Reinforcement Learning to Control

a Commander

for Capture The Flag

Student: Liam Bressler  Faculty Advisor: Sekhar Tatikonda,  Associate Professor of Electrical Engineering & Statistics

Abstract – Reinforcement learning (RL) is a machine learning area that has many applications in artificial intelligence. While it has advantages over more rigid techniques, the correct application of RL models in complex games require extensive domain knowledge and testing, giving a learning agent a strong human prior. The goal of this paper is to attempt to create as unsupervised an RL algorithm as possible in the game Capture-the-Flag (CTF), a multifaceted game that requires analysis of a large amount of state variables.

First, a previous implementation of RL in a software environment created by the platform AISandbox is used to explore this issue. The previous program creates a commander for CTF. After the human aspects of the previous implementation are summarized, ideas are discussed for lowering the amount of supervision a learning algorithm needs, and the feasible ideas are applied into the creation of new commanders. These commanders are tested against both themselves and built-in programs provided by AISandbox. The results are then analyzed, examining tasks that the computer did well, areas where a human prior might be necessary, and possible combinations of successful algorithms. Promising avenues including the changing of training times, even to extreme levels, and the introduction of a large amount of possible commands. These approaches are sometimes unintuitive, and this paper explores why they might work. Finally, applications of this research to other fields of artificial intelligence are discussed, including which successful semi-supervised techniques are specific to CTF, and which may have broader applications.
1. Introduction

The development of artificial intelligence is one of the key goals of machine learning. While human-like intelligence has not yet been approached, much progress has been made on narrower fields such as neuroscience, linguistics and mathematics. There are a wide variety of AI fields, and many of them are concerned with learning strategies for winning games.

In particular, reinforcement learning (RL) has stood out as a tool for an AI to “learn” strategies, and it has had some success (examples). Unlike algorithms that have to be applied in very specific contexts, RL can be applied to dynamic games with many variables, such as Pac-Man, the game of life, or capture-the-flag. Thus, RL can be seen as semi-supervised learning.

A major issue with RL, however, is that in a game with a decent amount of complexity, a prior weighting of events is needed. This is not a problem for programmers who want to combine RL with human intuition. However, RL has weaknesses when a programmer doesn’t have enough domain knowledge to properly weight events. In this sense, RL is too supervised to make inroads in this area, and other approaches must be explored.

This paper attempts to explore how unsupervised a program can be while still providing a viable strategy. The main method to do this is to extend a method in Ivanovic et al.’s paper [1] that attempts to beat other commanders in “Capture the Flag.” While “RLCommander” learns a policy throughout a series of games, it is first given carefully crafted weights to 14 different variables, and necessitates 44 state variables. The goal of this paper is to make this, or a similar method, as unsupervised as possible, while still concocting a viable strategy that wins games. This could be achieved in a variety of ways: taking away weights, making the state variables as vague as possible, etc.

In addition, this paper explores the basics of reinforcement learning and unsupervised learning, and discusses possible combinations between the two, as well as their distinct advantages and disadvantages.

2. Problem Domain

Capture-the-flag (henceforth known as CTF) is a popular game in which the goal is to “capture” a flag at a known point, then carry it to another known point, called the “score location”. A team can defend the other from scoring by “tagging” them, forcing them to return to their base for a set period of time. The simulation environment used in this paper, created by AISandbox [2], usually has a player the player “start point”, flag area, and flag-return area as far away from each other as possible, creating the need for a long-term policy for linking many goals together. In this implementation, a game is played for a set amount of time, and the team that brings the flag to the score location the most times in this period wins the game.

The AISandbox environment adds a few wrinkles to the typical CTF game. Instead of the usual “tag”, AISandbox allows members of a team to shoot at enemy team members at range; firing delay is inversely dependent on how quickly a player is moving, so a successful strategy needs to weigh the benefits of moving and shooting quickly. Also, CTF games can be played on a variety of maps, with varying team sizes, topologies, and flag capture points. Thus, a successful commander should not make a strategy that only works on one type of map. Finally, AISandbox provides each maps with a variety of obstacles, such as boxes which agents can’t “see” over, which necessitate strategies other than “run and shoot at the nearest target”.

