Learning to Mix $n$-Step Returns: Generalizing $\lambda$-Returns for Deep Reinforcement Learning

Abstract

Reinforcement Learning (RL) can model complex behavior policies for goal-directed sequential decision making tasks. A hallmark of RL algorithms is Temporal Difference (TD) learning: value function for the current state is moved towards a bootstrapped target that is estimated using next state’s value function. $\lambda$-returns generalize beyond 1-step returns and strike a balance between Monte Carlo and TD learning methods. While $\lambda$-returns have been extensively studied in RL, they haven’t been explored a lot in Deep RL. This paper’s first contribution is an exhaustive benchmarking of $\lambda$-returns. Although mathematically tractable, the use of exponentially decaying weighting of $n$-step returns based targets in $\lambda$-returns is a rather ad-hoc design choice. Our second major contribution is that we propose a generalization of $\lambda$-returns called Confidence-based Autodidactic Returns (CAR), wherein the RL agent learns the weighting of the $n$-step returns in an end-to-end manner. This allows the agent to learn to decide how much it wants to weigh the $n$-step returns based targets. In contrast, $\lambda$-returns restrict RL agents to use an exponentially decaying weighting scheme. Autodidactic returns can be used for improving any RL algorithm which uses TD learning. We empirically demonstrate that using sophisticated weighted mixtures of multi-step returns (like CAR and $\lambda$-returns) considerably outperforms the use of $n$-step returns. We perform our experiments on the Asynchronous Advantage Actor Critic (A3C) algorithm in the Atari 2600 domain.

1 Introduction

Reinforcement Learning (RL) (Sutton & Barto [1998]) is often used to solve goal-directed sequential decision making tasks wherein conventional Machine Learning methods such as supervised learning are not suitable. An RL agent is not explicitly told the optimal actions but must instead discover them based on evaluative feedback given in terms of rewards sampled from the environment. Goal-directed sequential decision making tasks are modeled as Markov Decision Processes (MDP) (Puterman [2014]). Traditionally, tabular methods were extensively used for solving MDPs wherein value function or policy estimates were maintained for every state. Such methods become infeasible when the underlying state space of the problem is exponentially large or continuous. Traditional RL methods have also used linear function approximators in conjunction with hand-crafted state spaces for learning policies & value functions. This need for hand-crafted task-specific features has limited the applicability of RL, traditionally. It was necessary to learn hierarchical and abstract state representations which can capture the essential features required for optimal control in a generalizable fashion.
Recent advances in representation learning in the form of deep neural networks provide us with an effective way to achieve such generalization. Deep neural networks can learn hierarchically compositional representations that enable RL algorithms to generalize over large state spaces. The use of deep neural networks in conjunction with RL objectives has shown remarkable results such as learning to solve the Atari 2600 tasks from raw pixels, learning to solve complex simulated physics tasks, and showing super-human performance on the ancient board game of Go.

Building accurate and powerful (in terms of generalization capabilities) state and action value function estimators is important for successful RL solutions. This is because many practical RL solutions (Q-Learning, SARSA, TRPO) use TD Learning (Sutton, 1988). The ability to build better estimates of the value functions directly results in better policy estimates as well as faster learning. \( \lambda \)-returns (LR) (Sutton & Barto, 1998) are very effective in this regard. They are effective for faster propagation of delayed rewards and also result in more reliable learning. LR provide a trade-off between Monte Carlo and TD learning methods. They model the TD target using a mixture of multi-step returns, wherein the weights are exponentially decayed. With the advent of deep RL, the use of multi-step returns has gained a lot of popularity. However, \( \lambda \)-returns have not been explored extensively in the deep RL setting. Having said that, it is to be noted that the use of exponentially decaying weighting for various \( n \)-step returns seems to be an ad-hoc design choice made by LR. In this paper, we start off by extensively benchmarking \( \lambda \)-returns (our experiments only use truncated \( \lambda \)-returns due to the nature of the DRL algorithm that we work with) and also propose a generalization of \( \lambda \)-returns called the Confidence-based Autodidactic Returns (CAR). In CAR, the DRL agent learns in an end-to-end way, the weights to assign to the various \( n \)-step return based targets. Also note that in CAR, the weights assigned to various \( n \)-step returns change based on the current state that the DRL agent is in. In this sense, CAR weights are dynamic and using them represents a significant level of sophistication as compared to the usage of \( \lambda \)-returns. In the Deep Reinforcement Learning (DRL) setting not much has been done in the context of using mixture of multi-step returns to build better value function estimators. An exception to that statement is the investigation done in Harb & Precup (2017). It shows promising results on two Atari 2000 tasks using the backward view of eligibility traces - Pong and Tennis. The contributions of this work are four-fold:

1. We propose an extension to the classical \( \lambda \)-returns algorithm in the DRL setting.
2. We propose a novel generalization of \( \lambda \)-returns, called Confidence-based Autodidactic Returns (CAR).
3. We empirically demonstrate that using sophisticated mixtures of multi-step return methods like \( \lambda \)-returns and Confidence-based autodidactic returns leads to considerable improvement in the performance of a DRL agent. While both \( \lambda \)-returns and CAR can be used with any algorithm which uses TD-learning, all our experiments have been with the A3C.
4. We provide an analysis of how the weights learned by CAR for the \( n \)-step returns vary during training and also within an episode.

