Transfer Learning Between RTS Combat Scenarios Using Component-Action Deep Reinforcement Learning

Richard Kelly and David Churchill
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada
richard.kelly@mun.ca, dave.churchill@gmail.com

Abstract

Real-time Strategy (RTS) games provide a challenging environment for AI research, due to their large state and action spaces, hidden information, and real-time gameplay. StarCraft II has become a new test-bed for deep reinforcement learning systems using the StarCraft II Learning Environment (SC2LE). Recently the full game of StarCraft II has been approached with a complex multi-agent reinforcement learning (RL) system, however this is currently only possible with extremely large financial investments out of the reach of most researchers. In this paper we show progress on using variations of easier to use RL techniques, modified to accommodate actions with multiple components used in the SC2LE. Our experiments show that we can effectively transfer trained policies between RTS combat scenarios of varying complexity. First, we train combat policies on varying numbers of StarCraft II units, and then carry out those policies on larger scale battles, maintaining similar win rates. Second, we demonstrate the ability to train combat policies on one StarCraft II unit type (Terran Marine) and then apply those policies to another unit type (Protoss Stalker) with similar success.

1 Introduction and Related Work

Real-Time Strategy (RTS) games are a popular testbed for research in Artificial Intelligence, with complex sub-problems providing many algorithmic challenges (Ontañón et al. 2015), and the availability of multiple RTS game APIs (Heinermann 2013; Synnaeve et al. 2016; Vinyals et al. 2017), they provide an ideal environment for testing novel AI methods. In 2018, Google DeepMind unveiled an AI agent called AlphaStar (Vinyals et al. 2019a), which used machine learning (ML) techniques to play StarCraft II at a professional human level. AlphaStar was initially trained using supervised learning from hundreds of thousands of human game traces, and then continued to improve via self play with deep RL, a method by which the agent improves its policy by learning to take actions which lead to higher rewards more often. While this method was successful in producing a strong agent, it required a massive engineering effort, with a team comprised of more than 30 world-class AI researchers and software engineers. AlphaStar also required an enormous financial investment in hardware for training, using over 80000 CPU cores to run simultaneous instances of StarCraft II, 1200 Tensor Processor Units (TPUs) to train the networks, as well as a large amount of infrastructure and electricity to drive this large-scale computation. While AlphaStar is estimated to be the strongest existing RTS AI agent and was capable of beating many players at the Grandmaster rank on the StarCraft II ladder, it does not yet play at the level of the world's best human players (e.g. in a tournament setting). The creation of AlphaStar demonstrated that using deep learning to tackle RTS AI is a powerful solution, however applying it to the entire game as a whole is not an economically viable solution for anyone but the worlds largest companies. In this paper we attempt to demonstrate that one possible solution is to use the idea of transfer learning: learning to generate policies for sub-problems of RTS games, and then using those learned policies to generate actions for many other sub-problems within the game, which can yield savings in both training time and infrastructure costs.

In 2017, Blizzard Entertainment (developer of the StarCraft games), released SC2API: an API for external control of StarCraft II. DeepMind, in collaboration with Blizzard, simultaneously released the SC2LE with a Python interface called PySC2 designed to enable ML research with the game (Vinyals et al. 2017). AlphaStar was created with tools built on top of SC2API and SC2LE. PySC2 allows commands to be issued in a way similar to how a human would play; units are selected with point coordinates or by specifying a rectangle the way a human would with the mouse. Actions are formatted as an action function (e.g. move, attack, cast a spell, unload transport, etc.) with varying numbers of arguments depending on the function. This action representation differs from that used in other RTS AI research APIs, including the TorchCraft ML library for StarCraft: Broodwar (Synnaeve et al. 2016). In this paper we refer to the action function and its arguments as *action components*. Representing actions as functions with parameters creates a complicated action-space that requires modifications to classic RL algorithms.