In AISandbox CTF, a team of players, who we’ll call “bots” from now on, is controlled by a central commander. The commander is not omnipotent, but it has access to the sightlines of all bots, the flag and score locations, and other general game information. Because of the combination of changing game settings and the partially observable environment, RL is a natural choice for learning a winning strategy. The wide variety of maps would normally be a source of difficulty, but RL provides a framework for building an adaptable, sustainable strategy for any environment.

RL has been applied to Capture the Flag before [1][4]. Ivanovic et al.’s paper [1] provides a good framework, which is summarized in the next two parts. However, this paper implements the system with one key difference, which we will note later on.

3. Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning in which an agent receives a reward for performing an action in its environment. There are no strict rules for which action an agent should take. Instead, an agent learns a long-term strategy over time, taking into account its current state and reward.
A basic RL model consists of:

- A policy: a mapping of current environmental states to actions to be taken. Basically, a policy is the general way an agent should act at a specific time.
- A reward function: A function that maps each state-action pair to a single number, based on the desirability of the agent. While it can only be calculated in the short term, the sole goal of an agent is to maximize its long-term reward.
- A value function: As opposed to the short-term desirability of reward, a value function describes the long-term desirability of a state taking into account which states are likely to follow.
- A model, which predicts reward and value from state-action pairs, and the transition probabilities between these pairs. More formally, the model consists of a set of actions, a set of states, the expected reward of transitioning of states, and the described transition probabilities.

Many reinforcement learners use discrete-time Markov Decision Processes as their models. However, as [1] notes, AISandbox CTF has some actions which take longer to implement than others, so [1] uses Semi-Markov Decision Processes (SMDP) as the model; this allows the agent to pick from a set of options, which can vary in execution length. The advantages of this can be confirmed from a simple implementation of MDP’s in the AISandbox environment; it is impossible for a bot to learn any long-term strategy if it is not allowed to take only short actions.

This paper implements the SMDP model used in [1], which was first used by Sutton et al. [3] Formally, the SMDP consists of a set of states $S$, a set of options $O$, the expected reward $r_\omega^t$, where $r_\omega^t = \{ r_{t+1} + y r_{t+2} + \cdots + y^{k-1} r_{t+k} | \epsilon(o,s,t) \}$

and a set of transition probabilities $p_s^{o,k} = \sum_{k=1}^{\infty} p(s',k) y^k$

where $\epsilon(o,s,t)$ denotes the event of $o$ being initiated in state $s$ at time $t$, $t+k$ is the random time at which $o$ terminates and $k$ is the number of steps after taking option $o$ in state $s$. [1][3]

In [1]’s SMDP,

\[
Q(s,o) = Q(s,o) + \alpha [ r + y^k \max_{o' \in O} Q(s',o') - Q(s,o) ]
\]

[1][3]

which is an adapted Q-learning algorithm. In addition, [1] implements eligibility traces, which mark certain events as eligible for learning. This is important in cases where certain actions lead to the capturing of a flag; if these events also lead to the scoring of a point, then they should have an even greater reward, and eligibility traces allow state-action pairs’ rewards to be changed multiple steps after they have already been used.

RL has not only been used in Capture the Flag, as [1] notes. These games include one-on-one fighters [5] and shooting games [4][6]. However, only [1] has the property of having a central commander lead multiple agents.

4. Previous Implementation

[1] then implemented this reinforcement learning algorithm to AISandbox CTF. Their implementation is summarized briefly here.

[1] used 44 variables to summarize each state, “22 which are equivalent among bots and 22 which may differ…dependent on the bot.” [1] These included such myriad factors as “whether there are allies nearby…or whether the controller has more bots alive than the enemy.” [1] Although it made attempts to shorten the state space, this results in a large Q-learning table, and requires a bit of domain knowledge to build.

AISandbox has four built-in commands: (all definitions taken from commenting of source code)

Move – Order a bot to run to a specified position without attacking visible enemies.

Attack – Order a bot to attack a specified position. If an enemy bot is seen by this bot, it will be attacked.

Charge – Order a bot to attack a specified position at a running pace.

This is faster than Attack but incurs an additional firing delay penalty.

Defend - Order a bot to defend its current position.

