Hierarchical, Discontinuous Agent Reinforcement Learning Rewards in Complex Military-Oriented Environments

Charles Newton[*], **Christopher Ballinger, Michael Sloma**
Soar Technology, Inc
3600 Green Court
Ann Arbor, MI

Keith Brawner
U.S. Army Combat Capabilities Development Command
Orlando, FL 32826

Abstract

Artificially intelligent agents are seeing increased adoption in both the video game and simulation industry for training, education, and entertainment purposes. These systems often need realistic and believable opponents that must achieve objectives in the face of competing and contradictory priorities and frequently require the rapid creation of a wide spectrum of agents with disparate behaviors that reflect tactical realism. This in turn drives the need for the dynamic training of such agents from available source data. Approaches to do so have yet to be widely investigated due to the smaller scales of these simulation environments. This paper discusses techniques to quickly design and generate a variety of AI agents that follow desired tactics and procedures, including realistic situations that require trade-off decisions between competing objectives. Techniques described include an investigation into deep reinforcement agents that have separable reward structures and can prioritize and re-prioritize goals based on a hierarchy.

Introduction: Use of training simulations in the military

The United States Army has invested in 6 primary Modernization efforts (Lynch 2020), including improving soldier lethality through training. The Soldier Lethality Cross-Functional Team (CFT) leverages technology such as a synthetic training environment (STE) to enable soldiers to acquire the skills to perform militarily-relevant tasks (Rozman 2020). Simulation offers the ability to perform many soldiering tasks while enabling cost-savings with the introduction of artificially intelligent agents.

Soldiers in these virtual training environments require oppositional forces (OPFORs) to train and hone tactics against. Artificially intelligence agents provide an opportunity to provide soldiers with a variety of OPFOR that are tailorable in accordance to preferred difficulty level and taking generally appropriate actions in response to new situations in accordance with understood doctrine.

Deep reinforcement learning (DeepRL; see Li's 2017 review) promises to provide military simulation environments with flexible, realistic OPFORs that can adapt to a variety of situations and settings. However, DeepRL has yet to make significant in-roads into military simulation environments. This is due to several factors including: 1) inefficiency in the number of simulation runs required to train DeepRL agents (Botvinick et al. 2019), 2) current DeepRL methods are prone to learning unrealistic and unexpected solutions through specification gaming (Rossi and Mattei 2019), 3) DeepRL policies are blackbox methods that do not explain to system stakeholders why they selected particular actions (Heuillet, Couthouis, and Díaz-Rodríguez 2021), and 4) DeepRL approaches have difficultly handling data inconsistencies that characterize interactions with the real-world (Irpan 2018).

This paper investigates the application of DeepRL methods that employ reward signals that are hierarchical, discontinuous, and conflicting in nature to address each of these pain points and, as a result, enable the greater adoption of DeepRL in the military simulation world.

Background: Deep Reinforcement Learning

The goal of reinforcement learning is to learn agent behavior through interaction with an environment. We consider a reinforcement learning setup consisting of an agent's interactions with an environment over discrete timesteps t. Agents operate in partially-observed environments summarized by a real-valued vector x_t, take discrete actions a_t, and receive a scalar reward $r_t(s_t, a_t) \in [0, 1]$. Agents' behaviors are determined by a learned policy π which maps observations into a probability vector over the discrete action choices. Agent policies are optimized to maximize the expected value of future reward $R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r_i(s_i, a_i)$. Note that the actions chosen depend on the policy function π.

Deep Reinforcement Learning with Discontinuous and Conflicting Rewards

Many real-world policy learning situations deal with conflicting priorities and goals. For example, agent goals designed to maximize adversary losses may conflict with engagement rules surrounding preserving civilian infrastructure. We consider the problem of learning policies that have these conflicting goal structures that take the form of policies that maximize rewards with caveats. Formally, the reward signal r_i is assumed to be a sum of rewards r_i^k associated

[*]Direct correspondence to: charles.newton@soartech.com
Copyright © 2021by the authors. All rights reserved.

with conditions $(c_1^k, \ldots, c_m^k)$ from a space of conditions C:

$$r_i(s_i, a_i) = \sum_k g(r_i^k(s_i, a_i)) \quad (1)$$

where $g(\cdot)$ is a gate function that determines if preconditions are met for receiving the reward signal:

$$g(r_i^k(s_i, a_i)) = \begin{cases} g(r_i^k(s_i, a_i)) & if \ \texttt{true}(c_1^k, \ldots, c_m^k) \\ 0 & \textit{otherwise} \end{cases}$$