Since the release of PySC2 several works other than AlphaStar have been published using this environment with deep RL. Tang et al. (2018) trained an agent to learn unit and building production decisions (build order) using RL while handling resource harvesting and combat with scripted modules. Sun et al. (2018) reduced the action space by training

Figure 1: 8m vs. 8m scenario showing game screen (left) and PySC2 features used (right).

an agent to use *macro actions* that combine actions available in PySC2 to play the full game of StarCraft II against the built-in AI. Macro actions have also been used in combination with a hierarchical RL architecture of sub-policies and states and curriculum transfer learning to progressively train an agent on harder levels of the built-in AI to learn to play the full game of StarCraft II (Pang et al. 2019). Samvelyan et al. 2019 introduced a platform using the SC2LE called the StarCraft Multi-Agent Challenge used for testing multi-agent RL by treating each unit as a separate agent. Notably, these other works have not used the complex action-space directly provided by PySC2, which mirrors how humans play the game.

This paper is organized as follows: in the next section we describe how our experiments interact with the the SC2LE; following that, we present our implementation of component-action DQN; in the Experiments section we describe our experimental setup and results; finally we present our conclusions and ideas for expanding on this research.

2 StarCraft II Learning Environment

The PySC2 component of the SC2LE is designed for ML/RL research, exposing the gamestate mainly as 2D feature maps and using an action-space similar to human input. An RL player receives a state observation from PySC2 and then specifies an action to take once every n frames, where n is adjustable. The game normally runs at 24 frames per second, and all our experiments have the RL player acting every 8 frames, as in previous work (Vinyals et al. 2017). At this speed the RL player acts at a similar rate as a human, and the gamestate can change meaningfully between states.

2.1 PySC2 observations

PySC2 observations consist of a number of 2D feature maps conveying categorical and scalar information from the main game map and minimap, as well as a 1D vector of other information that does not have a spatial component. The observation also includes a list of valid actions for the current frame which we use to mask out illegal actions. In our experiments we use a subset of the main map spatial features relevant to the combat scenarios we use:

player_relative - categorical feature describing if units are "self" or enemy (and some other categories we don't use)

selected - Boolean feature showing which units are currently selected (i.e. if the player can give them commands)

unit_hit_points - scalar feature giving remaining health of units, which we convert to 3 categories

2.2 PySC2 actions

Actions in PySC2 are conceptually similar to how a human player interacts with the game, and consist of a function selection and 0 or more arguments to the function. We use a small subset of the over 500 action functions in PySC2 which are sufficient for the combat scenarios in our experiments. Those functions and their parameters are:

no_op - do nothing

select_rect - select units in a rectangular region

- screen (x, y) (top-left position)
- screen2 (x, y) (bottom-right position)

select_army - select all friendly combat units

attack_screen - attack unit or move towards a position and stop to attack enemies in range on the way

- screen (x, y)

move_screen - move to a position while ignoring enemies

- screen (x, y)

Any player controlled with PySC2 will still have its units automated to an extent by built-in AI, as with a human player. For example, if units are standing idly and an enemy enters within range, they will attack that enemy.

3 Component-Action DQN

For this research we implemented the DQN RL algorithm, first used to achieve human-level performance on a suite of Atari 2600 games (Mnih et al. 2015), with the double DQN enhancement (Van Hasselt, Guez, and Silver 2016), a dueling network architecture (Wang et al. 2015), and prioritized experience replay (Schaul et al. 2016). The number of unique actions even in the small subset we are using is too large to output an action-value for each one (for instance, if the screen size is 84x84 pixels, there are 7056 unique "attack screen" commands alone). To address this problem we output a separate set of action-values for each action component. Our version of DQN, implemented using Python and Tensorflow, features modifications for choosing actions and calculating training loss with actions that consist of multiple components (component-actions). Algorithm 1 shows our implementation, emphasizing changes for handling component-actions.

Invalid action component choices are masked in several parts of the algorithm. Random action function selection in step 8 is masked according to the valid actions for that state. When choosing a non-random action using the network output in step 11, the action function with the highest action-value is chosen first, disregarding actions marked as unavailable in the state observation, and then each parameter to the action function is chosen according to the highest action-value (step 11). Invalid choices for parameters to the action

Algorithm 1 Double DQN with prioritized experience replay and component-actions for StarCraft II