Defend has the lowest firing delay among the four. [1] extended these four commands into eight custom-made ones; Flank, Greedy Path, Greedy Path Cautious, Defend, Attack Team Flag, Intercept, Run_Home_Greedy, Run_Home_Cautious, and Ambush. (The first three commands are ways to get to the enemy team flag, while the sixth and seventh are meant to score an already captured flag). Once again, all of these commands required domain knowledge, as many useless commands needed to be weeded out.

Finally, [1] carefully crafted a reward function for its SMDP. Here are a few examples:
Captured Flag: 40
Returning Flag Close [to score location]: 2
Not Returning Flag: -3
Killed Enemy: 10
Defending in Spawn: -7
[No one on team] Holding Flag: -4

While the reward functions are well-mapped to good/bad events, these weights are carefully tuned. [1] addresses this by admitting that “defining a reward function...relies on us correctly rewarding the agent in order to get the behavior we desire.”

[1] also discusses the difficulty of getting an agent to link together a policy once a flag is picked up. Thus, the authors had to implement a constant negative reward for the flag holder if it was not close to the score location.

The authors tested the system against four built-in commanders that AISandbox provided. While they are beatable by human-programmed AI, each built-in commander provided a challenge by implementing a unique strategy. For instance, the Defender Commander had all agents but one defend their own flag, with one attacker.

5. Discussion of Possible Lowering of Supervision

While [1] was able to create a decent strategy, and one that did better as it learned and was adaptable, it required a carefully crafted set of commands and reward function. While this is not an issue for someone who knows the game well, this paper is concerned with creating a commander that is less supervised and requires less domain knowledge. With that in mind, some ideas to give the bots less of a human influence are summarized, split into sections:

A. Reward function

[1] does not specifically state how it came up with its reward function, but it is fair to assume that it required some human trial-and-error. Perhaps a more general reward function would allow the commander to create a policy more on its own. Some ways to do this include:

Binary reward function: Each “good” event has a reward of 1, each “bad” event has a reward of -1, and each neutral event has a reward of 0. This could let the commander realize the weighting of certain events, while creating a light prior distribution so that the bots aren’t completely helpless during the first few rounds.

Fewer elements of reward function: The agent is only rewarded for point-scoring events, flag captures, and kills. This could take away some of the arbitrariness of certain awards, such as whether or not a bot is close to the flag, and base the reward function on solely results. This would lean heavily on eligibility traces because of the paucity of reward events.

Using time as a reward function: If a training commander can cause a drawn-out loss rather than a quick loss, it is possible that the commander a solid defensive strategy which could be negatively rewarded. Thus, rewarding a commander’s previous 50 moves, if the enemy team has not scored in the past 50 moves, is an option.

B. Command Set

Fewer commands in command set: Since there are only eight commands in [1]’s command set, this is probably counterproductive, because the agent would have very few options to choose from.

More commands in command set: This is more promising. [1]’s command set seems tailored to human perceptions of the game. It is possible that commands that don’t seem to have a use (such as moving to a random spot on the map) could have a place in a viable long-term policy.

C. State Space

Smaller number of states: It is possible that 44 state variables leads to a policy that is too sparse, and thus makes it difficult for an agent to learn a long-term policy in a reasonable amount of time.

Larger number of states: With similar logic to adding more commands, adding a larger number of states could counter human perceptions of what is useful information in CTF. However, since there is no way to weed out states, this would cause the state space to grow enormously, and would likely cause a much higher training time for a commander.

D. Training Time

Shorter/longer training time: Although neither a very short nor very long training period seems ideal, a commander may need a different amount of time than a human might think it needs to learn.

6. Experiments

To test these possibilities for lowering supervision of the CTF commanders, four steps were taken:

- Each of the ideas in (5) was programmed into a commander, using the framework provided by [1].
- A few games against the “PlaceholderCommander” provided by AISandbox were analyzed. If a commander
failed to show any sign of progress, it was discarded.
- The surviving commanders played a set of games against each other, using a round-robin format.
- Each commander played against the gauntlet of the built-in commanders provided by AISandbox (following the format of [1]).