Agents are trained in an on-policy manner and g induces discontinuities in the reward signal $r_i(s_i, a_i)$ as conditions turn on or off. Conditions can be written hierarchically to ensure that certain mission aspects (e.g., those related to mission safety) are prioritized above others (e.g., engaging a threat.) Agent rewards are generally designed against the following principles:

1. Large macro-goals, such as winning a mission, are established as over-arching rewards.
2. Agents are decomposed as required into hierarchical elements such as a strike or supply package.
3. Gates are primarily determined by observing undesired behavior was observed and captured in the form of conditions.
4. Supplemental rewards are established that are correlated with necessary conditions for achieving rewards. For example, an attack agent is provided supplemental rewards for moving into a position from which they can attack. The primary purpose of supplemental rewards is to accelerate training.

Background: DeepRL in Defense Applications

Our work uses several emerging technologies for agent-based development in defense applications. These enabling technologies include DeepAgent, a development framework for rapidly writing trainable agents, and LARIAT, a testing environment for instrumenting agent metrics in simulation and gaming environments.

DeepAgent

DeepAgent is a development pipeline for training various agents across different environments. SoarTech started developing DeepAgent as an internal tool to help accelerate the development of new machine learning approaches and the training of agents in an environment-agnostic manner. DeepAgent helps facilitate the development process by providing a generic interface for both environments and agents. New environments can be added by writing a new adapter using environment API, allowing existing agents to start training in the environment without further adaption. Likewise, developing a new agent that follows the API allows the agent to begin training against existing environments. All agents described in this paper were developed in the DeepAgent environment.

LARIAT

LARIAT (Brawner, Sottilare, and Ballinger 2022) is a testing pipeline meant to provide a user-friendly front-end for making adjustments to agents, running experiments, and creating/gathering/comparing performance metrics. LARIAT takes the information submitted by users and the front-end, submits it to DeepAgent, and gathers and organizes performance statistics for end-user presentation. While LARIAT currently uses DeepAgent for the processing back-end, in the future LARIAT may integrate with different agent models such as reinforcement learning more broadly, state machines, and cognitive architectures.

Training Environments

There are several training environments AI researchers can draw from in today's research landscape. Most of these training environments offer a wide variety of research environments, and usually offer tools and editors that allow developers to customize scenarios with new challenges. However, each of these environments has their own technical requirements and pose different technical challenges that AI researchers must contend with when deploying their agents in these new environments. As such, we limited ourselves to two environments that provided easy integration and have a long history of use in prior research: StarCraft II (SC2) and the Rapid Integration & Development Environment (RIDE).

StarCraft II

StarCraft II (SC2) is a real-time strategy game developed by Blizzard Entertainment. SC2 provides an environment for several players to compete with potentially hundreds of units for control of the map and resources. DeepMind has created an API for Python (PySC2) (Vinyals 2017) which provides an interface for interacting with SC2 including receiving operations and sending actions to the environment. The environment allows for custom scoring and also includes a map score which is $+5$ for each enemy killed and -1 for each friendly unit killed. Our research primarily utilizes SC2 as an interface for creating agents to control and accomplish tasks while needing to work around OPFORs in the area.

RIDE

The Rapid Integration & Development Environment (RIDE) is a testbed environment developed by the University of Southern California Institute for Creative Technologies. RIDE was created with DoD communities in mind, such as the Army's STE efforts. RIDE provides a collection of tools, APIs, and object prefabs for the Unity game engine/editor, in order to make authoring interactive training scenarios easy. RIDE also utilizes Unity's ml-agent library, to provide an interface to researchers for training artificial agents against RIDE scenarios. Since DeepAgent already has a ml-agents compatible wrapper implemented, and the goals of our research and the goals of RIDE are likely of interest to many of the same research communities, we leveraged RIDE in our current research efforts.

Agent Goals and Rewards

Agents trained via reinforcement learning achieve various goals by giving the agent a reward (or penalty) based on how the agent's past decisions affected progress towards the goals. This section describes specific design considerations developed to allow continuous agents in real-time, long-duration environments to learn in the face of potentially conflicting goals. For example, a scenario may have the objective to safely escort a civilian to a safe-house, while maintaining goals such as minimizing military casualties and maximizing enemy casualties to help achieve the objective.

