RICE UNIVERSITY
Enhancing Exploration in Reinforcement Learning through Multi-Step Actions

By

Tharun Kumar Reddy Medini

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE

Anshumali Shrivastava
Assistant Professor of Computer Science

Richard Baraniuk
J.S. Abercrombie Professor of Electrical and Computer Engineering

Ankit Patel
Assistant Professor of Electrical and Computer Engineering

HOUSTON, TEXAS
October 2020
ABSTRACT

Enhancing Exploration in Reinforcement Learning through Multi-Step Actions

by

Tharun Kumar Reddy Medini

The paradigm of Reinforcement Learning (RL) has been plagued by slow and uncertain training owing to the poor exploration in existing techniques. This can be mainly attributed to the lack of training data beforehand. Further, querying a neural network after every step is a wasteful process as some states are conducive to multi-step actions. Since we train with data generated on-the-fly, it is hard to pre-identify certain action sequences that consistently yield great rewards. Prior research in RL has been focused on designing algorithms that can train multiple agents in parallel and accumulate information from these agents to train faster. Concurrently, research has also been done to dynamically identify action sequences that are suited for a specific input state. In this work, we provide insights into the necessity and training methods for RL with multi-step action sequences in conjunction with the main actions of an RL environment. We broadly discuss two approaches. First of them is A4C - Anticipatory Asynchronous Advantage Actor-Critic, a method that squeezes twice the gradients from the same number of episodes and thereby achieves higher scores and converges faster. The second one is an alternative to Imitation Learning that mitigates the need for having state-action pairs of expert. With as few as 20 action trajectories of expert, we can identify the most frequent action pairs and append to the novice’s action space. We show the power of our approaches by
consistently and significantly outperforming the state-of-the-art GPU-enabled-A3C (GA3C) on popular ATARI games.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
</tr>
<tr>
<td>List of Illustrations</td>
</tr>
<tr>
<td>List of Tables</td>
</tr>
</tbody>
</table>

### 1 A4C: Anticipatory Asynchronous Advantage Actor-Critic

1.1 Introduction ................................................. 1

1.2 Slow Progress with Deep RL .................................. 2

1.3 Options Framework ........................................... 3

1.4 Background .................................................... 5

1.4.1 Value-based Methods ....................................... 5

1.4.2 Policy-based Methods ...................................... 6

1.4.3 Actor-Critic Methods ...................................... 7

1.4.4 Asynchronous Advantage Actor Critic (A3C) .............. 7

1.4.5 GPU Enabled A3C (GA3C) .................................. 8

1.4.6 Options Framework ........................................ 9

1.5 Our First Proposal: A4C ..................................... 10

1.5.1 Intuition .................................................... 11

1.5.2 Anticipatory asynchronous advantage actor-critic (A4C) ... 14

1.6 Evaluations ..................................................... 17

1.6.1 Study of Exploration using Cartpole Game .................. 17

1.6.2 Atari games ................................................. 19

1.7 Conclusion of A4C ............................................. 21

### 2 Efficient Reinforcement Learning from Expert Action Logs 25
2.1 Introduction ....................................................... 25
2.2 Background ...................................................... 28
  2.2.1 Preliminaries ............................................... 28
  2.2.2 Previous Work .............................................. 28
2.3 Our Proposal .................................................... 30
  2.3.1 Framework .................................................. 30
  2.3.2 Motivation .................................................. 32
  2.3.3 Theoretical Intuition ...................................... 33
2.4 Experiments ..................................................... 34
  2.4.1 Baselines ................................................... 35
  2.4.2 Statistical Significance of Action Pairs ................. 39
  2.4.3 How Do We Acquire Expert Meta-actions? ............. 39
  2.4.4 Network Architecture .................................... 41
  2.4.5 Results ..................................................... 41
2.5 Visualization through Atlantis ............................... 43
2.6 Conclusion ..................................................... 45

Bibliography .......................................................... 47
Illustrations

1.1 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 \(LR\). .......................... 13

1.2 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. .......................... 18

1.3 Comparison of three variants of A4C against GA3C. The baseline 
GA3C is shown in red, Dependent Updates(DU) in blue, Independent 
Updates(IU) in cyan and the Switching (Sw) in green. The light color 
fill is one standard deviation away on either side of the mean curve. 19

2.1 Example with 2 primitive actions \{L, R\} and 2 meta-actions \{LR, RR\}. 33

2.2 Action pair distribution for Atlantis with \(4^2 = 16\) pairs. After 13 hours 
of training, the distribution of action pairs is stagnant. Hence, we pick 
the best action pairs 0-2,0-2-0, and 3-0. \((x\)-axis is time in hours, 
\(y\)-axis is % action pairs) .......................... 40

2.3 Time-wise comparison of our proposal against GA3C and DAGGER. 
The common legend for all plots is shown in the last game Fishing Derby. 44

2.4 Components of Atlantis .......................... 45

2.5 Typical tendencies for GA3C trained network at the start (on the left) 
and end (on the right) of a game. .......................... 45
2.6 Typical tendencies for our approach with frequent action pairs at the beginning (on the left) and end (on the right) of a game.
2.1 Game information. Column 2 is the number of basic actions. Column 4 is the percentage of top action pairs (cf. $\alpha$ in Theoretical Intuition section). We modify the notation for action pairs with a dash for clarity.
Chapter 1

A4C: Anticipatory Asynchronous Advantage Actor-Critic

1.1 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.

The introduction of Deep Q-Learning Networks (DQN) [1] was the major advancement in showing that Deep Neural Networks (DNNs) can approximate value and policy functions. Prior work on reinforcement learning suffered from myopic handcrafted designs. By storing the agent’s data in an experience replay memory, DQN showed that the data can be batched [2, 3] or randomly sampled [1, 4, 5] from different time-steps. Consequently, 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 [6] and demonstrating super-human performance on the ancient board game Go [7].

In order to solve the problem of heavy computational requirements in training DQN, several follow-ups have emerged. These increase parallelism while decreasing the computational cost and memory footprint [8, 9]. A breakthrough was shown in [9], where the authors propose a novel lightweight and parallel method called Asynchronous Advantage Actor-Critic (A3C). A3C maintains a common policy for multiple players playing simultaneously and periodically updates the parameters from all the players. A3C achieves the state-of-the-art results on many gaming tasks quickly and efficiently compared to previous methods. In a remarkable followup to A3C, [10] proposed a careful implementation of A3C on GPUs (called GA3C) and showed the A3C can be accelerated significantly over GPUs, leading to the best publicly available Deep RL implementation, known till date.