1: **Input:** minibatch size k, batch update frequency K, target update frequency C, max steps T, memory size N, $M_{min} \leq N$, initial ϵ and annealing schedule
2: $M[] \leftarrow \emptyset$
3: Initialize Q with random weights θ, components $d \in D$
4: Initialize Q_{tar} with weights $\theta_{\text{tar}} = \theta$
5: Initialize environment env with state s_1
6: **for** $t = 1$ to T **do**
7: **if** rand() $< \epsilon$ **or** $t \leq M_{min}$ **then**
8: Select valid action function a_t^0 randomly
9: Select action parameters $a_t^1, a_t^2, \ldots$ randomly
10: **else**
11: $a_t = \{a_t^d = \arg\max_{a^d} Q^d(s_t, a^d) \mid d \in D\}$
12: Take action a_t in environment, observe s_{t+1}, r_{t+1}
13: $M[t \mod N] \leftarrow (s_t, a_t, r_{t+1}, s_{t+1})$
14: Priority$[M[t \mod N]] = \max(\text{Priority})$
15: Anneal ϵ according to schedule
16: **if** $t \equiv 0 \mod C$ **and** $t > M_{min}$ **then** $\theta_{\text{tar}} \leftarrow \theta$
17: **if** $t \equiv 0 \mod K$ **and** $t > M_{min}$ **then**
18: Sample k transitions (s_j, a_j, r_j, s'_j) from M
19: $a^{0'} \leftarrow \arg\max_{a^0} Q^0(s'_j, a^0)$
20: $m^d \leftarrow 1$ **if** a^d is a parameter to $a^{0'}$, **else** $m^d \leftarrow 0$
21: **if** s'_j is terminal **then**
22: $y_j \leftarrow r_j$
23: **else**
24: Set y_j as in Equation 2
25: Update θ minimizing $\sum_d m^d L(y_j, Q^d(s_j, a_j^d))$
26: Priority$[j] \leftarrow \sum m^d |y_j - Q^d(s_j, a_j^d)|$

function could be masked out in steps 9 and 11 in general, but are not in this work since all values of the parameters we used (screen and screen2) are valid. Only parameters that are used by the chosen action function a_t^0 are used in the environment update in step 12.

3.1 Loss Functions

In double DQN network updates are made during training using the TD target value

$$y = r + \gamma Q_{tar}(s', \arg\max_{a'} Q(s', a')),$$

where r is the reward, γ is the discount, Q and Q_{tar} are the primary and target network, and s' and a' are the next state and action taken in the next state. With a single action component the loss to be minimized is $L(y, Q(s, a))$, where $L()$ is some loss function (e.g. squared error). When using component-actions, there are separate action-values for each component of the action. In general, an action is a tuple of action components $(a^0, a^1, ..., a^n)$, with an individual action depending on a subset of those components. That is, some components are not used for every action. This raises the question of how to calculate training loss when the action components used from one action to the next differ.

Several previous works have used variations of action components with different RL algorithms. Tavakoli, Pardo, and Kormushev (2018) use a method they call Branching Dueling Q-Network (BDQ) in several MuJoCo physical control tasks with multidimensional action spaces. In these tasks, all components are used in each complete action. They experimented with 1) using a separate TD target, y^d, for each action component a^d corresponding to a component $d \in D$, the set of all action components; 2) using the maximum y^d as a single TD target; and 3) using the average of the y^d as a single TD target, which was found to give the best performance:

$$y = r + \frac{\gamma}{|D|} \sum_{d \in D} Q_{tar}^d(s', \arg\max_{a'^d} Q_d(s', a'^d)) \quad (1)$$

Their training loss is the average of the squared errors of each component's action-value and y from Equation 1.

Action components were also used (Huang and Ontañón 2019) to represent actions in the microRTS environment to train an agent using the Advantage Actor Critic (A2C) algorithm, a synchronous version of the policy gradient method Asynchronous Advantage Actor Critic (A3C) (Mnih et al. 2016). In that work each component is not used in every action, but the policy gradient action log-probability ($log(\pi_\theta(s_t, a_t))$, where π_θ is the policy network) used in training is always the sum of the action log-probabilities for each action component. AlphaStar (Vinyals et al. 2019b) also uses a policy gradient method and sums the action log-probabilities from each action component, but masks out the contribution from unused components.

In our implementation, on step 24, we use the mean target component y as in Equation 1, but modified so that unused components of the target action are masked out using $m^d = 1$ if component d is in use for an action, $m^d = 0$ otherwise:

$$y = r + \frac{\gamma}{|D|} \sum_{d \in D} m^d Q_{tar}^d(s', \arg\max_{a'^d} Q_d(s', a'^d)) \quad (2)$$

The loss to be minimized with gradient descent on step 25 of Algorithm 1 is the sum of the losses of each used component's action-value compared to the target y:

$$L_{Total} = \sum_{d \in D_{used}} L(y, Q^d(s, a^d))$$

where D_{used} are the components in use for a particular action, and $L()$ is the Huber loss used in DQN (Mnih et al. 2015), which is squared error for error with absolute value less than 1, and linear otherwise.