In the simplest case, a single objective function provides the agent with enough positive and negative reinforcement to learn a behavior that achieves the desired goal. However, in our complex scenarios, not only might there be multiple goals, but they may compete and fluctuate over time as the scenario progresses and situations change. We utilize hierarchical, discontinuous rewards for reinforcement learning in training agents for these complex scenarios.

Baseline reinforcement learning

We created some simple scenarios to act as a baseline for PPO (Schulman et al. 2017) agent training. These scenarios were created with only one squad for the agent to control, and with several constant goals all focused on progressing towards one objective. Training agents against these single team, static goal scenarios serve as a basic point of reference we can look to when analyzing the time and performance of training agents in scenarios with more stochastic and discontinuous rewards.

Continuous environments and long time horizons

In the past, reinforcement learning environments have traditionally been turn-based problems, such as chess (Schrittwieser et al. 2020) and Go (Silver et al. 2017). Recent research has focused on environments such as Real-Time Strategy (RTS) games like SC2 (Vinyals 2019). These games provide two challenges for learning agents to overcome. First, much like chess, agents must plan strategically not only for the near future, but over more distant time horizons as well. Secondly, the agent must operate in a real-time, continuous environment. Unlike chess where players take turns, giving the player time to deliberate over a static board state, in real-time environments each player acts independently of the other, constantly taking as many actions as they want whenever they want. Not only does this mean an agent must act against a constantly changing state, but the agent must make decisions quickly or risk being outmaneuvered by their opponent. Also, unlike chess, which has discrete, finite locations where the agent can choose to move pieces, RTS games allow the agent to move their game pieces to extremely precise locations on the board, allowing for nearly limitless possibilities for positioning and maneuvering.

Cooperative planning through shared situational awareness

In the scenarios we developed, we expected that successful strategies would require multiple squads handling different goals in tandem. For example, one squad may be in charge of safely escorting a high value target to a bunker, while a separate squad engages in more high-risk activity such as intercepting and engaging hostile groups. In order for these different squads to cooperate towards a common mission objective, we experimented with different methods that allowed an agent to control both squads and prioritize actions. We experimented with two action representations to allow the agent to control the different squads. In our first representation, the actions for each group were encoded separately, allowing the agent to issue one action to one group in each simulation step. In the second representation, actions for each group were encoded in pairs, allowing the agent to issue one order to both groups in each simulation step.

Results

Agent concepts were validated and prototyped in a set of increasingly complex StarCraft II scenarios that reflect elements of dismounted infantry tactics. The composition and results for each scenario are presented below.

Banshee vs Corruptor

This scenario utilized a combination of ground and air units on both sides; On the blue team the ground units (Marines) could attack ground and air units and the air units (Banshees) could only attack ground units. On the red team, the ground units (Zerglings) could only attack ground units and the air units (Corruptor) could only attack air units. The goal was to see if the blue team agent could learn to use this attack parity to its advantage; thus the reward function was to minimize blue force casualties and to maximize red force casualties. We observed that the agent would send in the blue team ground units to bate the red team ground units out and then engage the red team ground units with both the air and ground units. The agent would then send the ground units to finish off the red team's air units. As shown in 6, the transition from engaging Zerglings to Corruptors is primarily based on observing the threat's HP and map position.

Shoot-and-Scoot

In this scenario, we simulated a "Shoot-and-Scoot" tactic where we have 9 Marines against 5 Zerglings and 4 Banelings. Banelings explode when around enemy units and do area damage so they can affect multiple units in the area where they explode. The goal is for the agent to learn to move the marines further back when the red team comes closer and engage the red team when they are further away. We tried an additional scenario where the Marines have faster movement, so they can more easily take advantage of "Shoot-and-Scoot" tactics.

In Figure 1 we plot the reward of the original slower marines against the faster marines and see that the faster marines are able to maximize the reward better and more quickly than their original counterparts.