## 2 Background

### 2.1 Preliminaries

An MDP (Puterman, 2014) is defined as the tuple \( \langle S, A, \pi, r, \gamma \rangle \), where \( S \) is the set of states in the MDP, \( A \) is the set of actions, \( r : S \times A \mapsto \mathbb{R} \) is the reward function, \( \mathbb{P} : S \times A \times S \mapsto [0,1] \) is the transition probability function such that \( \sum_{a \in A} p(s, a, s') = 1 \), \( p(s, a, s') \geq 0 \), and \( \gamma \in [0,1) \) is the discount factor. We consider a standard RL setting wherein the sequential decision-making task is modeled as an MDP and the agent interacts with an environment \( \mathcal{E} \) over a number of discrete time steps. At a time step \( t \), the agent receives a state \( s_t \) and selects an action \( a_t \) from the set of available actions \( A \). Given a state, the agent could decide to pick its action stochastically. Its policy \( \pi \) is in general a mapping defined by: \( \pi : S \times A \mapsto [0,1] \) such that \( \sum_{a \in A} \pi(s, a) = 1 \), \( \pi(s, a) \geq 0 \) \( \forall s \in S \), \( \forall a \in A \). At any point in the MDP, the goal of the agent is to maximize the return, defined as: \( G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \) which is the cumulative discounted future reward. The state value function of a
policy \( \pi \), \( V^\pi(s) \) is defined as the expected return obtained by starting in state \( s \) and picking actions according to \( \pi \).

2.2 Actor Critic Algorithms

Actor Critic algorithms (Konda & Tsitsiklis, 2000) are a class of approaches that directly parameterize the policy (using an actor) \( \pi_a(a|s) \) and the value function (using a critic) \( V^\pi(s) \). They update the policy parameters using Policy Gradient Theorem (Sutton et al., 1999; Silver et al., 2014) based objective functions. The value function estimates are used to reduce the variance in policy gradient estimates.

2.3 Asynchronous Advantage Actor Critic

Asynchronous Advantage Actor Critic (A3C) (Mnih et al. (2016)) introduced the first class of actor-critic algorithms which worked on high-dimensional complex visual input space. The key insight in this work is that by executing multiple actor learners on different threads in a CPU, the RL agent can explore different parts of the state space simultaneously. This ensures that the updates made to the parameters of the agent are uncorrelated.

The actor can improve its policy by following an unbiased low-variance sample estimate of the gradient of its objective function with respect to its parameters, given by:

\[
\nabla_{\theta_a} \log \pi_{\theta_a}(a_t|s_t) (G_t - V(s_t))
\]

In practice, \( G_t \) is often replaced with a biased lower variance estimate based on multi-step returns. In the A3C algorithm \( n \)-step returns are used as an estimate for the target \( G_t \), where \( n \leq m \) and \( m \) is a hyper-parameter (which controls the level of rolling out of the policies). A3C estimates \( G_t \) as:

\[
G_t \approx \hat{V}(s_t) = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V(s_{t+n})
\]

and hence the objective function for the actor becomes:

\[
L(\theta_a) = \log \pi_{\theta_a}(a_t|s_t) \delta_t
\]

where \( \delta_t = \sum_{i=t}^{t+j-1} \gamma^{i-t} r_i + \gamma^j V(s_{t+1}) - V(s_t) \) is the \( j \)-step returns based TD error.

The critic in A3C models the value function \( V(s) \) and improves its parameters based on sample estimates of the gradient of its loss function, given as: \( \nabla_{\theta_c}(\hat{V}(s_t) - V_{\theta_c}(s_t))^2 \).

2.4 Weighted Returns

Weighted average of \( n \)-step return estimates for different \( n \)'s can be used for arriving at TD-targets as long as the sum of weights assigned to the various \( n \)-step returns is 1 (Sutton & Barto (1998)). In other words, given a weight vector \( w = (w^{(1)}, w^{(2)}, \cdots, w^{(m)}) \), such that \( \sum_{i=1}^{m} w^{(i)} = 1 \), and \( n \)-step returns for \( n \in \{1, 2, \cdots, m\} \): \( G_t^{(1)}, G_t^{(2)}, \cdots, G_t^{(m)} \), we define a weighted return as

\[
G_t^w = \sum_{n=1}^{m} w^{(n)} G_t^{(n)}
\]

(1)

Note that the \( n \)-step return \( G_t^{(n)} \) is defined as:

\[
G_t^{(n)} = \sum_{i=1}^{n} \gamma^{i-1} r_{t+i} + \gamma^n V(s_{t+n})
\]

(2)
2.5 $\lambda$-Returns

A special case of $G_t^n$ is $G_t^1$ (known as $\lambda$-returns) which is defined as:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{m-1} \lambda^{n-1} G_t^{(n)} + \lambda^{m-1} G_t^{(m)}$$  \hspace{1cm} (3)

What we have defined here are a form of truncated $\lambda$-returns for TD-learning. These are the only kind that we experiment with, in our paper. We use truncated $\lambda$-returns because the A3C algorithm is designed in a way which makes it suitable for extension under truncated $\lambda$-returns. We leave the problem of generalizing our work to the full $\lambda$-returns as well as eligibility-traces ($\lambda$-returns are the forward view of eligibility traces) to future work.