We considered and tested two alternative loss calculations. The first method was to sum the Huber losses of the components compared pairwise with

$$y^d = \frac{r}{|D_{used}|} + \gamma Q_{tar}^d(s', \arg\max_{a^{d'}} Q^d(s', a^{d'})),$$

masking out unused components from the action taken but not from the target action. This method is similar to method 1) tested with BDQ (Tavakoli, Pardo, and Kormushev 2018). It has the advantage of comparing losses from different action component branches of the network to their corresponding branch, but since it updates the primary network toward

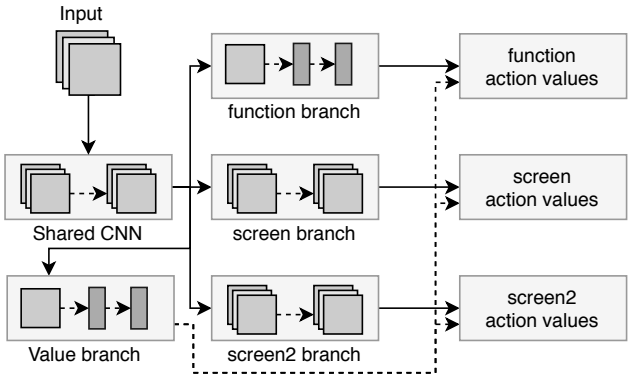

Figure 2: Diagram showing overall design of the network.

the action-values of unused components of the target action we didn't expect it to perform well.

The second alternative method we tried was to use the Huber loss of the average of the action-values of the action taken by the primary network (unused components masked out) compared with the same average y from Equation 1. This method seems reasonable since it uses the same formula to calculate both the primary network total action-value and discounted target network next state value. However, branches with positive and negative action-values can cancel each other out in both inputs to the loss function, resulting in a small loss when individual components of the primary network action-value may have a large error relative to the target network. Both alternative loss calculations resulted in worse performance and were not used in our final experiments.

3.2 Network and training parameters

The neural network used to predict action-values consists of a shared convolutional neural network (CNN), followed by separate branches for state value and the three action components we used (function, screen, screen2). The network's overall design is shown in Figure 2.

The state input categorical feature maps are first preprocessed to be in one hot format with dimensions 84x84x6. Next the input is run through a series of blocks consisting of parallel convolutional layers whose outputs are concatenated. First the one-hot input is run through each of 3 convolutional layers with 16 filters each and kernel sizes 7, 5, and 3. The output of those layers is concatenated along the channel dimension and given as input to each of two convolutional layers with 32 filters and kernel sizes 5 and 3. Those two outputs are concatenated and fed to a final convolutional layer with 32 filters and kernel size 3. All convolutional layers here use padding and a stride of 1 to keep the output to the same width and height, so as to preserve spatial data for the action parameters which target parts of the screen.

Next the network splits into a value branch and one branch for each action component. The value and function branches each have a max pooling layer of size and stride 3, followed by 2 dense layers of size 256. The function branch ends with a final dense layer outputting the action advantages of the function action component.

Both the screen and screen2 branches receive as input the output of the shared CNN, which is fed into a 32 filter 3x3 convolutional layer, followed by a 1 filter 1x1 convolutional layer as described in (Vinyals et al. 2017), giving output dimensions of 84x84x1 and the action advantages of each screen position. We experimented with adding the one-hot encoded output from the function branch as input to the screen branch followed by additional convolutional layers, and similarly with the screen and screen2 branch with a single layer added with value 1 in the position corresponding to the screen choice and 0 elsewhere, but found that for the action functions and scenarios used in these experiments there was surprisingly no gain in performance. We believe a larger network combined with more training time may be required to take advantage of these connections.

Each convolutional and dense layer (except those leading to action advantage outputs) is followed by a ReLU activation and batch normalization. Training hyperparameters were selected through informal search. We use the Adam optimizer with learning rate of 0.001, a discount of 0.99, batch size of 64, and L2 regularization. Minibatch training updates are performed every 4 steps, and the target network is updated every 10,000 steps. The prioritized experience replay memory size is 60,000 transitions, and all parameters are as described in the sum tree implementation (Schaul et al. 2016). In each training the exploration ϵ is exponentially annealed from 1 to 0.05 over the first 80% of total steps.