1.2 Slow Progress with Deep RL

However, even for very simple Atari games, existing methods take several hours to reach good performance. Slow progress due to poor exploration is still a major fundamental barrier in the current Deep RL algorithms. During the early phases, when the network is just initialized, the policy is nearly random. Thus, the initial experiences are primarily several random sequences of actions with very low rewards. Once, we observe sequences which give 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 sequences of
actions which give high rewards, while most others give low or zero rewards. The exploration can take a significantly long time to witness those rare combinations of good moves.

1.3 Options Framework

Finding a good sequence needs a policy that can have some sort of ‘memory’, i.e., it can remember its previous actions and make decisions for longer terms with minimal state information. The most popular concept to solve this is ‘Hierarchical Control’ which has multiple levels of states like a tree and the policy chooses a path of actions. The foremost framework in Hierarchical Control is the ‘Options Framework’ [11]. Options framework maintains multiple policies with each policy suited for only a subset of the possible states. It also has a parameter called ‘termination probability’ which tells whether to switch to another option from the current one. This is analogous to divide-and-conquer algorithms where all the states are divided into options and the model learns policies specific to each options. One major drawback of this framework is that it needs human intervention to design options. A recent work ‘Discovery of Deep Options’ (DDO) [12] mitigates this problem by a unique proposal called ‘Expectation Gradient’ (EG) similar to Expectation Maximization (EM) algorithm. However, EG, like EM algorithm is computationally hard and not suitable for large scale optimization, especially in the case that we need GPUs to accelerate our training of deep neural networks. DDO uses Ray, a High Performance Computing framework developed by RISE lab intended for large scale machine learning.

**Beating GA3C on Time is Not Easy:** A3C is a parallel gradient descent based algorithm and the GPU version GA3C was developed and extensively tested by researchers at NVIDIA. It is the most optimized algorithm for Reinforcement Learning
particularly on Atari games, specially tailored for HPC platforms such as GPUs. Outperforming GA3C in running time is hard as it will require both algorithmic and systems advancement. Options Framework, particularly DDO incorporate sophisticated and expensive updates which make the running time slower. Parallelizing algorithms like ‘Expectation Gradient’ is an orthogonal field of research and it cannot be easily integrated to A3C in a fashion that can fully exploit GPUs. We would also like to stress that most of the prior work compare their results against A3C episode wise. Episode-wise comparison is not a valid comparison most of the time. A reasonable gain in rewards at a considerable cost of time is believable and possible. However, such expensive benefit can give a false impression of episode wise improvements, which can be poor from a time perspective. Thus, given the same hardware, running time to achieve a given reward is a better measure of superiority.

Considering the above factors, we present a simple opportunity of improving the convergence of deep reinforcement learning and in fact beating GA3C in terms of speed. 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 $(a_1, a_2, ..., a_k)$. While using the same episode information like A3C, we extract more information by training with actions pairs apart from the basic actions.

Our proposal is a strict generalization of existing Deep RL framework where we 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. The most exciting part is that this method can be naturally incorporated in any existing implementation, including Deep Q Network and A3C. We simply have to extend the action set to include extra sequences of actions and calculate rewards with
them for training.

As a result, we can directly use the optimized GA3C implementations. As noted before, since we are taking all the advantages of a highly optimized implementation there are hopes of beating GA3C on running time over the same platform. In fact, our experiments precisely demonstrate this point. We do not believe that currently there is an algorithm and the corresponding implementation which is faster than the GA3C algorithm on the same platform.

1.4 Background

In this section, we will summarize most of the important concepts in Reinforcement Learning including the state-of-the-art A3C algorithm. Although vastly different in concept, we’ll also discuss the basic algorithm of Options Framework as it serves a similar purpose to our proposal. The various methods for reinforcement learning can be classified into three broad classes of solutions: Value-based, Policy-based and Actor-Critic.

1.4.1 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 \[4\]. 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

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

### 1.4.2 Policy-based Methods

In policy-based model-free methods, a function approximator such as a neural network computes the policy $\pi(a_t|s_t; \theta)$, where $\theta$ is the set of parameters of the function. $\theta$ 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 [13]. 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.
1.4.3 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, $V^\pi$ 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 [14]. We still solve the parameters for policy function but use a Value function to decide on the ‘goodness’ of a reward.

1.4.4 Asynchronous Advantage Actor Critic (A3C)

A3C [9] 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 $\theta$. The second part is the entropy over the policy distribution of actions. From information theory, we know that entropy of a random variable is maximum when all of its possible values are equally likely. Here actions are the equivalent of the possible values of a Random Variable. Hence, the entropy term favors exploration of new actions by enforcing some probability to unlikely actions. The weight $\beta$ 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 $\theta$, 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.

1.4.5 GPU Enabled A3C(GA3C)

GA3C [10] 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 _PredictionQueues_ 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 transfer 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.

### 1.4.6 Options Framework

As mentioned earlier, options framework is a divide-and-conquer algorithm that divides a task into several sub-tasks and develops individual policies for each sub-task. Technically, an Options Framework has two major parameters:

- **Meta Control Policy** - $\eta(h|s)$ which tells what option $h$ to choose for a given state $s$.
- **Option parameters** - $\pi_h(a|s), \phi_h(s)$ which gives the policy $\pi_h$ specific to the option $h$ and also the termination probability $\phi_h$ which says whether the current option has to end based on the current state $s$.

Actions and parameters may be shared among various options. Each options
can be re-used multiple times. The model starts with a state $s_0$ and chooses option $h_0 = \text{argmax}_h \eta(h|s_0)$. At each iteration $t$, the model follows the following steps:

- Perform an action $a_t$ as per policy $\pi_{h_t}(\cdot|s_t)$
- Get the next state $s_{t+1}$
- Check whether to terminate the option by choosing a random bernoulli with probability $\phi_{h_t}(s_{t+1})$
- If the number is zero, continues to next iteration with the same option, i.e., $h_{t+1} = h_t$.
- If not, choose a new option using meta-policy $h_{t+1} = \eta(\cdot/s_{t+1})$ and proceed to the next iteration.

While Options seems to be a compelling framework, it needs manual design of options which is not desirable. On games with large number of actions, even humans cannot decisively design proper sub-tasks and assign subsets of states to an option. As mentioned earlier in 1.3, most of the methods in Options Framework are time consuming per episode/iteration and cannot be easily scaled to match or beat likes of A3C on time. Also, Deep Neural Networks don’t seem to work well with hierarchical outputs as they work with flat outputs. Designing loss functions for switching options is also a challenging task.