3 $\lambda$-Returns and Beyond: Autodidactic Returns

3.1 Autodidactic Returns

Autodidactic returns are a weighted average of the $n$-step returns wherein the weight vector is also learned alongside the value functions which are being approximated. It is this generalization which makes the returns autodidactic. Since the autodidactic returns we propose are constructed using weight vectors that are dynamic (the weights change with the state the agent encounters in the MDP), we denote the weight vector as $w(s_t)$. The autodidactic returns can be used for learning better approximations for the value functions using the TD(0) learning rule based update equation:

$$V_t(s_t) \leftarrow V_t(s_t) + \alpha \left( G_t^{w(s_t)} - V_t(s_t) \right)$$  \hspace{1cm} (4)

In contrast with autodidactic returns, $\lambda$-returns assign weights to the various $n$-steps returns which are constants given a particular $\lambda$. We reiterate that the weights assigned by $\lambda$-returns don’t change during the learning process. Therefore, the autodidactic returns are a generalization and assign weights to returns which are dynamic by construction. The autodidactic weights are learned by the agent, using the reward signal it receives while interacting with the environment.

3.2 Confidence-based Autodidactic Returns

All the $n$-step returns for state $s_t$ are estimates for $V(s_t)$ bootstrapped using the value function of corresponding $n^{th}$ future state ($V(s_{t+n})$). All the value functions are estimates themselves. Hence, a natural way for the RL agent to weigh an $n$-step return ($G_t^{(n)}$) would be to compute this weight using the confidence, $c(s_{t+n})$, that the agent has in the value function estimate, $V(s_{t+n})$, using which the $n$-step return was estimated. The agent weights the $n$-returns based on how confident it is about that $n$-step return being a good estimate for $V(s_t)$. We denote this confidence as $c(s_{t+n})$. Since the $n$-step return in turn depends on $V(s_{t+n})$, $c(s_{t+n})$ also reflects the confidence that the agent has about the value function estimate $V(s_{t+n})$. Next, the weight vector $w(s_t)$ is computed as:

$$w(s_t) = \left( w(s_t)^{(1)}, w(s_t)^{(2)}, \ldots, w(s_t)^{(m)} \right)$$

where $w(s_t)^{(i)}$ is given by:

$$w(s_t)^{(i)} = \frac{e^{c(s_{t+i})}}{\sum_{j=1}^{m} e^{c(s_{t+j})}}$$  \hspace{1cm} (5)

3.3 Using $\lambda$-returns in A3C

While $\lambda$-returns have been well studied in literature [Peng & Williams1996, Sutton & Barto1998, Seijen & Sutton2014], their usage in DRL has been limited. We propose a straightforward way to incorporate (truncated) $\lambda$-returns into the A3C framework. We call this combination as LRA3C.

The critic in A3C using TD(0) algorithm for arriving at good estimates for the value function. However, note that the TD-target can in general be based any $n$-step returns (or a mixture
The A3C algorithm in specific is well suited for using weighted returns such as $\lambda$-returns since the algorithm already uses $n$-step returns for bootstrapping. Using eqs. (1) to (3) makes it very easy to incorporate weighted returns into the A3C framework. The respective sample estimates for the gradients of the actor and the critic become:

$$\nabla_{\theta_a} \log \pi_{\theta_a}(a_t|s_t)(G^\lambda_t - V(s_t))$$

$$\nabla_{\theta_c} (G^\lambda_t - V_{\theta_c}(s_t))^2$$

### 3.4 Using Autodidactic Returns in A3C

We propose to use *autodidactic returns* in place of normal $n$-step returns in the A3C framework. We call this combination as CARA3C. In a generic DRL setup, a forward pass is done through the network to obtain the value function of the current state. The parameters of the network are progressively updated based on the gradient of the loss function and the value function estimation (in general) becomes better. For predicting the confidence values, a distinct neural network is created which shares all but the last layer with the value function estimation network. So, every forward pass of the network on state $s_t$ now outputs the value function $V(s_t)$ and the confidence the network has in its value function prediction, $c(s_t)$. Figure 1 visually demonstrates how the confidence values are calculated using an A3C network. Next, using eqs. (1) to (3) the weighted average of $n$-step returns is calculated and used as a target for improving $V(s_t)$. Algorithm 1 contains detailed pseudo-code for training CARA3C agents is in Appendix F. The policy improvement is carried out by following sample estimates of the loss function’s gradient, given by: $\nabla_{\theta_a} \log \pi_{\theta_a}(a_t|s_t)\delta_t$, where $\delta_t$ is now defined in terms of the TD error term obtained by using *autodidactic returns* as the TD-target. Overall, the sample estimates for the gradient of the actor and the critic loss functions are:

$$\nabla_{\theta_a} \log \pi_{\theta_a}(a_t|s_t)(G^w_t(s_t) - V(s_t))$$

$$\nabla_{\theta_c}(G^w_t(s_t) - V_{\theta_c}(s_t))^2$$

![Figure 1: CAR-A3C Network - Confidence-based weight vector calculation for state $s_1$](image)
3.5 Avoiding pitfalls in TD learning of Critic