A. Game Format

A “game” is played between two coded commanders on the AISandbox environment. The commanders compete for 5 minutes to bring as many flags to their respective score locations as possible, with dead bots respawning every 45 seconds. (A game could be accelerated to take fewer than 5 minutes while preserving the amount of actions taken by the commanders.) A set of games is a 40-game long matchup between two commanders, taking place on one randomly chosen map.

Three possible maps were used: a small map (map01), a medium-sized maps with many obstacles (map25), and a somewhat larger map that allowed freedom of movement. The three-map set was chosen in order to strike a balance between variety, so that a commander had to be adept at multiple strategies in order to succeed, and in-match consistency, so that a commander had time to learn a policy.

The topologies included a spawn location, a flag location and a score location for each team. White spaces indicate spaces where an individual agent can run and see across, black spaces designate obstacles which impede a bot's sightline, and gray space indicates unpassable terrain that can be seen and shot over. A bot can see in a 90 degree cone, spanning about one-fifth of the height of map01.

The experiments used the default team size for each map, provided by the AISandbox initialization files. Map01 and map25 required 15 bots per team, while map34 needed 13 bots per team. It is possible that certain strategies would work better or worse with varying number of bots.

B. Commanders

Based on the ideas in Part 5, the experiment involved ten different commanders.

Each was programmed using the framework provided in [1], with one major change; when a bot picks up an enemy flag, it immediately asks the Q-learning algorithm for an option, and uses a different command set than bots that don't possess the flag. From observation, this allows a Q-learning commander to perform much better at bringing a flag home. Although this may seem more artificial than every bot using the same command set, it allows more differentiation between performances of different commanders. We feel that the increased performance caused by this change is worth the small loss of generality, as the AI's performances are easier to evaluate on a wider scale.

Also, [1] does not list all 44 of its state variables (s.v.), so this paper implements either AISandbox-provided s.v.'s or variables inferred from [1]. All 44 state variables are Boolean values. They are not listed here because of length concerns.

Finally, each commander trained for 30 games and tested (i.e., no experimentation) for the final 10 games, unless otherwise noted.

Each commanders' name is listed in bold, implemented with the same framework as above, except for the changes listed in italics:

**BinaryCommander:** Each positive reward is change to 1, and each negative reward is change to -1. (This is opposed to rewards on a scale from -20 to 40)

**FewRewardsCommander:** The agent is only rewarded as such:
- Captured Flag: 40
- Picked Up Flag: 15
- Killed Enemy: 10
- Died: -5
- Enemy Picked Up Flag: -10
- Captured Flag: -20

(This eliminates all rewards that try to spur bots to action, such as the constant negative reward for having no bots moving to the flag.)

Eligibility traces are applied to an entire bot's actions since its last death (or the beginning of the game).

**FewRewardsDefaultNegativeCommander:** The same as above, but the commander is giving a default reward of -1. This is so that the algorithm will gradually drop policies that encourage inaction.

map25  map01
StalemateRewardCommander: Same as FewRewardsCommander, but if the enemy team has not picked up a flag in the last ten seconds, the previous ten seconds of moves are given a small reward.

FewCommandsCommander: Only the moves Greedy_Path, RunHome_Greedy, Intercept, and Defend are included in the list of commands. This is enough to span all necessary functions. Flag-bearing bot command sets remain the same.

ManyCommandsCommander: The original eight commands from [1], as well as Randomly_Charge, Randomly_Move, Randomly_Attack, Patrol_Area and Follow_Ally are included in the command set. All of the “Randomly_X” commands order a bot to perform the order X directed at a random spot within the bot’s half of the map.

FewStatesCommander: The twenty-two state variables deemed the least necessary by the researcher were removed. These included s.v.’s such as SightlineBlocked, IsCloseToMultipleEnemies, and RespawnEventSoon.

ManyStatesCommander: Twenty-two state variables were added, most of possibly tenuous importance. Examples include IsRunningParallelToAlly, IsOnKillingStreak, and RespawnEventRecently.

ShorterTrainingTime: Training time changed to 15 games, testing time changed to 25 games.

LongerTrainingTime: Training time changed to 40 games, no testing time.