1.5 Our First Proposal: A4C

At a high level, our proposal extends the basic action set $A$ to an enlarged action space $A^+ = \bigcup_{k=1}^{K} A^k$, which also includes sequences of actions up to length $K$. As an illustration, let us say $A = \{L, R\}$ and we allow 2-step anticipation, therefore our new action space is $A^+ = A \cup A^2 = \{L, R, LL, LR, RL, RR\}$. Each element $a^+$ belonging to $A^+$ 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.

### 1.5.1 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, [15] 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 1.1,
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, we illustrate the generality of our idea by applying it to DQN, as shown in Figure 1.1. 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 $A^+ = \{L, R, LL, LR, RL, RR\}$. 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 $s_i$:

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

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

In this way, we could obtain two gradient updates for each state. This update improves the intermediate representation (parameter sharing) aggressively leading to superior convergence. In practice, we could organize them into one single training vector, as illustrated in the Figure 1.1. This algorithm performs very well on CartPole game (see Section 1.6.1).

**Formalism to Enhanced Exploration:** Reinforcement Learning works fundamentally under the assumption that there exists an optimal policy that predicts the
Figure 1.1: 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 \(LR\).

best action for any input state. Let us denote such an optimal policy by \(p^*\). In the toy example in figure 1.1, let us suppose that trajectory \((S_0 - a_0) - (S_1 - a_1) - (S_2 - a_2) - S_3\) is optimal (given by \(p^*\) where \(a_1 = L\), \(a_2 = R\) and \(a_3 = L\). Although we model only \(p^*(a_i/S_i)\), it is not hard to realize that \(p^*(a_{i+1}/S_i, a_i)\) is a non-uniform distribution as an RL environment behaves like a Markov Decision Process (MDP) \([16]\).

We are trying to compare two ways of learning \(p^*\). One is \(p_1\) which has a default action space of \(n\) actions. On the other hand, we have our proposed approach \(p_2\) which has an enlarged action space with \((n + n^2)\) actions. At the beginning of training, both networks are equally random. Hence,

\[
p_1(a_i/S_i) = \frac{1}{n} \quad \text{and} \quad p_2(a_i/S_i) = \frac{1}{n + n^2}
\]
Just as they begin to play and learn from few episodes, both the networks get $\epsilon$ better than random networks at predicting the optimal action for a given state, i.e.,

$$p_1(a_i/S_i) = \frac{1}{n} + \epsilon \text{ and } p_2(a_i-a_{i+1}/S_i) = \frac{1}{n+n^2} + \epsilon$$

Since $p^*$ suggested that $a_{i-1}a_{i+1}$ is the optimal pair for $S_i$, we are interested to see whether

$$p_2(a_{i-1}a_{i+1}/S_i) > p_1(a_{i-1}a_{i+1}/S_i) = p_1(a_i/S_i) * p(a_{i+1}/S_i, a_i)$$

Hence, it boils down to whether the following holds:

$$\frac{1}{n+n^2} + \epsilon > \left(\frac{1}{n} + \epsilon\right)^2 = \frac{1}{n^2} + \frac{2\epsilon}{n} + \epsilon^2$$

For $\epsilon = 0.01$, the above relation holds $\forall n > 5$. Hence, most of the realistic RL environments explore better action sequences when they are included in the action space as opposed to using vanilla action space.

### 1.5.2 Anticipatory asynchronous advantage actor-critic (A4C)

In the previous sections, 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 $\mathcal{A}^+ = \bigcup_{i=1}^{K} \mathcal{A}^i$. 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.

**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.

**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.

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.

**Switching**

Our experiments suggest that DU-A4C converges faster than GA3C on Atari games for the first few hours of training. Particularly, DU-A4C shows a big gap over the speed of original A3C in the Pong game (see Section 1.6.2). 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 [17].

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 duration. 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 straightforward to implement. We notice that this approach consistently stabilizes training on many scenarios (explained in the Section 1.6.2). 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. Technically, we should switch when DU starts to saturate. Confirming the saturation without human observation is difficult as the rewards are extremely variant. In our experiments, we realize that even the mean and median of the rewards of the last 1000 episodes oscillate heavily and we cannot mathematically conclude if the growth has stalled or not. Hence, we set the switching time for each game separately after manually observing the reward curves for DU. Among the games that we report, switching time was 2 hrs for Pong and SpaceInvaders and 2.5 hrs for Qbert and Beamrider. We observe that switching seems to have robust performance in experiments with regards to different choice of hyperparameters (discussed in Section 1.6.2).

1.6 Evaluations

1.6.1 Study of Exploration using Cartpole Game

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
Figure 1.2: 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. The state space is $\mathbb{R}^4$. We perform a 2-step Anticipatory DQN (mentioned in section 1.5.1) 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 is simple, the results in the Figure 1.2(a) reveal the power of anticipation on value-based reinforcement learning.

In the right plot (Figure 1.2(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 out the right policy and seems to select basic actions only.
1.6.2 Atari games

Next, we demonstrate our A4C experiments on 4 popular Atari-2600 games namely Pong, Qbert, BeamRider, and SpaceInvaders. We use the environments provided by OpenAI Gym for these 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). As mentioned

Figure 1.3 : Comparison of three variants of A4C against GA3C. The baseline GA3C is shown in red, Dependent Updates(DU) in blue, Independent Updates(IU) in cyan and the Switching (Sw) in green. The light color fill is one standard deviation away on either side of the mean curve.
earlier, this implementation is made by experts from NVIDIA Lab, which gives the best performance on OpenAI Gym Atari Games up-to-date. We compare our results time-wise on the same games chosen by GA3C paper.

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 14 core CPU and a single Tesla K20 GPU. We ran each experiment for 3 times on each game and plotted the average scores and standard deviation of the scores with respect to time. 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. We use 2-step anticipation in our experiments of A4C. For Atari games with 6 basic actions (like Qbert), 2-step anticipation will result in an enlarged action set with size $6 + 6^2 = 42$, which means the policy output layer of A4C would have 42 nodes. The network architecture of A4C algorithm is the same as GA3C [10] except the output policy layer. It consists of 2 convolutional layers; first layer with $8 \times 8$ filters (16 of them) and the second layer with $4 \times 4$ filters (32 of them). They are followed by a dense layer with 256 nodes. The last layer is the typical softmax layer with as many nodes as the number of effective actions (basic+multi-step). There is a parallel last layer for Value function which is similar to A3C. The input video frames provided by the environment are all $84 \times 84$. We would be making our code public once the review period is over.
Figure 1.3 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 much better than GA3C on SpaceInvaders and BeamRider games (with 6 and 9 basic actions respectively). But it doesn’t do that well on Pong and Qbert games (with 3 and 6 basic actions respectively). In particular, IU achieves a score of 4800 on BeamRider game which is way better than the best result mentioned in GA3C paper. IU crosses 3000 score in just 6 hrs while it takes 13 hrs for GA3C to achieve the same score. At the same time, we notice that 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 1.5.2. The only case where DU maintains the growth for considerable amount of time is Pong (3 basic actions). It is evident that DU is suited for small action spaces and IU is suited for larger action spaces. 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 5 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 also achieves a score of > 700 on SpaceInvaders game where the best result in GA3C paper achieves < 600. In all, we notice that Switching blends the advantages for both DU and IU and is the most robust method for varied scenarios.