4 Experiments

We did three experiments to test the performance of our network structure and action representation in different environments. First, we compared performance when training for different amounts of training steps; second, we compared performance on different scenarios, ranging from small (4 units each) to large (32 units each) instances of a combat scenario; third, we tested transfer learning performance by using models trained exclusively on one scenario in other scenarios. Performance of all trained models is compared to that of a number of simple scripted players. All training and evaluation is done with the opponent being the game's built-in AI, i.e. the same opponent a human player playing the single-player game would face.

4.1 StarCraft II Combat Scenarios

Experiments were conducted on custom StarCraft II maps that each contain a specific combat scenario. The scenarios limit the size of the battlefield to a single-screen sized map, which removes the requirement to navigate the view to different areas of the environment. In each episode an equal number of units are spawned for both the RL or scripted player being tested, and a built-in AI controlled player. Units appear in two randomized clusters, randomly assigned to the left or right side of the map, which are symmetric about the centre of the map. The built-in AI enemy is immediately given an order to attack the opposite side of the screen, causing them to rush at the RL player's units, attacking the first enemies they reach. Once given this command, the in-game AI takes over control of the enemy units, which executes a policy

which prioritizes attacking the closest units of the RL player. This use of the built-in AI to control the enemy for combat experiments has been shown to be effective for testing and training combat algorithm development (Churchill, Lin, and Synnaeve 2017).

We trained and evaluated our models and scripted players on scenarios with equal numbers of Terran Marines (a basic ranged unit in the game) per side, numbering 4, 8, 16, or 32. The Marine scenarios will be referred to as "#m vs. #m", where # is the number of Marines. We also evaluated our models on scenarios with the same counts of Protoss Stalkers, referred to as "#s vs. #s". Stalkers are larger units with longer range and a shield which must be reduced to zero before their health will deplete. These scenarios were constructed in a symmetric fashion in order to give both sides an equal chance at winning each battle. If one side is victorious from an even starting position, it must mean that their method is more effective at controlling units for combat. An effective human policy for such scenarios involves first grouping up the players' units into a tight formation, and then using focus-targeting to most efficiently destroy enemy units.

Episodes end when all units of one side are destroyed (health reduced to 0), or until a timer runs out. The timer is set at 45 real-time game seconds for Marine maps, and 1:30 for the Stalker maps since Stalkers have more health and shields causing the battles to last longer. The map then resets with a new randomized unit configuration and a new episode begins. The maps output custom reward information to PySC2, which are calculated using the LTD2 formula (Churchill 2016), which values a unit as by its *lifetime damage*, a function of its damage output per second multiplied by remaining health. The unit values are normalized to equal 1 for a full health unit of the highest value in the scenario, and the difference in total unit value (self minus enemy) between steps is added to the step reward. Since there are positive rewards for damaging enemy units and negative rewards for taking damage, episode rewards tend to be similar in scenarios with different numbers of units. The damage part of the LTD2 calculation doesn't affect the rewards in scenarios that feature only one unit type, as in the experiments presented here, but it will affect future experiments with more scenarios with mixed unit types. The total reward per step is observed by the RL player in step 12 of Algorithm 1.

4.2 Training and Evaluation

We trained models in scenarios with 4, 8, 16, and 32 Marines per side, for 300k and 600k steps. In informal testing we found that 300k steps was enough to see good performance in these scenarios, and we chose double the number of steps to compare with a longer training time. Training rewards per episode for 300k and 600k steps is shown in Figure 3. Training was performed on a machine with an Intel i7-7700K CPU running at 4.2 GHz and an NVIDIA GeForce GTX 1080 Ti video card. It takes 5 hours to train for 300k steps using the GPU and running the game as fast as possible. To train a model we use PySC2 to run our custom combat scenarios for as many episodes as needed to reach the step limit. The RL player receives input from the game environment and takes actions as described in Algorithm 1.

