Reinforcement Learning for Building *StarCraft 2* Agents

Andrei Dumitrescu
University Politehnica of Bucharest
313 Splaiul Independetei, Bucharest, Romania
andreidumitrescu99@yahoo.com

Traian Rebedea
University Politehnica of Bucharest
313 Splaiul Independetei, Bucharest, Romania
traian.rebedea@cs.pub.ro

ABSTRACT
StarCraft 2 is a real-time strategy game, where two or more players compete against each other by gathering resources, building a base and an army. This game represents a huge challenge for the Reinforcement Learning domain, as it is a really complex game presenting a large number of possible states and actions. In this paper we will present a possible solution to solve the entire game of *StarCraft 2*, by experimenting on different mini games offered by the game's API. These mini games are specially designed for *Reinforcement Learning* experiments as they are representative of a concrete set of skills that an agent has to learn in order to master the entire game of *StarCraft 2.*

Author Keywords
Reinforcement Learning; Intelligent Agents; StarCraft 2; Advance Actor Critic; Convolutional Neural Networks

ACM Classification Keywords
I.2.6: Artificial Intelligence: Learning

DOI: 10.37789/rochi.2022.1.1.23

INTRODUCTION

Context
In recent years, the advancement of technology made possible the development and training of complex *Machine Learning* architectures, with hundreds of millions of trainable parameters. Nowadays, graphics cards, which are really useful in training *Deep Learning* models due to the computational power offered by them, begin to be more and more accessible to everyone. This evolution of technology also made a great impact on *Reinforcement Learning (RL)*, a subdomain of *Machine Learning.*

In *Reinforcement Learning*, an agent is trained to perform a task, by offering to it a reward depending on the actions made by the agent. Usually, in this domain different agents are trained for solving different tasks in games, or to learn how to play the entire game. The main subject of this paper revolves around the *StarCraft 2* game. *StarCraft 2* is a more complex game than the other ones as it offers very large action and state spaces, making the training process more complicated and the convergence of models more difficult. *StarCraft 2* is a classic real-time strategy game, in which two or more players compete against each other in a war type of situation. The players have to gather resources, to build a base and to recruit an army in order to destroy the opponent player's base. The game offers a lot of challenges that the player needs to overcome in order to improve his skill, such as: resource management and collection, correct base placement, correct army composition or scouting the enemy territory. There are multiple research papers which have tried to build an intelligent agent for this game, such as: Modular Architecture for StarCraft2 with Deep Reinforcement Learning [3], The StarCraft Multi-Agent Challenge [6] or On Reinforcement Learning for Full-Length Game of StarCraft [5].

Developing an intelligent agent for this game has a great impact in multiple areas. First, *StarCraft 2* is a game which is played at professional level for which there are held multiple international tournaments. Therefore, having an intelligent bot would help the professional players to train for these kinds of competitions. Second, due to its complexity the process of building such bots will lead to the discovery of new innovations in the *Reinforcement Learning* domain.

Problem
The main problem that is tackled by our paper is building multiple intelligent agents that try to learn and solve the mini games offered by the *StarCraft2* API. The game API offers 7 different mini games on which researchers can experiment with their agents. These mini games are designed to isolate different tasks that an agent must learn in order to fully master the *StarCraft2* game. We succeeded in training different agents for 4 different mini games out of the 7 ones. In the following section we will present the logic behind each mini game, the algorithms that are used in order to build the solution and the general idea behind the solution. Next, we will present the results obtained by the agents on each of the mini games and a comparison between different variations of the architecture behind the experiments.

RELATED WORK
In this chapter we will present two similar state-of-the-art algorithms that are widely used in *Reinforcement Learning*. These algorithms are Advantage Actor Critic [7] and Asynchronous Advantage Actor Critic [4]. These two algorithms represent an important key point in understanding the current proposed solution for training the agents. Both algorithms are designed to be used in *Deep*

Reinforcement Learning and their goal is to help the agents to learn the parameters of deep neural networks policies.

Both algorithms are from the class of *Actor-Critic methods.* These kinds of methods are characterized by two components: an *actor* and a *critic*. The *actor* component tries to learn what actions should be taken in different states, and the *critic* component "criticizes" the *actor* component by telling how good the chosen action was. Both components can be represented by two different *Neural Networks.* Usually, the *actor network* receives as input a state and outputs a probability distribution over the possible actions that could be applied in the respective input state. The *critic network* receives as an input a state and returns the *Q-value* of the given state. In order to understand how the weights of the network are updated, first we have to introduce the concept of an *Advantage Value*, from which also comes the name of the method. The *Advantage Value*, or also called the *Temporal Difference Error* is described by the Equation (1).

$$A(s,a) = R(s,a) + V_{\pi}(s') - V_{\pi}(s) \quad \textbf{(1)}$$

In *Equation 1*, *s* represents the current state, *a* represents the selected action, *s'* is the state that results after applying the respective action, the *R* function represents the reward, and lastly, V_{π} represents the *value function* corresponding to a concrete policy, which is approximated by the *critic network.* A positive value for the *Advantage function* tells the actor network that the selected action that is a good one and should exploit it more. In a similar way, a negative value for the function tells the actor network that the action is not so rewarding.