The researcher’s hypothesis before the simulations was that both TrainingTime commanders and the FewCommandsCommander would perform the best, and that the ManyCommandsCommander would be unviable.

C. Commander Viability

Before any competition occurred, each commander was tested against the PlaceholderCommander provided by AISandbox, which provided a bare-bones opponent to determine the basic viability of an algorithm. Each commander was observed for a maximum of 10 games; if a system showed progress in a shorter time period, it moved on to the next round.

If a commander showed no progress towards a workable strategy or was otherwise unviable, it was eliminated. The rejected algorithms for this round were the:

FewRewardsDefaultNegativeCommander: Because the training commander is weak during the beginning, the vast majority of rewards this commander received were -1 for “inaction.” While this reward is appropriate when it is one of many, it appears as if using it as the sole basis for a reward function is untenable.

ManyStatesCommander: The 66 possible states made for an unsustainably large state space. While the other algorithms showed varying aptitudes at exploring better as more games were played, this commander seemed to pick randomly far too often, as many state spaces had not been reached yet.

D. RL Commander vs. RL Commander

The eight remaining commanders were then pitted against each other in a round-robin format. Each matchup included a set of games on a random choice of one of the three maps. Because the single per matchup could result in advantages for certain commanders, each algorithm played on each map a minimum of two out of seven times.

The results in Figure 1 show the results show the result of each matchup, the overall win rate for each commander, and the average number of flags
captured per game for the matchups that a commander was involved in (a proxy for how aggressive an algorithm was). Ties were counted as 0.5 points.

Some notes from observation, with the worst-performing bots first:

- The **BinaryCommander** was the most inept. It was able to judge when a situation was a complete disaster (such as learning to avoid leaving no defenders), but its inability to judge the scale of events seemed to be its downfall. Figure 2 shows the declining

- The **FewStatesCommander** also performed poorly, losing to all seven sets of games. This algorithm’s shortcomings were difficult to exactly diagnose. For example, it had the largest losing streak (8 games) against the difficult Training commanders, but seemed to adjust near the end. This was in contrast to FewStates’s match against the Binary, in which its late performance deteriorated against a weak opponent. Also, the FewStatesCommander had the most unpredictable policy-making from the point of view of the researcher.

- The **FewRewardsCommander** and **FewCommandsCommander** had similar behavior, with the latter being more consistent. Both algorithms developed simple policies based on offense. From observation, FewCommands was much stronger at the beginning of the game than FewRewards, as the pruned command space seemed to make individual bots more efficient. Based on Figure 1, these two performed exceptionally well against weaker competition and poorly against stronger commanders, perhaps signifying that bare-bones policies fail at higher levels.

- The **StalemateRewardsCommander** had the most ties, and its results tended towards 20 out of 40 wins more often than any other commander. This algorithm was very defensive often dropping all of its bots back in the late stages of game sets, although policies seemed to take longer to develop.

- The **ManyCommandsCommander** fared very well during late games. While its early results were unimpressive, it adjusted its policies toward its individual opponents very well, as Figure 2 demonstrates. (In particular, the algorithm employed the command “Randomly Charge” much more often than I would expect.) However, ManyCommands somewhat struggled against the strong Training commanders, although it still performed better in this area than its five counterparts.

- In one sense, it is not a surprise that **ShorterTrainingTime** and **LongerTrainingTime** performed the best as they are closest to classic RL. However, in practice these commanders acted very differently. While they competed at similar levels, ShorterTraining had more variable results, perhaps suggesting that 10 games was not enough to guarantee an acceptable policy. ShorterTraining performed better against simple opponents (FewRewards and FewCommands), while LongerTraining executed at a higher level against adaptable opponents, as it was much harder to read throughout the game.

In general, **ManyCommandsCommander** was the algorithm that completed at a higher level than was hypothesized, as it was able to adapt very well to a wide range of strategies.

**E. RL Commander vs. Built-In Commander**

ManyCommandsCommander and the two Training commanders were then pitted against four built-in commanders provided by AISandbox. In addition, ManyCommandsCommander was combined with both shorter and longer training times to form two new participants.