1.7 Conclusion of A4C

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 propose a strategy that treats each multi-step action as a sequence of basic actions and extracts gradients for the complete action sequence as well as the preceding sub-sequences. 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 the top priority for the next part of this thesis. We believe human behavior information will help us select the best higher order actions. If we want to include longer action sequences as meta-actions, we could extract frequent "combo actions" from human-playing data, so that the enlarged action set will have limited size even with high-dimensional action combinations. This is precisely our premise for the subsequent work.
Algorithm 1 A4C with Dependent Updating (DU-A4C) - pseudocode for each actor learner thread

1: // Assume global shared parameter vectors $\theta$ and $\theta_v$ and global shared $T = 0$
2: // Assume thread-specific parameter vector $\theta'$ and $\theta'_v$
3: // Assume a basic set $\mathcal{A} = \{a_i\}$ and the corresponding enlarged action set $\mathcal{A}^+ = \{a_i^+\}$
4: // where $\mathcal{A}^+ = \bigcup_{k=1}^K \mathcal{A}^k$
5: Initialize thread step counter $t \leftarrow 1$
6: repeat
7: Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$
8: Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta$
9: $t_{start} = t$
10: Get state $s_t$
11: repeat
12: Choose $a_t^+$ according to policy $\pi(a_t^+|s_t; \theta')$
13: for $a_i$ in the basic action sequence $(a_1, a_2, ...)$ corresponding to $a_t^+$ do
14: Perform $a_i$, receive reward $r_t$ and new state $s_{t+1}$
15: $t \leftarrow t + 1$
16: $T \leftarrow T + 1$
17: end for
18: until terminal $s_t$ or $t - t_{start} >= t_{max}$
19: 
$$R = \begin{cases} 
0 & \text{for terminal } s_t \\
V(s_t, \theta'_v) & \text{for non-terminal } s_t
\end{cases}$$
for $i \in \{t-1, ..., t_{start}\}$ do
\[ R \leftarrow r_i + \gamma R \]
for $j \in \{i, ..., \min(i + K, t - 1)\}$ do
\[ \text{Let } a_{ij}^+ \text{ be the meta-action corresponding to the sequence } (a_i, ..., a_j) \]
\[ \text{Accumulate gradients wrt } \theta': d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_{ij}^+|s_i; \theta')(R - V(s_i; \theta_v')) \]
end for
\[ \text{Accumulate gradients wrt } \theta_v': d\theta_v \leftarrow d\theta_v + \partial(R - V(s_i; \theta_v'))^2 / \partial \theta_v' \]
end for
Perform asynchronous update of $\theta$ using $d\theta$ and of $\theta_v$ using $d\theta_v$. 
until $T > T_{\text{max}}$
Chapter 2

Efficient Reinforcement Learning from Expert Action Logs

2.1 Introduction

In reinforcement learning (RL), an agent makes sequential decisions through interacting with a given environment, via actions, states, and rewards, to achieve a goal. An important problem is to improve the agent’s decision making process (i.e., policy) through experience. In a typical RL setting, an agent must intelligently trade off the focus of the initial learning phase between exploration and exploitation. A passive approach is to inject stochasticity or randomness into the initial process. An alternative, more prudent, approach is to bootstrap or augment the initial learning phase of the agent with demonstrations (i.e., actions) for achieving the goal from an expert agent (e.g., human). Inspired by the biological learning phenomenon observed in infants, children, and even adults, the motivation here is to give a novice agent the ability to mimic the actions of the expert in order to boost exploration efficiency, and in turn, speed up learning.

A key challenge in applying the so-called bootstrapping approach is the need to elicit, and store state and action information from experts. For example, in the problem of training an agent to successfully play Atari games [1], such requirement requires prohibitive memory and careful calibration of the state distribution. If the approach is to apply convolutional neural networks, then the state information
consists of image frames, for which each frame is represented by a matrix of dimension \( \mathbb{R}^{210 \times 160 \times 3} \). Multiplying each frame by the thousands for each game episode, across multiple episodes, quickly leads to a memory blow up. Furthermore, careful and very specific feature engineering [18] is often required in practice. A domain shift or the change in distribution of input states in the testing scenarios can make the whole effort futile.

In this paper, we argue that a very robust form of information can be gathered by ignoring the state and simply storing the sequence of actions performed by an expert. Consider the example of developing an RL based autonomous driving algorithm. An expert driver’s actions can be easily recorded during a driving session. We refer to such records as expert logs. The driving logs contain rich action sequence information, which can be leveraged in training an RL algorithm. For example, actions in the form of pairs can be signal-turn, release accelerator-brake, etc. Thus, if we tweak the policy to produce a trace with same distribution as the expert logs we can expedite the exploration phase of the RL algorithm, which is otherwise very random initially.

Referring back to playing Atari games, the actions of an expert player can also be easily recorded. For example, actions can be the directional movements of a controller such as left, right, up, or down. Such information is less demanding to curate from an expert and has a significantly lower memory footprint (cf. image frames in video games). Furthermore, because we are not storing any state, the information is robust to shift in data distribution. For example, it does not matter if you are playing Pong on an old low resolution cell phone or on a ultra-high resolution laptop.

A related area of research is imitation learning (IL) [19, 20, 21], where the aim is to provide an agent with demonstrations from an expert and the agent tries to learn an optimal policy by mimicking the expert’s decisions. IL falls into two main
categories: behavioral cloning (BC) \[22\] and reverse imitation learning (RIL) \[23\]. BC is the simplest form of IL. In BC, both state and action pairs are collected from an expert, and are then used to train a policy via supervised learning. In this paper, we consider a different and new paradigm. Namely, we consider the case for which we are only given action sequences (or trajectories) in the form of expert logs and our objective is to leverage the expert logs to improve and accelerate learning for a novice RL agent. Utilizing only state information has been previously proposed \[24\], but is less computationally efficient compared to using only action information and is also highly susceptible to the aforementioned scalability challenges. Our setting is related to RIL in the sense that the action sequences are utilized at the beginning of the learning phase to improve exploration. We want to emphasize and make the distinction that our objective is not to perform association rule mining \[25\], but to leverage short action sequences that are useful to bootstrap the initial learning phase of RL. In addition, rule mining could potentially introduce unwanted expert biases into exploration.