Asynchronous Advantage Actor Critic

Asynchronous Advantage Actor Critic [4], or *A3C* for short, is a training method different from the classical *Q-Learning* based ones. This model introduces the concept of asynchronous actor-learners. The main idea behind this algorithm is that it will start multiple agents in parallel which will try to optimize the training parameters of the used networks. The parallel agents are spawned on the same machine using multiple CPU threads at a time. Using multiple agents assures us that most probably they will try different strategies from one another and also that they will explore differently the environment in which the training process takes place. Moreover, most of the training methods from *Deep Reinforcement Learning* domain rely on a replay memory buffer.

The idea behind this buffer is that consecutive game steps encode very similar information, so when training an agent, it will be inclined to fall in a local minimum that tries to maximize that concrete subset of samples. In order to avoid this problem, a memory replay buffer is held where multiple game steps are stored by the agent which continues to play. When we want to perform an update step to the network, we shuffle the replay memory buffer and extract mini batches from it with which we train the agent. This way, the samples won't be from the same time frame. Using the *Asynchronous Advantage Actor Critic*, we don't need to use a replay buffer anymore. Using multiple agents in parallel which update the same network assures us that they will explore differently the environment and that the network will receive updates from multiple perspectives. The "asynchronous" part from the method's name comes from the fact that the agents don't wait for each other to finish to update the network. This can create multiple problems as different agents may use different versions of the network, depending on when they have updated it lastly.

Advantage Actor Critic

Advantage Actor Critic [7] is a new method, similar to *A3C*. *Advantage Actor Critic*, or *A2C* for short, is also a method from the *Actor Critic* family. The main difference between this method and the previous one is that it is not asynchronous. This means that the main process waits for all the agents to finish their episodes before updating the principal network. After updating it, the network is broadcasted back to the parallel agents which engage in other episodes. As an advantage, the synchronous behavior assures a better GPU utilization, as it can perform updates in batches.

StarCraft2 Learning Environment

The first step in starting our project was to research what existing APIs are available online that allow us to build a RL agent for the *StarCraft 2* game. We discovered that the best API to use is the *StarCraft 2 Learning Environment (SC2LE)* [8], specially build by researchers from *Google DeepMind* and *Blizzard*, the company that produces and distributes the game. *SC2LE* is an environment designed for building *Machine Learning* bots for the game, which exposes three main components: a Linux StarCraft 2 binary, StarCraft 2 API[1] (a tool that allows the user to control the game and perform actions such as starting games or analyzing replays), and *PySC2*[2], which is an open-source Python library that enables the user to build Reinforcement Learning bots and connect them to the StarCraft 2 API.

The environment also provides us with different mini games that would help an agent to learn different basic things such as: moving units to specified locations, collecting resources, and combating enemy units. Moreover, the original paper that presents the environment also describes some RL approaches on how to solve the tasks presented by the mini games. The presented approaches use a deep neural network, either Atari-Net or a

[1] GitHub - Blizzard/s2client-proto: StarCraft II Client - protocol definitions used to communicate with StarCraft II. Last accessed at: July 07, 2022

[2] GitHub - deepmind/pysc2: StarCraft II Learning Environment Last accessed at: July 07, 2022

Fully Convolutional Network and the Asynchronous Advantage Actor-Critic to learn a policy for the agent.

PROPOSED METHOD

In the following chapter, we will describe in full details the experiments. The main idea behind them was to replicate the experiments regarding the mini games proposed in the *StarCraft2 Learning Environment* paper [8]. The experiments consist in training multiple agents using the *Advantage Actor Critic* method and using the same neural network architecture on different mini games. Firstly, we will describe the mini games used for the experiments, then we will present in detail the official proposed solution and finally different variations with which we experimented and the logic behind them.

Mini Games Description

Each mini game was designed with a unique task in mind that an agent has to solve. The tasks gradually get more complex in difficulty and represent different skills that a final generic agent should master in order to play efficiently the game. Each of them has a specific reward system and all of them have a fixed time limit in which the agent has to achieve the goal. The time limit imposed to each mini game defines how much an episode lasts for each of them. In total there are 7 mini games, but we only ran experiments on 4 of them.

