomas, and Remi Munos. Increasing the action gap: New operators for reinforcement learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Rich Caruana. Multitask learning. In *Learning to learn*, pp. 95–133. Springer, 1998.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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. *Nature*, 518(7540):529–533, 2015.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pp. 1928–1937, 2016.

Arun Nair, Praveen Srinivasan, Sam Blackwell, Cagdas Alcicek, Rory Fearon, Alessandro De Maria, Vedavyas Panneershelvam, Mustafa Suleyman, Charles Beattie, Stig Petersen, et al. Massively parallel methods for deep reinforcement learning. *arXiv preprint arXiv:1507.04296*, 2015.

Martin Riedmiller. Neural fitted q iteration-first experiences with a data efficient neural reinforcement learning method. In *ECML*, volume 3720, pp. 317–328. Springer.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pp. 1889–1897, 2015.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2016.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Supplementary Materials

Algorithm 2 Anticipatory asynchronous advantage actor-critic with Independent Updating (IU-A4C) - pseudocode for each actor learner thread

```
// Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared $T = 0$
// Assume thread-specific parameter vector $\theta'$ and $\theta'_v$
// Assume a basic set $\mathcal{A} = \{a_i\}$ and the corresponding enlarged action set $\mathcal{A}^+ = \{a_i^+\}$
// where $\mathcal{A}^+ = \bigcup_{k=1}^{K} \mathcal{A}^k$
Initialize thread step counter $t \leftarrow 1$
**repeat**
    Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta$
    $t_{start} = t$
    Get state $s_t$
    **repeat**
        Choose $a_t^+$ according to policy $\pi(a_t^+|s_t;\theta')$
        **for** $a_i$ in the basic action sequence $(a_1, a_2, ..., a_k)$ corresponding to $a_t^+$ **do**
            Perform $a_i$, receive reward $r_{t,i}$ and new state $s_{t,i}$
        **end for**
        $r_t = \sum_{i=1}^{k} r_{t,i}$
        $s_t = s_{t,k}$
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    **until** terminal $s_t$ or $t - t_{start} >= t_{max}$
    $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$
    **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**
    Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$.
**until** $T > T_{max}$
```