The built-in commanders, named Greedy, Defender, Balanced and Random, implemented simple yet effective algorithms that provided a decent challenge. Compared to this experiment’s custom commanders, the built-Ins did not adapt throughout the set of games, and they made use of bot assignment to different roles (such as “attacker” and “defender”). Thus, these commanders were expected to perform well in the early game.

The results of the sets of games, done in the same format as Part D, are as follows (each score is out of a possible 160 points, as one match was played against each built-in commander):

**ShorterTrainingTime**: 32.5 points

**ManyCommandsCommander**: 29 points

**LongerTrainingTime**: 27.5 points

**ManyCommands/ShorterTraining**: 20.5 points

**ManyCommands/LongerTraining**: 13.5 points

As in Part D, ShorterTraining performed better than LongerTraining against inflexible counterparts. From observation, LongerTraining’s “experimental” moves cost it more often than not against disciplined opponents. This reinforces the notion that LongerTraining’s added randomness only helps to throw off adaptable opponents.

While the combination of two suboptimal strategies predictable performed worse than one suboptimal
strategy, it is interesting to note that one amalgamation worked much better than the other. ManyCommands/ShorterTraining often decided on nonsensical policies, such as having no bot attempt to get the flag, if its truncated training period had too many anomalies.

As a point of reference, the commander in [1] that this paper implemented scored 40.5 points in the same trial, run locally.

6. Conclusions

A. Explanations for Performance

In terms of performance, the extreme-training-time algorithms performed the best against both RL commanders and built-in opponents. While LongerTrainingTime performed better against certain opponents, ShorterTrainingTime competed at a higher level against quality competition, especially when its training period was fairly normal. This points to possible diminishing returns for having a lengthy 30-game learning period; the decay for our RL learning rates may not have been enough.

The ManyCommandsCommander proved the most adaptable out of all competitors. It had a far richer possible policy states than most of its counterparts, and the trade-off of a few weak early games for overall malleability was a worthy one. The commander came up with a few “non-human” policies that the researcher observed; for instance, the commander seemed to use “Randomly Charge” as a filler for otherwise unusable bots. While the algorithm could use some improvement against regimented competition, its performance demonstrated the usefulness of not having a human fine-tune an RL model.

Meanwhile, the other algorithms showed little promise, with the possible exception of FewCommandsCommander. This failure means that both the state space and the magnitude of the reward function are crucial for our RL model in CTF; perhaps this is because neither can be changed throughout the course of learning. The FewCommandsCommander, however, would be interesting to explore further, because it didn’t require as much domain knowledge to build. This is opposed to the ManyCommandsCommander, which, while successful, needed a human eye for what exactly a command could be. So, FewCommands should be considered the greatest success in this experiment that reduces human artificiality.

B. Applicable to Other Fields?

While [1] and [4] both carefully crafted RL models, this paper successfully showed some ways in which these models could be simplified. The foremost of these changes were changes to the training time and action space.

The training time changes caused varying types of performance, but minimal drop-off in level of performance. While different RL applications will obviously differ in scope, perhaps this experiment shows that future modelers should not be afraid to drastically change the training times for reinforcement learning.

While the expansion of the command space caused minimal drop-off in CTF, such a change may not be cause similar negligible effects in outside applications. One still has to be able to deduce a proper action in order to add to the action space of an RL model; while “randomly charge to an area” seems irrational to someone who has played the game for a while, it would take some domain knowledge to even consider it as an option. Therefore, the addition of more actions should be an option for reducing supervision, but it may not be feasible in many cases, and may in fact add to increased supervision.

Finally, the decent performance of the reduced-action-space commander holds some promise for semi-supervised models. However, the algorithm’s policies didn’t show much nuance or adaptability, so perhaps this approach is not widely applicable.

The results of this experiment bode well for RL models on environments that have a changeable set of commands. However, some aspects of RL-based artificial intelligence may always need some human fine-tuning, such as an agent’s current state and its reward function, although this may be different for models not based off of points and wins.

However, these results also show potential for reduction of supervision in a model’s action space. While previous implementations of RL in CTF required a specific amount of customized actions, this paper implemented a model that varied these actions with minimal change in performance.

Ivanovic, Zambetta, Li, Rivera-Villicana.
Reinforcement Learning to Control a Commander for Capture-the-Flag, 2014