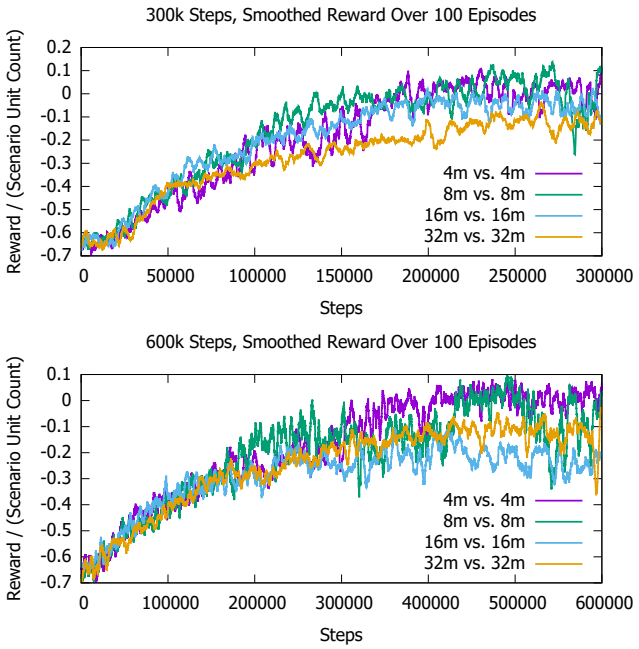

Figure 3: Model training reward (normalized to the number of Marines in the scenario) per episode (smoothed over 100 episodes) for 300k (top) and 600k (bottom) step models.

For each scenario/step count combination we trained three models and used the best performing of those for the transfer learning experiments. Trained models were evaluated by running 1000 episodes with a deterministic policy (i.e. $\epsilon = 0$), using the trained model for inference only. In evaluation, the result can be a win (1 point), draw (0.5) or loss (0). Wins occur when all enemy units are destroyed. Draws happen if both sides simultaneously lose their last unit. If the map timer runs out, the side with the most combined health/shields wins. If that metric results in a tie then the episode is counted as a draw. Evaluation results are presented as a score, equal to the number of wins plus half of the number of draws, divided by the number of evaluation episodes.

We also evaluated several scripted players:

No Action (NA) - This player takes the no_op action only, which is equivalent to a human player providing no input. Its units will attack and follow enemy units if they move into range, controlled by the in-game AI.

Random (R) - This player takes random actions. Both the function choice and any arguments are randomized.

Random Attack (RA) - This player selects all friendly units, and subsequently attacks random screen positions.

Attack Weakest Nearest (AWN) - This player selects all friendly units on the first frame and then attacks the enemy with the lowest health, choosing the enemy nearest to the average of the friendly units' positions as a tie breaker.

5 Results and Discussion

Results of our experiments can be seen in Table 1. The first column shows the scenario for which the experiment is being

Scenario	Scripted Players				Learned Policy - 300k Steps Trained on Marines				Learned Policy - 600k Steps Trained on Marines			
	NA	R	RA	AWN	4m	8m	16m	32m	4m	8m	16m	32m
4m vs. 4m	0.173	0.013	0.570	**0.907**	**0.691**	0.661	0.619	0.625	**0.663**	0.606	0.444	0.414
8m vs. 8m	0.146	0.019	0.561	**0.721**	**0.694**	**0.691**	**0.692**	0.541	**0.693**	0.585	0.356	0.440
16m vs. 16m	0.119	0.004	**0.391**	0.038	0.591	0.546	**0.695**	0.450	**0.643**	0.213	0.300	0.503
32m vs. 32m	0.044	0.002	**0.116**	0.000	0.329	0.280	**0.441**	0.332	0.340	0.247	0.130	**0.471**
4s vs. 4s	0.142	0.000	0.352	**0.910**	**0.497**	0.491	0.008	0.479	**0.525**	0.291	0.236	0.408
8s vs. 8s	0.179	0.000	0.240	**0.678**	**0.420**	0.406	0.015	0.386	**0.512**	0.192	0.204	0.303
16s vs. 16s	**0.211**	0.001	0.030	0.013	0.114	0.183	0.034	**0.356**	**0.288**	0.152	0.154	0.196
32s vs. 32s	**0.130**	0.000	0.002	0.000	0.075	0.239	0.223	**0.320**	**0.386**	0.193	0.355	0.170

Table 1: Experiment scores $((\text{wins} + \text{draws}/2)/\#\text{ episodes})$ of scripted players and learned policies evaluated for 1k episodes in multiple scenarios. Columns give the policy being evaluated, and rows give the scenario. e.g. the number in the 300k 4m column, 8m vs. 8m row gives the score of the model trained for 300k steps on the 4m vs. 4m scenario when evaluated on the 8m vs. 8m scenario. The best performing policy in each category (scripted, trained for 300k steps, and 600k steps) is bolded.

conducted along a row, with the other column values showing which policy controls the player units for the experiment.