Our contribution is a framework for expanding the action space of a novice agent, and the observation and hypothesis that by incorporating most frequent short action sequences (i.e., meta-actions), demonstrated by an expert (via expert logs), into the action space helps improve learning speed, in terms of runtime, and final performance. In addition, extracting an effective set of action sequences requires only a trivially small number of action trajectories, therefore further reducing the memory footprint. By incorporating short action sequences, we are able to capture added information on the action space distribution, which improves on the independent and identical distribution assumption that is potentially problematic in BC. We comparatively demonstrate through experiments that our approach shows consistent improvement
in scores achieved on popular Atari games. For comparison purpose, we emphasize runtime to contrast with other approaches that evaluate on number of episodes, which can have high runtime per episode.

2.2 Background

2.2.1 Preliminaries

We consider standard Markov decision processes (MDP) represented by the tuple $(\mathcal{S}, \mathcal{A}, P, r, \gamma)$. $\mathcal{S}$ and $\mathcal{A}$ are state and action spaces respectively. $P(s_{t+1}|s_t, a_t) : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition probability function for $s_{t+1}, s_t \in \mathcal{S}, a_t \in \mathcal{A}$, and $t \geq 0$. $r : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function and $\gamma \in (0, 1)$ is the discount factor. A trajectory $\xi$ is a temporal state-action sequence. Namely, $\xi = s_0, a_0, s_1, a_1, s_2, \ldots$ The high-level MDP problem is to learn a policy $\pi : \mathcal{A} \rightarrow \mathcal{S}$ to optimize expected rewards with respect to an optimal policy $\pi^*$, which we also denote as an expert policy.

2.2.2 Previous Work

Dataset Aggregation (DAGGER) [26] is an IL algorithm that uses a blended policy consisting of the expert’s policy $\pi^*$ and the novice agent’s policy $\hat{\pi}_i$. At iteration $i$, the policy update rule is defined as $\pi_i = \beta_i \pi^* + (1 - \beta_i) \hat{\pi}_i$. $T$-step trajectories are sampled using $\pi_i$ and a dataset $D$ consisting of states visited by $\pi_i$ and states given by the expert, $\{(s, \pi^*(s))\}$, are collected and aggregated. In practice, $\beta_i \in [0, 1]$ is initially set to 1 and decays with $i$ to iteratively reduce the dependence on the expert by the novice agent. At each of these states, the expert policy may give different action to what the updated policy $\pi_i$ suggests. The principle is to minimize this difference between the policies of the expert and the novice agent. Finally, a simple supervised
classifier $\hat{\pi}_{i+1}$ is trained on $D$.

Several subsequent works [27, 28] make improvements to DAgGER by proposing innovative loss functions to extract meaningful information from the expert’s data. Recently, building on the work of generative adversarial nets (GAN) [29], generative adversarial imitation learning (GAIL) [30] was proposed. The objective of GAIL is to minimize the difference between the expert and novice distributions of state-action trajectories $\phi_E(s,a)$ and $\phi_\theta(s,a)$ respectively. This is accomplished by training a generator network to obtain state-action trajectories and then training a discriminator network to distinguish these trajectories from the expert’s. This adversarial training optimizes an information theoretic loss function called Jensen-Shannon divergence that enforces high mutual information between distributions of both novice and expert.

While GAIL is model-free and unsupervised, it is difficult to have an asynchronous parallel setup where multiple agents play according to a central policy and periodically update the central policy as in the case of Asynchronous Advantage Actor-Critic (A3C) [9]. This presents a scalability issue and it is not straightforward to apply GAIL to Atari games or even complex domains. Also, training GANs are known to be hard in practice [31].

To reiterate, our work differs from the previously mentioned approaches in the fact that we do not need any information about the state-action correspondence for the expert. We only utilize the action log of an expert and identify the frequent sequences as potentially good options for the novice to explore.
2.3 Our Proposal

2.3.1 Framework

We now present our framework. Let $A$ be the given action space of a MDP. We define $A^k$, for $k > 0$, as a $k$-multiset (or repeated permutation) of $A$. For example, suppose the action space is defined as $A = \{L, R\}$, corresponding to (L)eft and (R)ight. Then, a 2-step ($k = 2$) meta-action set, denoted as $A^2$, is defined as $A^2 = \{LL, LR, RL, RR\}$. We refer to the elements in $A$ as primitive actions and the elements in $A^k$ as $k$-step meta-actions. Note that the primitive actions and meta-actions are independent.

Let $A_D = (a_0, a_1, \ldots)$, where $a_i \in A$, be primitive action sequences observed in an expert log. Given $A^k$, let $A = a_0a_1 \ldots a_k$ such that $A \in A^k$. We define

$$A^k_D = \{A \mid A \in A_D\}.$$ 

Namely, $A^k_D$ is the set of observed meta-actions. Our objective is to define a new action set, $A^+$, as follows.

$$A^+ = A \cup \hat{A}^k \text{ s.t. } \hat{A}^k \subset A^k_D \land |\hat{A}^k| \ll |A^k_D|,$$

where $\hat{A}^k$ is constructed according to the expert’s frequent actions in the same environment. We further build on and illustrate our idea with the following simple example.

Suppose we are given an expert log containing 3 episodes of primitive action sequences. The 3 episodes consist of sequences: $LRRRLLLRL$, $LLRRRLRRLR$, and $LLLRLRRLRR$. Given $k$, we iterate through each episode and compute the frequency counts for each meta-action $a \in A^k$ that appears in the sequence. Using $A^2$ results in the following count histogram: $H_{A^2_D} = \{LL : 3 + 1 + 2 = 6, LR : 2 + 3 + 3 = 8, RL : 2 + 2 + 2 = 6, RR : 2 + 3 + 2 = 7\}$. $\hat{A}^k$ can be constructed by choosing the
top $N$ frequency meta-actions, which are $LR$ and $RR$ for $N = 2$. This results in the expanded action set $A^+ = A \cup \{LR, RR\} = \{L, R, LR, RR\}$. Similarly, for $A^3$, we have $H_{A^3} = \{LLL : 2+0+1 = 3, LLR : 1+1+1 = 3, LRL : 1+0+1 = 2, RLL : 1+0+0 = 1, LRR : 1+2+2 = 5, RLR : 0+2+2 = 4, RRL : 1+2+1 = 4, RRR : 1+1+0 = 2\}$. The result is $\hat{A}^3 = \{LRR, RLR, RRL\}$ for $N = 3$. Note that $N = k$ is coincidental in this example.