The LSTM-A3C neural networks for representing the policy and the value function share all but the last output layer. In specific, the LSTM [Hochreiter & Schmidhuber (1997)] controller which aggregates the observations temporally is shared by the policy and the value networks. As stated in the previous sub-section, we extend the A3C network to predict the confidence values by creating a new output layer which takes as input the LSTM output vector (LSTM outputs are the pre-final layer). Figure 1 contains a demonstration of how \( w(s_t) \) is computed. Since all the three outputs (policy, value function, confidence on value function) share all but the last layer, \( G_w(s_t) \) depends on the parameters of the network which are used for value function prediction. Hence, the autodidactic returns also influence the gradients of the LSTM controller parameters. It has observed that when the TD target, \( G_w(s_t) \) is allowed to move towards the value function prediction \( V(s_t) \), it can make the learning unstable. This happens because the \( L_2 \) loss between the TD-target and the value function prediction can now be minimized by moving the TD-target towards the egorusion value function predictions \( V(s_t) \) instead of the other way round. To avoid this instability we ensure that gradients do not flow back from the confidence values computation’s last layer to the LSTM layer’s outputs. In effect, the gradient of the critic loss with respect to the parameters decided to the autodidactic return computation can no longer influence the gradients of the LSTM parameters (or any of the previous convolutional layers) at all. To summarize, during back-propagation of gradients in the A3C network, the parameters specific to the autodidactic return computation do not contribute at all to the gradient which flows back into the LSTM layer. This ensures that the parameters of the confidence network are learned while treating the LSTM outputs as fixed feature vectors. This entire scheme of not allowing gradients to flow back from the confidence value computation to the LSTM outputs has been demonstrated in Figure 1. The forward arrows depict the parts of the network which are involved in forward propagation whereas the backward arrows depict the path taken by the back-propagation of gradients.

4 Experimental Setup and Results

We performed general game-play experiments with CARA3C and LRA3C on 22 tasks in the Atari domain. All the networks were trained for 100 million time steps. The hyper-parameters for each of the methods were tuned on a subset of four tasks: Seaquest, Space Invaders, Gopher and Breakout. The same hyper-parameters were used for the rest of the tasks. The baseline scores were taken from...
Sharma et al. (2017). All our experiments were repeated thrice with different random seeds to ensure that our results were robust to random initialization. All results reported are the average of results obtained by using these three random seeds. Since the A3C scores were taken from Sharma et al. (2017), we followed their training and testing regime as well. Appendix A contains experimental details about the training and testing regimes. Appendix G documents the procedure we used for picking important hyper-parameters for our methods.

4.1 General gameplay performance

Figure 3: Training curves for raw scores obtained by CARA3C and LRA3C

Figure 2 shows the percentage improvement of our sophisticated mixture of n-step return methods (CARA3C and LRA3C) over A3C. If the scores obtained by one of our methods and A3C in a task are \( p \) and \( q \) respectively, then the percentage improvement is calculated as: \( \left( \frac{p - q}{q} \times 100 \right) \). As we can see, CARA3C achieves a staggering 67× performance in the task Kangaroo. Table 1 (in Appendix B) contains a comparison of the raw scores obtained by our methods to A3C baseline scores. Evolution of the average performance of our methods with training progress has been shown in Figure 3. An expanded version of the graph for all the tasks can be found in Appendix C.

4.2 Analysis of the Evolution of Weights During Training

Figure 4 plots the weights assigned to various n-step (for \( n \in \{1, 2, \ldots, 20\} \)) returns for states \( s_1, s_2, \ldots \) in a trajectory. The weights given to each of the n-step returns are learned and hence change to minimize the critic loss function. Figure 4 demonstrates that the weights assigned to each of the n-step returns start off with being random (all equal to with 0.05) and slowly evolve to different values depending on the corresponding confidence value predicted by the network. An expanded version of the graphs for all the tasks can be found in Appendix D. We observe that the 1-step returns and the 20-step returns are assigned most confidence. This seems to indicate a bias-variance trade-off between the various n-step returns with both extremes being favorable, perhaps in different parts of the state space. In contrast, for \( \lambda \)-returns, since \( \lambda < 1 \), 1-step returns always get the highest weight. We observe a similar trend only in a few tasks, for Confidence-based Autodidactic Returns. However, in some of the tasks, such as Gopher one can observe that the network is much more confident about the 20-step returns (on an average) than the 1-step returns.