Move To Beacon

MoveToBeacon is the easiest mini game of them all. The environment consists out of a single marine unit which can be selected by the player and moved on the map. On the map there also spawns a beacon. This beacon signals the position on the map where the marine unit should reach. This mini game is designed for the agent to learn how to select a unit and move it on the map to a specific location. This time the map is perfectly fitted in the view screen, so the agent does not have to learn how to move the camera.

Also, there is no fog-of-war activated. This means that the agent can see everything on the map, not just the environment in the view range of its units. The reward system consists of giving a *+1* reward to the agent when it moves the marine to the correct location. When a beacon is reached it disappears and another one is spawned at a new random location on the map. The time limit for this mini game is 2 minutes. This mini game can also be seen as a unit test for learning agents. Usually, if an agent cannot learn this task there is no reason in moving to another mini game, as they all are more difficult than this one.

Collect Mineral Shards

CollectMineralShards is a more difficult mini game than *MoveToBeacon* but it expands on the same idea. This time the agent starts with 2 marine units. On the map there are spread out multiple mineral shards. The agent needs to learn how to select units and how to move them, but also it has to master a coordination in movement. For each mineral shard collected from the map the agent receives a +1 reward. When all the mineral shards are collected new ones spawn on the map. Similarly, the map is perfectly fitted in the view screen and no fog-of-war is activated. The time limit for this mini game is also 2 minutes.

Find and Defeat Zerglings

FindAndDefeatZerglings is a more complex game than the previous ones and probably is the most difficult mini game out of all the ones on which we experimented. The agent starts with 3 marine units placed at the center of the map. On the rest of the map, there are placed multiple enemy units called zerglings. The zerglings are placed in such a way that they don't form a group. The goal for the agent is to learn to find, identify, and defeat enemy units. For an agent to fully master this mini game, it has to learn the following skills: to select and move a friendly unit, to identify an enemy unit, to use its units to defeat the enemy unit, and to explore the surrounding environment. This time the fog-of-war is activated.

The reward system is more complex, as the agent can receive a *-1* negative reward if it loses an unit to the enemy. For each enemy unit defeated the agent receives a *+1* reward. The time limit is set to 3 minutes. Also, the map is bigger than the actual camera view, so the agent is encouraged to also learn on how to properly move it.

Defeat Roaches

DefeatRoaches is also a combat centered mini game. The agent starts with 9 marine units and must defeat an opposing group of 4 roaches, which are a more powerful type of enemy units than the zerglings. Every time the agent defeats all the 4 roaches, it receives an additional 5 marines as reinforcement units and 4 other roaches spawn on the map. This mini game does not involve exploration, as the roaches are placed right in front of the marines. The fog-of-war setting is not enabled, and the map is perfectly fitted in the camera view. The reward system offers a *+10* reward for each enemy unit destroyed and a *-1* reward for each friendly unit lost. This mini game could be considered a bit easier than the previous one, as the agent only has to learn how to combat the roaches and not fully move on the map or explore the environment. The time limit for this mini game is 2 minutes.

Proposed Solution

As mentioned previously, the experiments try to replicate the ones described in the original paper that introduced *PySC2*, so next we will present how the original solution works [8].

The first important step in building the solution is to understand how the actions are represented in the *PySC2 API*. An action is formally defined by an unique function identifier and a sequence of parameters which are needed for the function to run properly. As a simplified example, if

we would want to select a rectangle with the mouse we would have to use *select_rect* function identifier and pass two pairs of coordinates (x_1, y_1) and (x_2, y_2). In this case, the coordinates represent the top-left corner and the bottom-right corner of the rectangle. Using this representation, the action space contains around 300 unique function identifiers which can receive up to 13 different types of arguments. Each individual action-function has a fixed defined number of parameters and there are also actions which don't require any type of argument.

The proposed solution uses a simple neural network architecture based on convolutional neural networks. The algorithm used to optimize the parameters of the policy is the *Asynchronous Advantage Actor Critic* with a custom defined gradient. It would require a large number of parameters to represent the policy in the naive way, so the Vinyals et al. [8] proposed a simplified version, described in Equation (2).

$$\pi(a|s) = \prod_{i=0}^{L} \pi(a^i|s) \quad \textbf{(2)}$$

In this equation, L is the number of total parameters and a^0 represents the function identifier. More intuitively, the equation states that the function identifier and its parameters are chosen independently from one another depending on the state in which the agent is. The policy will be used to firstly predict what function identifier to use and then independently will also predict its parameters, if any are required. As another simplification, the actions that are unavailable in a given state are masked out. This is achieved by masking out the respective function identifiers and by normalizing the probability distribution over the identifiers.