As $k$ increases, picking top $N$ action sequences can run into the issue of meta-actions with same counts (e.g., $RLR$ and $RRL$ above). Empirically (see Experiments section), we find setting $k = 2$ to be an appropriate choice for Atari games. Note that we only pick top $N$ action sequences for any $k$ as we want the growth of effective action space $A^+$ to be linear in $kN$. The counts for the expert log action sequences give rise to a distribution, akin to $n$-gram models. Therefore, we ensure that the novice RL agent chooses action sequences closer to the distribution of expert logs.

We can train any popular RL algorithm on $A^+$ for the novice agent. In our case, we chose to use the state-of-the-art GPU enabled Asynchronous Advantage Actor-Critic (GA3C) [32]. GA3C has a similar structure to A3C [9]. In this setup, multiple agents play individual games simultaneously with a common central policy and send gradient updates to a central server. The server periodically applies the gradient updates from all the agents and sends out an updated policy to all the agents. This approach is heavily parallelizable and brings down the training time of RL agents from days to hours.

Our proposed meta-actions are in contrast to the work on options [11, 33]. Options is a hierarchical framework that has multi-level actions, typically in a tree structure, and an agent chooses a path. Our proposal differs from this in the fact that we are not discovering best options, but rather we are giving the agent some information
about potentially good options to explore better.

2.3.2 Motivation

The primary motivation to pursue this approach comes from the fact that a human baby does not try to infer the utility of the demonstrator while trying to explore the world, but the baby tries to replicate frequently performed actions taken by the demonstrator for exploring the world.

Analogously, given a very limited demonstration, trying to learn a mapping between state action pairs is implicitly trying to learn the utility function of the expert. It is too ambitious to learn a reliable mapping by training complex neural network with very few state-action pairs. Instead, during the exploration phase, we trust the observed action sequence rather than our ability to figure out the inner state-action mapping mechanism of the expert.

Consider a novice tennis player who wants to imitate the playing style of an expert player like Roger Federer. The player could identify Federer’s popular mini-moves (action sequences) and try to mimic them as and when possible. A forehand shot whenever possible will involve bending legs, swinging arm, and ending the swing with hands all the way over the other shoulder. By trying to mimic this sequence, humans naturally and reflexively explore faster, understand the importance of each step and later even modify expert behaviour with more experience. In the initial stages, it is pointless to figure out why we need to bend legs or swing your arms all the way long after hitting the ball. Inferring utility comes with enough experience.

Similarly, an expert piano player would learn to inculcate frequently used sequences of notes as fast reflexes rather than meticulously focus on one step at a time. We precisely experiment with this point and show that merely providing short frequent
action sequences (combo-action) taken by expert as an option improves the exploration significantly leading to significantly faster learning.

2.3.3 Theoretical Intuition

Although the dynamics of MDPs parameterized by a deep neural network RL framework are highly complex, to give some theoretical intuition to justify our approach, we analyze the following simple example. Suppose \( A = \{L, R\} \) and let \( LR \) and \( RR \) be frequent meta-action pairs such that \( A^+ = \{L, R, LR, RR\} \) (see Figure 2.1). Assume the expert is approximated by the MDP. Furthermore, assume there exists an optimal policy \( \pi^* \) for which we obtain the optimal trajectory \( \xi^* = s_0, L, s_1, R, s_2, L, s_3 \). Let two policies \( \pi_1 \) and \( \pi_2 \), corresponding to action sets \( A \) and \( A^+ \) respectively, be used to learn \( \pi^* \). At the start of training, we initialize both \( \pi_1 \) and \( \pi_2 \) to have uniform distributions (i.e., random). Then we have the following.

\[
\pi_1(L|s_0) = 1/|A| \quad \text{and} \quad \pi_2(L|s_0) = 1/|A^+|.
\]

Let \( \epsilon_1 \) and \( \epsilon_2 \) be so-called \( \epsilon \) improvements to the policies through learning such that we have

\[
\pi_1(L|s_0) = 1/|A| + \epsilon_1 \quad \text{and} \quad \pi_2(L|s_0) = 1/|A^+| + \epsilon_2,
\]
where $0 < \epsilon_1, \epsilon_2 < 1$ maintains a valid distribution for the respective policies. $\xi^*$ specifies an optimal meta-action $LR$. Assume $\pi_1(LR|s_0)$ can be factored as $\pi_1(L|s_0)\pi_1(R|s_1)$. Then we want to analyze the conditions for which the following holds.

$$\pi_1(L|s_0)\pi_1(R|s_1) < \pi_2(LR|s_0).$$

Let $|A^+| = 2|A|$ and $\alpha = |\hat{A}^k|/|A^k|$. Since $\hat{A}^k$ is constructed using top $N$ actions pairs, we have $1/N^2 < \alpha < 1$. Then, we need to check if the following holds

$$(1/|A| + \epsilon_1)^2 < \alpha/2|A| + \epsilon_2.$$

As $|A|$ increases, $\alpha$ decreases. For a reasonable $|A| = 5$ and $\epsilon_1 = 0.01$ with $\epsilon_2 > \epsilon_1^2$, setting $\alpha > 0.44$ satisfies the condition $\pi_1 < \pi_2$. Similarly, for $|A| = 10$ and $\epsilon_1 = 0.01$, setting $\alpha > 0.24$ satisfies the condition $\pi_1 < \pi_2$. As we see in the Experiments section, the average value of $\alpha$ across several Atari games is approximately 0.3. Hence, we can see that for practical environments, the learnability of our proposed action space is better than the naive action space for any off-the-shelf RL algorithm. Note that this analysis can be carried forward similarly for higher order meta-actions.

### 2.4 Experiments

We validate our approach on 8 Atari 2600 games: Atlantis, Space Invaders, Qbert, Demon Attack, Beam Rider, Time Pilot, Asteroids, and Fishing Derby. The number of basic actions in these games ranges from 4 to 18 (see Table 2.1). Atari 2600 games are standard benchmarks for evaluating RL algorithms [1, 4, 9]. We implemented our method using TensorFlow and we use the environments provided by OpenAI Gym.

---

*We assume the $\epsilon$ improvements are symmetric. Namely, for $(1/|A| + \epsilon_i)(1/|A| + \epsilon_j)$, we have $\epsilon_i = \epsilon_j$.**
2.4.1 Baselines

GA3C: We compare our results time-wise against the state-of-the-art algorithm GA3C from NVIDIA whose code is publicly available at \url{https://github.com/NVlabs/GA3C}. GA3C was carefully developed and curated to make good use of GPU and is the best performing algorithm on Open AI Atari games to the best of our knowledge.