4.3 Analysis of the Evolution of Weights During an Episode

Figure 5 demonstrates the evolution of the weights assigned to various n-step returns during the duration of an episode. It can be seen that for many tasks, the weights evolve in a dynamic fashion as the episode goes on. This seems to validate our motivation for using dynamic Autodidactic Returns. An expanded version of the graphs for all the tasks can be found in Appendix E.
Figure 5: Evolution of weights given to each of the n-step returns (where $n \leq 20$) with training time

4.4 Analysis of the Learned Value Function

In this paper, we propose two methods for learning value functions in a more sophisticated manner than using n-step returns. Hence, it is important to analyze the value functions learned by our methods and understand whether our methods are indeed able to learn better value functions than baseline methods or not. For this sub-section we trained A3C agents to serve as baselines. To verify our claims about better learning of value functions, we conducted the following experiment. We took trained CARA3C LRA3C and A3C agents and computed the $L_2$ loss between the value function $V(s_t)$ predicted by a method and the actual discounted sum of returns ($\sum_{k=0}^{T-t} \gamma^k r_{t+k}$). We averaged this quantity over 10 episodes and plotted it as a function of time steps within an episode. Figure 6 demonstrates that our novel method CARA3C learns a much better estimate of the Value function $V(s_t)$ than LRA3C and A3C. The only exception to this is the game of Kangaroo. The reason that A3C and LRA3C critics manage to estimate the value function well in Kangaroo is because the policy is no better than random and in fact their agents often score score of 0 (which is easy to estimate).

Figure 6

5 Conclusion and Future Work

We propose a straightforward way to incorporate $\lambda$-returns into the A3C algorithm and carry out a large-scale benchmarking of the resulting algorithm LRA3C. We go on to propose a natural generalization of $\lambda$-returns called Confidence-based Autodidactic returns (CAR). In CAR, the agent learns to assign weights dynamically to the various n-step returns from which it can bootstrap. Our experiments demonstrate the efficacy of sophisticated mixture of multi-steps returns with at least one of CARA3C or LRA3C out-performing A3C in 18 out of 22 tasks. In 9 of the tasks CARA3C performs the best whereas in 9 of them LRA3C is the best. CAR gives the agent the freedom to learn and decide how much it wants to weigh each of its n-step returns.

The concept of Autodidactic Returns is about the generic idea of giving the DRL agent the ability to model confidence in its own predictions. We demonstrate that this can lead to better TD-targets, in turn leading to improved performances. We have proposed only one way of modeling the autodidactic weights wherein we use the confidence values that are predicted alongside the value function estimates. There are multiple other ways in which these n-step return weights can be modeled. We believe these ways of modeling weighted returns can lead to even better generalization in terms how the agent perceives it’s TD-target. Modeling and bootstrapping off TD-targets is fundamental to RL. We believe that our proposed idea of CAR can be combined with any DRL algorithm (Mnih et al., 2015; Jaderberg et al., 2017; Sharma et al., 2017) wherein the TD-target is modeled in terms of n-step returns.
References

Bellemare, Marc G, Naddaf, Yavar, Veness, Joel, and Bowling, Michael. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

Bengio, Yoshua et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Harb, Jean and Precup, Doina. Investigating recurrence and eligibility traces in deep q-networks. *arXiv preprint arXiv:1704.05495*, 2017.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997.

Jaderberg, Max, Mnih, Volodymyr, Czarnecki, Wojciech Marian, Schaul, Tom, Leibo, Joel Z, Silver, David, and Kavukcuoglu, Koray. Reinforcement learning with unsupervised auxiliary tasks. *To appear in 5th International Conference on Learning Representations*, 2017.

Konda, Vijay R and Tsitsiklis, John N. Actor-critic algorithms. In *Advances in neural information processing systems*, pp. 1008–1014, 2000.

LeCun, Yann, Bengio, Yoshua, and Hinton, Geoffrey. Deep learning. *Nature*, 521(7553):436–444, 2015.

Lillicrap, Timothy P, Hunt, Jonathan J, Pritzel, Alexander, Heess, Nicolas, Erez, Tom, Tassa, Yuval, Silver, David, and Wierstra, Daan. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Rusu, Andrei A., Veness, Joel, Bellemare, Marc G., Graves, Alex, Riedmiller, Martin, Fidjeland, Andreas K., Ostrovski, Georg, Petersen, Stig, Beattie, Charles, Sadik, Amir, Antonoglou, Ioannis, King, Helen, Kumaran, Dharshan, Wierstra, Daan, Legg, Shane, and Hassabis, Demis. Human-level control through deep reinforcement learning. *Nature*, February 2015.

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

Peng, Jing and Williams, Ronald J. Incremental multi-step q-learning. *Machine learning*, 22(1): 283–290, 1996.

Puterman, Martin L. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

Rummery, Gavin A and Niranjan, Mahesan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

Schulman, John, Levine, Sergey, Moritz, Philipp, Jordan, Michael I, and Abbeel, Pieter. Trust region policy optimization. *CoRR, abs/1502.05477*, 2015.

Seijen, Harm and Sutton, Rich. True online td (lambda). In *International Conference on Machine Learning*, pp. 692–700, 2014.

Sharma, Sahil, Lakshminarayanan, Aravind S., and Ravindran, Balaraman. Learning to repeat: Fine grained action repetition for deep reinforcement learning. *To appear in 5th International Conference on Learning Representations*, 2017.