In Figure 2 we decompose the reward signals for the slower marines to show the different rewards that we used during training. The *Map Score* is provided by the PySC2 environment and is $+5$ for eliminating an enemy unit and -1 for losing a friendly unit. The *Attack Bonus* of $+2$ is awarded when the agent is within attacking distance, but not too close to the enemy to be attacked, and chooses to perform an attack. The *Move Bonus* of $+10$ is awarded when the agent decides to move somewhere when the enemy is too close. The bonus scores are then added to the *Map Score* to get the *Overall Score*.

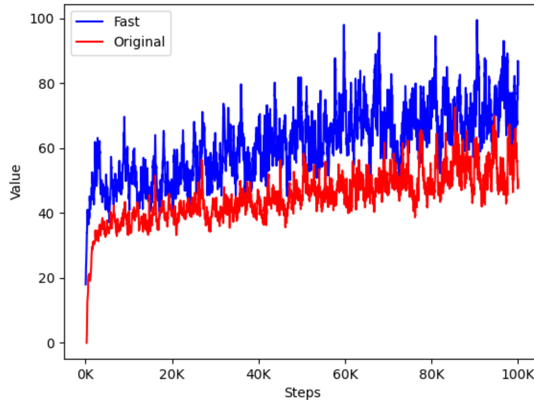

Figure 1: Reward scores for original vs fast movement over 100k training steps for “Shoot-and-Scoot”, used smoothing factor of 0.9

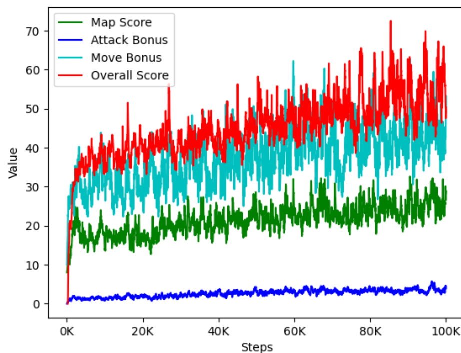

Figure 2: Reward score components for slower Marines over 100k training steps, used smoothing factor of 0.9

Chokepoint Tactics

In this scenario, shown in Figure 3, we attempt to lure the red team into a chokepoint where the agent will send out a scout which will lure the red team to where the blue team is waiting to ambush. We began by testing if the map score is enough to sufficiently train this action, however we end up with a direct attack on the enemy where the blue team loses. We then crafted a more rigid sequence of events that the agent needed to go through in order to successfully execute the chokepoint tactic. However, the agent was unable to learn the more ridged sequence of events, was unstable and skipped through most of the events and lost to the enemy. From this, we formed the idea of using more discontinuous reward functions.

Figure 3: Chokepoint Scenario Layout

Small Hierarchical Group v1

In this scenario, the blue team needs to engage OPFOR while protecting a high value target (HVT) as the HVT is in transit to a safe house. We split the blue team into two groups, a fire team and an escort team; the fire team engages the red team and the escort team ushers the HVT to safety. The teams had different reward functions that would trigger at different times depending on the observation state and spatial position within the scenario the actions were taking place.

The escort group’s reward function penalized the agent when the HVT was damaged or the group moved away from the safe location, while it was rewarded for moving towards the safe location, only if no friendly units were actively taking damage. The fire team was rewarded for damaging enemy units only if no friendly units were actively taking damage. Thus to gain any positive score, the agent needed to have no friendly units taking damage. The hierarchical reward structure is displayed in Figure 4.

We see that the fire team moves to engage the enemy and the escort team stands still (as there is no incentive for them to move while the fire team engages the enemy). Once the enemy has been cleared out, the escort team moves to the safe location.

Small Hierarchical Group v2

Similar to Small Hierarchical Group, this situation had the same overall goal but had cover positions on the map and utilized a different reward function. In this situation, we introduced the idea of “orders” where the agent could put itself into one of three possible states. While in these states there was a reward for picking a state and a penalty for leaving the state before the task was complete. There were three possible orders that the agent could issue:

- Offensive Combat: While hostile units exist, but are not an immediate threat to allies