5.1 Scripted Player Benchmarks

The results of the scripted players can be seen in columns 2-5 of Table 1. The performance of the benchmark scripted players on the Marine scenarios shows a large range of results. The NA policy scores as high as 0.173, as units will attack when enemies get within range even if the player does nothing. The random bot gets scores from 0 to 0.019, whereas the random attacking bot ranges from a score of 0.116 in the 32m vs. 32m scenario to 0.570 in the 4m vs. 4m scenario. This shows that simply choosing to attack all the time is a good strategy, but that it isn't enough to win often with higher unit counts (in the 32m vs. 32m scenario the bot is likely attacking its own units sometimes). AWN achieves a score of 0.907 in the 4m vs. 4m scenario, but scores 0 in the 32m vs. 32m scenario, indicating that formation becomes a bigger factor in more complex scenarios. We observed that units often get stuck while trying to move to attack the same target. The scripted players perform best in the 4m vs. 4m scenario and perform worse as the number of units increases in all cases except R in the 8m vs. 8m scenario. Scripted players perform similarly in the Stalker scenarios.

5.2 Learned Policies - Battle Size Transfer

Results for the learned policies can be seen in columns 6-9 (300k training steps) and 10-13 (600k training steps) of Table 1. We believe that these results yield two major observations. The first is that like scripted players, smaller sized battles yielded higher scores for learned policies. We believe this is because smaller battles are less complex, with far fewer possible states, and therefore are easier for learning effective policies. Also, smaller battles end faster, so more battles are able to be carried out in the same number of training steps. The second observation is that the experiments demonstrated the ability to transfer a policy trained on battles of one given size to another, while maintaining similar results. For example, a policy trained on 4 vs. 4 Marines for 600k time steps was able to obtain a score of 0.663 when applied to the 4 vs. 4 Marine scenario, and 0.643 when applied to the 16 vs. 16 Marines scenario, which was even better than the policy trained itself on 16 Marines. We did however notice that results get worse as the difference in unit count between training and testing scenarios gets larger, which was expected.

One surprising result however was that for several scenarios, the best policy was not one one that was trained on that same scenario. For example, the policy trained on 16 vs. 16 Marines for 600k time steps was the 2nd worst policy when applied in the 16 vs. 16 Marine scenario. Another surprising result was that most of the policies trained for 300k time steps ended up performing as good, or even better than those trained for 600k time steps. This indicates that either the scenarios are not complicated enough for more training to result in better policies, or that the variability in model performance is too large to see a trend for the number of models we trained.

By visual observation, most trained models learn to select all friendly units and then mainly attack. Some learn to target the ground near friendly units, causing those units to cluster. Some models also learn to target damaged enemy units.

5.3 Learned Policies - Unit Type Transfer

The results in the bottom 4 rows of Table 1 are for experiments carried out with policies learned on scenarios with Marine vs. Marine battles, but applied to scenarios with Stalker vs. Stalker battles. In general, the results show that while the scores for these Stalker scenarios are not as high as the Marine scenarios, the policies can indeed be transferred to units of different types. In particular, we can see that smaller sized scenarios perform well, with scores tapering off for larger scenarios. We believe this is due to the fact that as more units enter the battlefield, the differences between those units such as size and damage type become more apparent, causing the policy to perform worse. One surprising result was that the policies trained on 16 vs. 16 Marines, especially the one trained for 300k steps, performed much worse than the other policies overall, for which we currently have no explanation.

6 Conclusion and Future Work

In this paper we presented an application of component-action DQN to combat scenarios in a complex real-time strategy game domain, StarCraft II. We showed that with short training times and a relatively easy to implement RL system, good performance can be achieved in these combat scenarios. We successfully demonstrated transfer learning between battle scenarios of different sizes: that policies can be learned in a scenario of a given size, and then applied to scenarios of different sizes with comparable results. We also demonstrated transfer learning between different unit types, with policies being learned in scenarios with Marine unit battles being successfully applied to battles with Stalker units. We believe that these results show promise for the future of RL in RTS games, by allowing us to train policies in smaller, less complex scenarios, and then apply those policies to different areas of the game, reducing the need for longer training times and much larger networks, like those found in AlphaGo.