Silver, David, Lever, Guy, Heess, Nicolas, Degris, Thomas, Wierstra, Daan, and Riedmiller, Martin A. Deterministic policy gradient algorithms. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pp. 387–395, 2014. URL http://jmlr.org/proceedings/papers/v32/silver14.html
Silver, David, Huang, Aja, Maddison, Chris J, Guez, Arthur, Sifre, Laurent, Van Den Driessche, George, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Sutton, Richard S. Learning to predict by the methods of temporal differences. *Machine learning*, 3 (1):9–44, 1988.

Sutton, Richard S. and Barto, Andrew G. Introduction to reinforcement learning. *MIT Press*, 1998.

Sutton, Richard S, McAllester, David A, Singh, Satinder P, Mansour, Yishay, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pp. 1057–1063, 1999.

Todorov, Emanuel, Erez, Tom, and Tassa, Yuval. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

Watkins, Christopher J. C. H. and Dayan, Peter. Technical note: Q-learning. *Mach. Learn.*, 8(3-4):279–292, May 1992.
Appendix A: Experimental Details

Since the baseline scores used in this work are from Sharma et al. (2017), we use the same training and evaluation regime as well.

On hyper-parameters

We used the LSTM-variant of A3C (Mnih et al., 2016) algorithm for the CARA3C and LRA3C experiments. The async-rmsprop algorithm (Mnih et al., 2016) was used for updating parameters with the same hyper-parameters as in Mnih et al. (2016). The initial learning rate used was $10^{-3}$ and it was linearly annealed to 0 over 100 million time steps, which was the length of the training period. The $n$ used in $n$-step returns was 20. Entropy regularization was used to encourage exploration, similar to Mnih et al. (2016). The $\beta$ for entropy regularization was found to be 0.01 after hyper-parameter tuning, both for CARA3C and LRA3C, separately. The $\beta$ was tuned in the set $\{0.01, 0.02\}$. The optimal initial learning rate was found to be $10^{-3}$ for both CARA3C and LRA3C separately. The learning rate was tuned over the set $\{7 \times 10^{-4}, 10^{-3}, 3 \times 10^{-3}\}$. The discounting factor for rewards was retained at 0.99 since it seems to work well for a large number of methods (Mnih et al., 2016; Sharma et al., 2017; Jaderberg et al., 2017). The most important hyper-parameter in the LRA3C method is the $\lambda$ for the $\lambda$-returns. This was tuned extensively over the set $\{0.05, 0.15, 0.5, 0.85, 0.9, 0.95, 0.99\}$. The best four performing models have been reported in Figure 14b. The best performing models had $\lambda = 0.9$.

All the models were trained for 100 million time steps. This is in keeping with the training regime in Sharma et al. (2017) to ensure fair comparisons to the baseline scores. Evaluation was done after every 1 million steps of training and followed the strategy described in Sharma et al. (2017) to ensure fair comparison with the baseline scores. This evaluation was done after each 1 million time steps of training for 100 episodes, with each episode’s length capped at 20000 steps, to arrive at an average score. The evolution of this average game-play performance with training progress has been demonstrated for a few tasks in Figure 3. An expanded version of the figure for all the tasks can be found in Appendix C.

Table 1 in Appendix B contains the raw scores obtained by CARA3C, LRA3C and A3C agents on 22 Atari 2600 tasks. The evaluation was done using the latest agent obtained after training for 100 million steps, to be consistent with the evaluation regime presented in Sharma et al. (2017) and Mnih et al. (2016).

Architecture details

We used a low level architecture similar to Mnih et al. (2016); Sharma et al. (2017) which in turn uses the same low level architecture as Mnih et al. (2015). Figure 1 contains a visual depiction of the network used for CARA3C. The common parts of the CARA3C and LRA3C networks are described below:

The first three layers of both the methods are convolutional layers with same filter sizes, strides, padding and number of filters as Mnih et al. (2015, 2016); Sharma et al. (2017). These convolutional layers are followed by two fully connected (FC) layers and an LSTM layer. A policy and a value function are derived from the LSTM outputs using two different output heads. The number of neurons in each of the FC layers and the LSTM layers is 256. These design choices have been taken from Sharma et al. (2017) to ensure fair comparisons to the baseline A3C model and apply to both the CARA3C and LRA3C methods.