Random subset of action pairs: A natural comparison that arises in the context of our method is with a random subset of action-pairs. We hypothesize that most frequent action pairs have useful information that can help a novice learn faster. To validate this, we compare against a randomly picked $N$ subset of action pairs ($N$ is the number of basic actions, we have a total of $N^2$ action pairs). We append these random pairs to the original action space and train the network from scratch.

All possible action pairs: Another natural comparison is to append all possible $N^2$ action pairs and train from scratch. This baseline helps us understand whether having unnecessary action pairs would corrupt the useful information given by frequent action pairs. Note that this would cause the effective action space to explode exponentially.

DAGGER: Additionally, we also compare against the popular DAGGER algorithm for IL. Pseudocode for our implementation of DAGGER is given in Algorithm 2. It trains a classifier network by obtaining training data from expert’s policy and acts according to a mixed policy. As mentioned earlier, DAGGER has a problem of high-memory usage in the case of Atari games because the input video frame size is $210 \times 160 \times 3$ and we play thousands of episodes each with number of steps ranging from 790 to 7400 (Table 2.1); if we keep appending trajectories indefinitely to the training dataset $D$. Also, training a classifier after every episode with ever increasing data is a slow process. For this reason, we limit size of the dataset $D$ to 20 (i.e., we only store the last 20 trajectories of states that novice takes as per the blended policy and the
Algorithm 2 Algorithm for our variant of DAGGER

1: · Initialize $D \leftarrow \emptyset$

2: · Let $\hat{\pi}_i$ denote the novice’s policy at iteration $i$

3: · Let $\pi^*$ denote the expert’s policy

4: · Let $M$ be the maximum length of $D$

5: // At most $M$ trajectories are stored in the data buffer $D$

6: · Let $K$ be the maximum episode for which expert policy is available $D$

7: // After $K$ steps, residual network trains in usual GA3C style without $\pi^*$

8: · Obtain $\hat{\pi}_1$ by randomly initializing novice’s network weights

9: for each parallel thread (as in A3C) do

10: for $i = 1 : K$ do

11: · Let $\pi_i = \beta_i \pi^* + (1 - \beta_i)\hat{\pi}_i$

12: · Sample $T$-step trajectory by playing using $\pi_i$

13: · Get dataset $D_i = (s, \pi^*(s))$ where $s$ is a state visited in the $T$-step trajectory

14: · Append $D_i$ to $D : D \leftarrow D \cup D_i$

15: if $\text{len}(D) > M$ then

16: $D \leftarrow D[-M:]$

17: end if

18: Train novice’s classifier on supervised data $D$

19: end for

20: continued on next page

respective actions that the expert suggests).

But giving indefinite access to expert’s policy makes it an unfair comparison as the whole premise of our approach is the case where we only have very little information
about the expert. Hence, we chose to limit the number of episodes for which expert policy is available. We show results for different episode limits: 100, 500, and 1000 to identify how many episodes are needed to give a good start that can match our proposal in the longer run. After the limit is reached, access to expert policy is removed and the residual network of the novice continues to train using the usual GA3C algorithm until we complete a total 15 hours training time (explained in the How Do We Acquire Expert Meta-actions? section).
Table 2.1: Game information. Column 2 is the number of basic actions. Column 4 is the percentage of top action pairs (cf. $\alpha$ in Theoretical Intuition section). We modify the notation for action pairs with a dash for clarity.

<table>
<thead>
<tr>
<th>Game</th>
<th>$N$</th>
<th>Top action pairs</th>
<th>% of all pairs</th>
<th>Saturation episode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlantis</td>
<td>4</td>
<td>0-2, 0-0, 2-0, 3-0</td>
<td>31.32%</td>
<td>17000</td>
</tr>
<tr>
<td>Space Invaders</td>
<td>6</td>
<td>4-4, 4-2, 2-4, 2-2, 1-1, 5-5</td>
<td>28.82%</td>
<td>39000</td>
</tr>
<tr>
<td>Qbert</td>
<td>6</td>
<td>3-3, 2-2, 5-5, 0-0, 1-1, 4-4</td>
<td>67%</td>
<td>60000</td>
</tr>
<tr>
<td>Demon Attack</td>
<td>6</td>
<td>4-4, 5-5, 3-3, 1-1, 2-2, 4-2</td>
<td>27.21%</td>
<td>32000</td>
</tr>
<tr>
<td>Beam Rider</td>
<td>9</td>
<td>1-1, 8-8, 8-1, 7-7, 7-1, 1-8, 4-8, 8-4, 7-8</td>
<td>14.78%</td>
<td>21000</td>
</tr>
<tr>
<td>Time Pilot</td>
<td>10</td>
<td>8-8, 0-0, 0-8, 4-8, 8-4, 8-0, 4-4, 0-4, 4-0, 1-8</td>
<td>15.41%</td>
<td>32000</td>
</tr>
<tr>
<td>Asteroids</td>
<td>14</td>
<td>0-1, 1-0, 1-1, 0-0, 13-0, 13-13, 0-13, 1-8, 7-13, 8-1, 8-8, 3-1, 10-10, 1-10</td>
<td>21.67%</td>
<td>63000</td>
</tr>
<tr>
<td>Fishing Derby</td>
<td>18</td>
<td>17-17, 13-13, 13-17, 17-13, 9-9, 17-9, 9-17, 5-5, 17-5, 13-9, 9-13, 13-5, 5-17, 12-12, 5-13, 5-9,12-17, 12-13</td>
<td>33.85%</td>
<td>22000</td>
</tr>
</tbody>
</table>
2.4.2 Statistical Significance of Action Pairs

For a game with \( N \) basic actions, we have a total of \( N^k \) \( k \)-step meta-actions. Using all possible \( k \)-tuples of actions leads to exponentially exploding action space and is infeasible for games with large basic action space. Hence, we choose the top \( N \) meta-actions taken by the expert among the \( N^k \) possibilities. This keeps the effective action space to \( kN \) which grows linearly in \( k \) and \( N \). We limit the size of meta-actions \( k \) to 2 because large action spaces may lead to poor convergence. This can also be corroborated from the fact that the most frequent action pairs contribute to a significantly large proportion of all the observed action pairs among 25 expert episodes (shown in Table 2.1). The same cannot be said for action triplets because as \( k \) grows larger, the total possible number of action \( k \)-tuples becomes very high and the statistical significance of frequent tuples becomes very low. Also, as the effective action space \( kN \) increases, the final layer of neural network becomes large and the learnability reduces. Hence, we limit our analysis to action pairs.