Future work for this project can include implementing a similar network architecture and action component system using policy gradient RL methods to compare them to component-action DQN. Also, testing in scenarios that require using more action types to achieve high scores may help to better explore contributions of the component-action method, which may improve transfer learning performance.

References

Churchill, D.; Lin, Z.; and Synnaeve, G. 2017. An analysis of model-based heuristic search techniques for StarCraft combat scenarios. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Churchill, D. 2016. *Heuristic Search Techniques for Real-Time Strategy Games*. Ph.D. Dissertation, University of Alberta.

Heinermann, A. 2013. Broodwar API. https://github.com/bwapi/bwapi.

Huang, S., and Ontañón, S. 2019. Comparing observation and action representations for deep reinforcement learning in MicroRTS. *arXiv preprint arXiv:1910.12134*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In Balcan, M. F., and Weinberger, K. Q., eds., *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, 1928–1937. New York, New York, USA: PMLR.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2015. RTS AI problems and techniques. In Lee, N., ed., *Encyclopedia of Computer Graphics and Games*. Cham: Springer International Publishing. 1–12.

Pang, Z.-J.; Liu, R.-Z.; Meng, Z.-Y.; Zhang, Y.; Yu, Y.; and Lu, T. 2019. On reinforcement learning for full-length game of StarCraft. In *Thirty-Third AAAI Conference on Artificial Intelligence (AAAI-19)*, volume 33, 4691–4698.

Samvelyan, M.; Rashid, T.; de Witt, C. S.; Farquhar, G.; Nardelli, N.; Rudner, T. G. J.; Hung, C.-M.; Torr, P. H. S.; Foerster, J.; and Whiteson, S. 2019. The StarCraft multi-agent challenge. *CoRR* abs/1902.04043.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2016. Prioritized experience replay. In Bengio, Y., and LeCun, Y., eds., *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

Sun, P.; Sun, X.; Han, L.; Xiong, J.; Wang, Q.; Li, B.; Zheng, Y.; Liu, J.; Liu, Y.; Liu, H.; and Zhang, T. 2018. TStarBots: Defeating the cheating level builtin AI in StarCraft II in the full game. *arXiv preprint arXiv:1809.07193*.

Synnaeve, G.; Nardelli, N.; Auvolat, A.; Chintala, S.; Lacroix, T.; Lin, Z.; Richoux, F.; and Usunier, N. 2016. TorchCraft: A library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*.

Tang, Z.; Zhao, D.; Zhu, Y.; and Guo, P. 2018. Reinforcement learning for build-order production in StarCraft II. In *2018 Eighth International Conference on Information Science and Technology (ICIST)*, 153–158.

Tavakoli, A.; Pardo, F.; and Kormushev, P. 2018. Action branching architectures for deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 4131–4138.

Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double Q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Vinyals, O.; Ewalds, T.; Bartunov, S.; Georgiev, P.; Vezhnevets, A. S.; Yeo, M.; Makhzani, A.; Küttler, H.; Agapiou, J.; Schrittwieser, J.; et al. 2017. Starcraft II: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*.

Vinyals, O.; Babuschkin, I.; Chung, J.; Mathieu, M.; Jaderberg, M.; Czarnecki, W.; Dudzik, A.; Huang, A.; Georgiev, P.; Powell, R.; Ewalds, T.; Horgan, D.; Kroiss, M.; Danihelka, I.; Agapiou, J.; Oh, J.; Dalibard, V.; Choi, D.; Sifre, L.; Sulsky, Y.; Vezhnevets, S.; Molloy, J.; Cai, T.; Budden, D.; Paine, T.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Pohlen, T.; Yogatama, D.; Cohen, J.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Apps, C.; Kavukcuoglu, K.; Hassabis, D.; and Silver, D. 2019a. AlphaStar: Mastering the real-time strategy game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019b. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782):350–354.

Wang, Z.; Schaul, T.; Hessel, M.; Van Hasselt, H.; Lanctot, M.; and De Freitas, N. 2015. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*.