Similar to Mnih et al. (2016) the Actor and Critic share all but the final layer. In the case of CARA3C, Each of the three functions: policy, value function and the confidence value are realized with a different final output layer, with the confidence and value function outputs having no non-linearity and one output-neuron and with the policy and having a softmax-non linearity of size equal to size of the action space of the task. This non-linearity is used to model the multinomial distribution.
### Appendix B: Table of Raw Scores

| Name               | CARA3C | LRA3C | A3C   |
|--------------------|--------|-------|-------|
| Amidar             | 973.73 | 954.79| 1028.34|
| Assault            | 2670.74| 2293.74| 1857.61|
| Asterix            | 16048.00| 19139.00| 2364.00|
| Bank Heist         | 1378.60| 1382.97| 1731.40|
| Beam Rider         | 5076.10| 2099.71| 2189.96|
| Bowling            | 40.94  | 26.90 | 16.88 |
| Breakout           | 678.06 | 733.87| 555.05|
| Centipede          | 3553.82| 3226.94| 3293.33|
| Demon Attack       | 17065.22| 69373.42| 26742.75|
| Freeway            | 4.9    | 19.97 | 17.68 |
| Frostbite          | 310.00 | 289.27| 306.8 |
| Gopher             | 9468.07| 9626.53| 9360.60|
| James Bond         | 421.50 | 118.33| 285.5 |
| Kangaroo           | 1777.33| 16.67 | 26.00 |
| Koolaid            | 66.67  | 663.67| 1136  |
| Name this game     | 11354.37| 11515.33| 12100.80|
| Phoenix            | 10407.83| 6084.73| 5384.1 |
| Sea quest          | 4056.4 | 13257.73| 2799.60|
| Space Invaders     | 2703.83| 3609.65| 1268.75|
| Star Gunner        | 24550.33| 44942.33| 39835.00|
| Tutankhamun        | 245.86 | 271.46| 252.82|
| Wizard of Wor      | 5529.00| 2205.67| 3230.00|

Table 1: Game Playing Experiments on Atari 2600

All the evaluations were done using the agent obtained after training for 100 million steps, to be consistent with the evaluation paradigm presented in [Sharma et al., 2017](#) and [Mnih et al., 2016](#). Both CARA3C and LRA3C scores are obtained by averaging across 3 random seeds. The scores for A3C column were taken from Table 4 of [Sharma et al., 2017](#).
Appendix C: Training Graphs

The evaluation strategy described in Appendix A was executed to generate training curves for all the 22 Atari tasks. This appendix contains all those training curves. These curves demonstrate how the performance of the CARA3C and LRA3C agents evolves with time.

Figure 7: Training curves for CARA3C and LRA3C
Appendix D: Evolution of confidence-based weights with training time

This appendix reports the results of an expanded version of the experiments done in Section 4.3. If we assume that states are numbered from 1, then for the set of states

\[ S_{t \equiv 1 \pmod{20}} = \{ s_t : t \equiv 1 \pmod{20} \} = \{ s_1, s_21, \cdots \} \]

the confidence values were used for obtaining the weights \( w_i \) as demonstrated in Section 3.2. These weights were plotted as a function of training progress. Figure 8 demonstrates that in general, the confidence-based weights assigned to \( n \)-step returns based on larger \( n \) become larger with training progress.

Figure 8: Evolution of weights given to each of the \( n \)-step returns (where \( n \leq 20 \)) with training time.
A similar plot for all states in the set $S_t \equiv 10 \pmod{20}$ is in Figure 9. Note that these states can only bootstrap from 10 possible next states, because of the way that A3C is implemented. Hence, these states use a 10-sized vector for confidence-based weights and subsequently the plots of these weights versus training time have ten lines. This plot helps us conclude that the trend of the confidence-based weights assigned to $n$-step returns based on larger $n$ becoming larger with training progress is not limited to the set of states $S_t \equiv 1 \pmod{20}$ but generalizes to states in the set $S_t \equiv 10 \pmod{20}$ as well.

Figure 9: Evolution of weights given to each of the $n$-step returns (where $n \leq 10$) with training time
Note that plots similar to Figure 9 and Figure 8 can be obtained for any Set $S_t \equiv i \pmod{20}$ with $i \in \{1, 2, \cdots, 19\}$. However, for the sake of keeping the length of this manuscript tangible we report such plots only for other extreme case, namely for the set of states $S_t \equiv 19 \pmod{20}$. Such states can bootstrap from only two possible next states and hence the plots of their corresponding weights versus training time have two lines.

Figure 10: Evolution of weights given to each of the n-step returns (where $n \leq 2$) with training time
Appendix E: Evolution of weights assigned to $n$-step returns within an Episode

This appendix is dedicated to understanding how the weights assigned to $n$-step returns evolve during the duration of a single episode. Similar to the previous section $S_{t \equiv 1 \pmod{20}}$ is defined as:

$$S_{t \equiv 1 \pmod{20}} = \{ s_t : t \equiv 1 \pmod{20} \} = \{ s_1, s_21, \cdots \}$$

Next, states in the set $S_{t \equiv 1 \pmod{20}}$ are sorted and the weights assigned to various $n$-step returns are plotted versus the index in that sorted set. This demonstrates how the weights evolve with the passage of time within an episode. Figure 11 demonstrates that in general the weights assigned to various $n$-step returns stabilize with the progress of time within an episode.

This seems to indicate that the vast majority of improvement brought about by our algorithm CARA3C is through the dynamic bootstrapping done at the beginning of the episode and that it becomes less important towards the end of the episode. This is to be expected as the Value estimates become more accurate towards the end of the episode. Figure 12 contains a plot similar to Figure 11 albeit for the set of states $S_{t \equiv 10 \pmod{20}}$. 