2.4.3 How Do We Acquire Expert Meta-actions?

Our focus is to train a novice agent under the assumption that we have access to human expert action sequences. For our experiments, obtaining human actions is hard because we need to find expert humans that play a variety of Atari games. Hence, we chose to replace a human expert with a pre-trained neural network that was independently trained using GA3C. This network is called the expert network. The question now would be: how do we judge that the expert network has achieved the expert status (i.e., has the expert network learned enough to be called an expert)? In the process of training this network, the distribution of action pairs taken by the agent saturates as the average scores per episode converge. We can then identify
Figure 2.2: Action pair distribution for Atlantis with $4^2 = 16$ pairs. After 13 hours of training, the distribution of action pairs is stagnant. Hence, we pick the best action pairs 0-2, 0-0, 2-0, and 3-0. ($x$-axis is time in hours, $y$-axis is % action pairs).

The onset of such a saturation is hard to quantify and detect automatically. Therefore, we standardize the choice of an *expert phase* while training GA3C (i.e., we accumulate the counts for each action pair ($N^2$ such pairs if a game has $N$ basic actions) taken after 15 hours of training GA3C. To further clarify, we train GA3C for $\sim 15$ hours in total. We then freeze this network and call it the expert network. We then play a few ($\sim 25$) episodes with each episode played until termination. We then obtain the histogram of all action pairs and pick the top-$N$ action pairs. The justification for the 15 hours threshold is shown in Figure 2.2. We notice that the distribution of the action pairs does not change significantly after training for about 14 hours. This suggests the onset of the expert phase. Figure 2.2 only shows the time distribution of meta-actions for Atlantis with 4 actions as plots of other games with larger action spaces would be too cluttered.
Note that the time for such saturation is dependent on the computing infrastructure. In our case, we use a 16 core CPU with 122GB RAM and a single Tesla M60 GPU with 8GB memory. Each game plays different number of episodes in the stipulated time. Hence, we provide the approximate number of episodes after which saturation happens for each of the 8 games in Table 2.1 to help reproduce results on other machines.

Also, the 15 hours time used to train the expert network should not be included in the training time of novice as we are only substituting humans with a trained network. Once we have an expert network, we obtain the frequent meta-actions and train the novice from scratch with enlarged action space. The same expert network is used to as an oracle for DAGGER baselines and also to generate expert state-action pairs for InfoGAIL [34] baseline.

2.4.4 Network Architecture

Our network architecture is similar to the one used by A3C and GA3C algorithms except for the output layer that has twice the number of actions as opposed to the former ones. We use 2 convolutional layers; first with 32 8×8 filters and the second with 64 4×4 filters. They are followed by a dense layer with 256 nodes. The last layer is the typical softmax layer with as many nodes as the effective number of actions (basic+meta). There is a parallel last layer for value function similar to A3C. The input video frames are all reshaped to 84×84.

2.4.5 Results

Figure 2.3 shows the comparison our proposal to GA3C and DAGGER (with varying number of expert episodes) for each of the 8 games mentioned before. Each dark line in the plot is the mean of 5 rounds of training. The standard deviation of the curves is
plotted in mild color to show how variant each algorithm is for every game. The green curves correspond to our proposed method, red ones corresponds to GA3C baseline, magenta ones correspond to the *random subset of action pairs* baseline, brown ones correspond to *all action pairs* baseline and the rest correspond to DAGGER with different limit on access to expert policy. We notice that our method consistently outperforms all other baselines by a huge margin in all games. The closest that GA3C or DAGGER could get to our scores are on the game Fishing Derby.

The improvement is huge in the games Asteroids and Atlantis and very glaring in the rest all games. In particular, we notice that giving the novice information about expert actions filters out unnecessary information and helps attain higher scores. For the all action pairs baseline (shown in brown), the bloated action space may help it achieve higher scores than GA3C (like in the case of Asteroids and Atlantis) but it is consistently worse than using only frequent actions. Similarly, on Atlantis, Asteroids and Time Pilot, a random subset of actions (shown in magenta) surpasses GA3C baseline but is significantly lower than our method. These plots establish the usefulness of *a priori* information for a novice.

As for DAGGER, we also observe that giving access to more expert episodes translates to better overall performance of DAGGER on games like Fishing Derby and Asteroids. In all other games, it’s interesting to note that the DAGGER curves seem to grow better than GA3C in the beginning of the training but do not sustain the same growth after the objective function is changed. Note that our method has multi-step actions which could potentially mean that the agent takes more actions per one query to the policy. Hence, in the given time, we play more episodes than other baselines.

Please refer to the supplemental for episode wise plots and also for a qualitative analysis (with screenshots) of why our approach is significantly better than GA3C
on Atlantis game. The main message is that symmetric environments benefit from multi-step actions.

2.5 Visualization through Atlantis

In this section, we will render the video of playing Atlantis game using both GA3C and our approach. This exercise is to understand what gives our approach an order of magnitude improvement in the rewards and in what scenarios does our approach work.

The game of Atlantis aims to protect the six bases shown in white circles in figure 2.4. It has 3 canons to fire (shown in orange circles). The frequency of the incoming airplanes increases with time.

As shown in figure 2.5, for GA3C, we notice that the tendency is to fire the middle cannon most of the time. Since there is only one possible action for each query, and the attacks happens from either side, the optimal strategy is to fire the central cannon and hope to hit as many airplanes as possible.

On the other hand, for our approach, we notice that the tendency is to fire either the two cross-cannons or fire one cross-cannon in quick succession (shown in figure 2.6). In the beginning, it tends to fire one of the cross-cannons repeatedly which results in fair amount of success. As it learns more, the tendency shifts to fire two side cannons at the same time as the frequency and speed of airplanes increases. This yields in much higher success arte, thereby sustaining the game longer and achieving super high rewards.
Figure 2.3: Time-wise comparison of our proposal against GA3C and DAGGER. The common legend for all plots is shown in the last game Fishing Derby.
2.6 Conclusion

We propose a framework that leverages the demonstrations of an expert via expert logs. The approach incorporates short action sequences (cf. both state and action) into a novice RL agent’s action space, which expedites and improves learning. Our approach is simple, effective, imposes low storage cost, and can scale to high-dimensional states. We show our approach outperforms state-of-the-art algorithms such as GA3C and
Figure 2.6: Typical tendencies for our approach with frequent action pairs at the beginning (on the left) and end (on the right) of a game.

DAGGER by significant margin time-wise on popular Atari 2600 games.
Bibliography