The proposed network architecture employs a *Convolutional Neural Network (CNN)* as shown next in Figure 1.

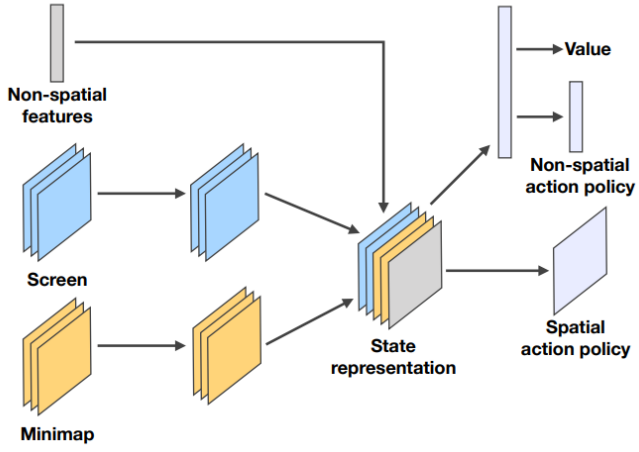

Figure 1 – Fully Conv Architecture [8]

The architecture was named *Fully Conv*. This agent is proposed as a baseline and the main idea behind using a CNN was that they help in building a more abstract representation of spatial features. This property helps a lot in the context of *StarCraft 2* where most of the actions are related to spatial features. The network receives 3 different types of inputs: the screen of the game, different types of mini maps offered by the *PySC2 API*, and all non-spatial numerical features such as: number of units, number of resources, etc.

All the discrete and numerical features are pre-processed, being rescaled with a logarithmic function, and mapped to a continuous space using a 1x1 convolution. The screen and mini map features are both passed each to a different pair of 2 *Convolutional Layers*. The structure of the layers is the same for both the screen features and the mini map features. The first layer uses 16 filters and a kernel size of 5x5. The second layer uses 32 filters and a kernel size of 3x3. All the layers don't have a stride and use padding.

In this way the original resolution of the features is maintained throughout the network. After encoding the 3 types of inputs they are concatenated in a single representation. To achieve this, the numerical features are broadcasted to the correct shape. Finally, this representation is used to define 2 policies, one over spatial actions and one over non-spatial actions. The former uses a 1x1 convolution with 1 filter over the state representation. The later initially flattens the state representation and passes it through 2 Linear Layers. The first Linear Layer has 256 output units and a ReLU activation function.

The input in the case of the screens / mini maps will be the following: *(Number of frames, Height, Width, Channels)*, where the "*Number of frames"* dimension acts like a batch size. In the original experiments, the researchers propose to use a 64x64 resolution and act at an 8 game frames rate. This baseline is reimplemented as an open-source *API*[3] which we also used in our experiments. The only difference is that they use the *Advantage Actor Critic* method and not the asynchronous variant.

Architecture Variations

Beside replicating the experiments described above, we also tried small variations of the architecture presented above that in theory could improve the behavior and the convergence of the agent.

The first obvious idea is to increase the *Receptive Field* of the CNNs that are used to embed the features of the screen and mini map inputs. In Deep Learning, the Receptive Field defines the size of the region in the input that is used to compute a single feature. In theory, the bigger the

[3] GitHub - simonmeister/pysc2-rl-agents: StarCraft II / PySC2 Deep Reinforcement Learning Agents (A2C) Last accessed at: July 07, 2022

Receptive Field the more information embeds an output feature from the networks' input. Using more information should produce more qualitative representations. More concretely, in the context of a Convolutional Layer the Receptive Field defines the size of the pixels patch that influences an outputted encoded pixel. To measure the Receptive Field, we can use Equation (3).

$$r = \sum_{l=1}^{L} \left((k_l - 1) \prod_{i=1}^{l-1} s_i \right) + 1 \quad \textbf{(3)}$$

In the previous equation, *L* represents the total number of layers, the *k* parameters the kernel size of each layer, and *s* parameters the strides of each layer. Calculating the Receptive Field for the given network we obtain a value of 7, which means that a total of 49 pixels contribute to each output pixel. There are multiple methods to increase the Receptive Field, such as adding a new convolutional layer, adding a pooling layer, or using depth-wise convolution layers. Our idea was to add another convolution layer before the other ones which has a kernel size of 7x7, 8 filters and no stride. By doing this, we increase the Receptive Field to 15, which means that 125 pixels are now used to encode information.