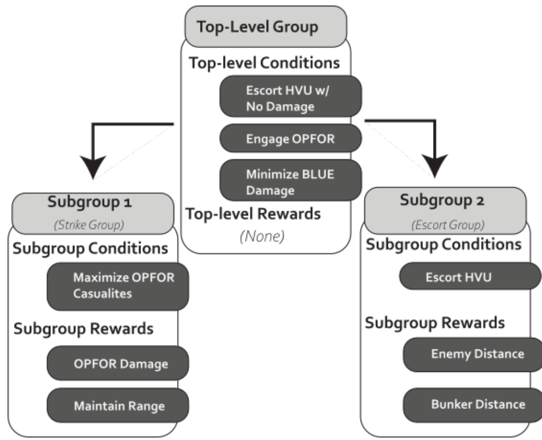

Figure 4: Small Hierarchical Group Rewards and Conditions

- Defensive Combat: While hostile units exist, and allies have received damage recently
- Clear: While hostile units have been eradicated

More formally, each order configuration can be modeled as a state machine with a set of states $Q : \{q_0, \ldots, q_n\}$. Each state q_j is associated with a separate reward function r_{q_j}. A transition function $\delta : Q \times C \rightarrow (Q, r_{q_j})$ rewards the agent on transition if condition c_{q_j} is met and penalizes otherwise. Agent transitions between orders are determined by conditions within C triggering a change in state. Reward functions r_{q_j} may themselves be separable and discontinuous as described in Equation 1, meaning the order configuration adds an additional layer of potential discontinuity.

In this small hierarchical group scenario, the two groups of units had their own reward functions in an order configuration. The escort group received a positive reward when they moved away from the enemy only while a "Defensive Combat" or "Offensive Combat" order had been issued and a positive reward when they moved closer to the safe location only when a "Clear" order had been issued. The fire team received a positive reward when they did damage to enemy units when the order of "Offensive Combat" was issued and received a positive reward when they moved further from the enemy when "Defensive Combat" was issued.

We noticed that the fire team would move to engage the enemy when the situation began and the escort group would move towards the safe location. Once the fire team began to take damage, the escort group would change goals and move away from the enemy even though it was further from the safe zone. Once the enemy was cleared out, the escort group would move to the safe zone.

Rapid Training Time

All agents were trained on a single Dell XPS 15 9560 laptop, with 16GB of RAM, Intel Core i7-7700HQ 4 x 2.8GHz CPU, and NVIDIA GeForce GTX 1050 Mobile GPU. Agents were trained and tested in our environments by running the simulation at faster than real-time. Most agents were trained over the course of about 300,000 simulation steps, which took less than four hours of wall clock time to complete for each scenario.

Validating Tactics Through Explainable Deep Reinforcement Learning

Reinforcement learning is a black-box model (Rudin and Radin 2019) that does not easily provide explanations for why agent policies produce specific action choices. As a result, reinforcement learning approaches can silently exhibit failure modes such as "specification gaming" that produce actions that successfully collect reward while being entirely disconnected from the behavior intended by the system designer. Techniques from explainable AI (Gunning et al. 2019) can mitigate these concerns by producing semantic explanations of system behavior. System designers can then verify these explanations align with their intention. This section reviews the application of sensitivity analysis to produce interpretations of policy behavior.

Policy sensitivity analysis

In Figures 5 and 6 we show histograms that display what changes in the observation space x_t were present when the agent's policy π decided to switch to a new action, rather than staying with the original action. Cumulative histograms were constructed that tallied each of the observed changes in x_t at each action switch. These histograms provide an explanation as to the specific observations that led to new actions and provide insight into the decision procedures learned by π.

We primarily looked for large differences in values in the histogram columns as this represents a larger change in the observation when the agent decided to go with the new action (listed on the y-axis). In Figure 5, which was taken at the beginning of a Bansehe vs Corruptor scenario (described in more detail in the Results section), we see in the columns that there is overall very few differences in the values per column, other than for the columns associated with marine health (*marine_hp*) and the spatial position of the marine group (*marine_group_x* and *marine_group_y*), we see some larger deltas for the attack actions. Rows and columns do not have large differences which indicate high uncertainty over the actions that should be executed in the scenario, which is an expected result for a scenario that is just beginning.

Figure 6 was taken at the end of the scenario, we see these deltas become more pronounced, as the policy has greater certainty about what actions to take and which input to observe when making decisions. This indicates that the agent only looks for specific information when switching to attack actions.

Conclusions

Many real-world situations, particularly those found in military applications, require agents to execute actions under goals that conflict. A hierarchical reward design that deliberately gates reward mechanisms on and off dependent on agent conditions can be leveraged to rapidly engineer a wide variety of complex agent tactics in simulation environments.