Figure 11: Evolution of weights given to each of the $n$-step returns(where $n \leq 20$) with training time

Figure 12: Evolution of weights given to each of the $n$-step returns(where $n \leq 20$) with training time
Figure 12: Evolution of weights given to each of the n-step returns (where \( n \leq 10 \)) with training time.
Figure 13 contains a plot similar to Figure 11 albeit for the set of states $S_{t \equiv 19 \pmod{20}}$

Figure 13: Evolution of weights given to each of the $n$-step returns (where $n \leq 2$) with training time
Appendix F: Algorithm for training CARA3C

The algorithm corresponding to CARA3C, our main novel contribution has been presented in Algorithm 8. A similar algorithm can be constructed for LRA3C easily.

Algorithm 1 CARA3C

1: // Assume global shared parameter vectors θ.
2: // Assume global step counter (shared) \( T = 0 \)
3: \( K \leftarrow \) Maximum value of \( n \) in \( n \)-step returns
4: \( T_{\text{max}} \leftarrow \) Total number of training steps for CARA3C
5: \( \pi \leftarrow \) Policy of the agent
6: Initialize local thread’s step counter \( t \leftarrow 1 \)
7: Let \( \theta' \) be the local thread’s parameters
8: repeat
9: \( t_{\text{init}} = t \)
10: \( d\theta \leftarrow 0 \)
11: \( dw \leftarrow 0 \)
12: Synchronize local thread parameters \( \theta' = \theta \)
13: confidences \( \leftarrow [] \)
14: Obtain state \( s_t \)
15: states \( \leftarrow [] \)
16: repeat
17: states.append(\( s_t \))
18: Sample \( a_t \sim \pi(a_t|s_t; \theta') \)
19: Execute action \( a_t \) to obtain reward \( r_t \) and obtain next state \( s_{t+1} \)
20: confidences.append(\( c(s_{t+1}; \theta') \))
21: \( t \leftarrow t + 1 \)
22: \( T \leftarrow T + 1 \)
23: until \( s_t \) is terminal or \( t = t_{\text{init}} + K \)
24: if \( s_t \) is terminal then
25: \( R \leftarrow 0 \)
26: else
27: \( R \leftarrow V(s_t; \theta') \)
28: end
29: \( C \leftarrow \text{get_weights_matrix(confidences)} \)
30: \( G_{ij} \leftarrow \) the \( i \)-step return used for bootstrapping estimate for states[\( j \)] // \( K \times K \) matrix
31: for \( i \in \{t - 1, \ldots, t_{\text{init}}\} \) do
32: \( j \leftarrow i - t_{\text{init}} \)
33: \( R' \leftarrow R \)
34: for \( k \in \{K - 1, \ldots, j\} \) do
35: \( R' \leftarrow r_i + \gamma R' \)
36: \( G_{kj} \leftarrow C_{kj}, R' \) // Assuming 0-based indexing
37: \( R \leftarrow V(s_i; \theta') \)
38: end
39: for \( i \in \{t - 1, \ldots, t_{\text{init}}\} \) do
40: \( j \leftarrow i - t_{\text{init}} \)
41: TD-target \( \leftarrow \sum_{j=0}^{K-1} G_{ij} \)
42: Gradients for \( \theta \) based on \( \pi \): \( d\theta' \leftarrow d\theta' + \nabla_{\theta} \left( \log(\pi(a_i|s_i; \theta')) \left( \text{TD-target} - V(s_i) \right) \right) \)
43: Gradients for \( \theta \) based on \( V \): \( d\theta' \leftarrow d\theta' + \nabla_{\theta} \left( \text{TD-target} - V(s_i; \theta') \right)^2 \)
44: Perform asynchronous update of \( \theta \) using \( d\theta' \)
45: until \( T > T_{\text{max}} \)
Algorithm 2 Creates a 2D weights matrix with the confidence numbers

1: function GET_WEIGHTS_MATRIX(C)
2: $w \leftarrow \text{softmax}(C)$
3: $W \leftarrow 1 \otimes w^T$ // $W$ is a $K \times K$ weight matrix with $W_i = w^T$. $\otimes$ is outer product.
4: for $w_{i,j} \in W$ do
5: if $i > j$ then
6: $w_{i,j} \leftarrow 0$
7: for $w_i \in W$ do
8: $w_i = \frac{w_i}{\text{sum}(w_i)}$
9: return $W$

Choice of Important hyper-parameters for our methods

(a) Tuning architecture for CAR
(b) Tuning the $\lambda$ for LR

Figure 14: Tuning the important hyper-parameters for our methods

Perhaps the most important hyper-parameter in CARA3C is the network which calculates the confidence values. Since gradients do not flow back from confidence computation to the LSTM controller, this becomes an important design choice. We experimented extensively with different types of confidence computation networks including shallow ones, deep ones, wide ones and narrow ones. We found a "zero hidden layer" network on top of the LSTM controller (much like one which computes the value function) works the best (Figure 14a). Similarly, the most important hyper-parameter in $\lambda$-returns is the $\lambda$ from eq. (3). While we experimented with a large number and range of values for $\lambda$ the best performing ones have been reported in Figure 14b.