The second proposed variation was to exploit the temporal information that the fixed number of game frames offers to us. More specifically, the architecture proposed in the paper uses 2D convolutions, which means that the frames are treated independently one from the other, as a batch. In practice, the frames encode temporal information that is not exploited by the agent. To capture it, we decided to use a 3D convolution layer which is specifically designed to apply filters to such kind of inputs [2]. In theory, the input of such a convolution is of shape *(Batch Size, Frames, Width, Height, Channels).* This new layer is added after the two convolution layers in the original architecture, and we need to transform the input to fit the size of the new convolution. To achieve this, we reshape the output of the last 2D convolution from *(Frames, Width, Height, Channels)* to *(1, Frames, Width, Height, Channels).* The reshaped representation is passed to the 3D convolution and after that is reshaped back to its original form. The hyperparameters set for this new layer are: 32 filters, 3x3 kernel size, and no stride. Moreover, this approach also exploits the idea described above of increasing the Receptive Field.

Lastly, we experiment with adding "*memory*" to the network. In the original paper [8], they try a similar approach by introducing a *Convolutional LSTM Layer* after obtaining the concatenated representation. Similarly, we add a *GRU Cell* [1] after the concatenated representation. As time steps, we decided to use the channels dimension of the representation. The representation was initially flattened and passed to the *GRU Cell.* After obtaining the output we reshaped it to the initial dimension.

Heuristic Based Agent

Additionally, we propose a hand-crafted agent for the *Collect Mineral Shards* mini game which does not use any type of *Machine Learning* methods. It is designed to select one marine at random and use it to collect the minerals. The marine selects at random a mineral shard placed on the map and goes straight to collect it. It may happen that on the way to the selected shard for the marine to collect accidentally more shards. The other unselected marine will remain unused during the running of the game. After collecting all the shard from a batch, the agent will not stop and will continue to apply the same algorithm to collect even more points. We chose the *Collect Mineral Shards* mini game because it is more complex than the easiest one and also has an obvious heuristic that could be used for the agents. It is important to note that this hand-crafted agent uses not only rules, but also information about the current game – such as position of mineral shards. In comparison, the RL agents do not use any such information, they need to learn to infer it from the frames of the game.

EXPERIMENTS

Experiments' Setup

Vinyals et al. [8] present the values for all hyper-parameters in order to reproduce their results. However, it is not easy to use all the mentioned settings on a commercial computer. The resolution proposed for representing the mini maps and the screen is 64x64. In their experiments, they use 64 parallel asynchronous actor critic agents which act at every 8 game steps. This number of game steps was chosen to limit the number of actions per minute of the agent to 180, which is equivalent to the performance of a good StarCraft player. The results reported in the original paper were the best results obtained after running the experiments with different hyper-parameters for ~100 times.

The computer used for running our experiments has the following setup: Nvidia 3080 GPU, 32GB RAM, and AMD Ryzen 9 3900XT 12-Core Processor. The processor has enabled hyper-threading, meaning that in theory it could run in parallel 24 processes at a time. Taking these specifications into consideration, the best set of hyper-parameters that we found is the following: 32x32 resolution and a maximum of 10 parallel actor critic agents which act also at every 8 game steps. Even if in practice we could run with 10 parallel actor critic agents, it didn't behave reliably. This means that the training process would result in a memory error due to having too much StarCraft2 clients started at the same time. Using just 8 parallel actor critic agents performed much better, as it didn't result in such an error.

For each mini game all the experiments were trained using approximately the same number of episodes. Except the easiest mini game, the experiments on all the mini games were run for around 3000-4000 episodes. For the easiest

mini game all the agents were trained until reaching the maximum attainable reward.

Results

During the experiments the first step was to make sure that each model variation could converge on the easiest game: *Move To Beacon.* All the models could converge on this game, except the variation that uses *GRU Cells.* Due to this fact, the following tables and results won't include any statistics about this variation. The reason why we think that this experiment failed is due to the fact that the channels are used as time steps for the *Recurrent Neural Network.* The intuition is that they don't encode enough important information about the state representation. Another possible problem is that we apply this recurrent cell on the concatenated input features which might not correlate so well as some of them are not spatial features.

For the rest of the models, we experimented with each of them on all the four described mini games. We decided to plot some graphs that show that the models succeed in converging to a concrete reward value. The graphs are plotted for *Find And Defeat Zerglings*, as we considered it a more challenging game than the basic mini game. In *Figure 2* it is plotted the evolution of the received reward during the training process of the model that uses a 3D convolution.

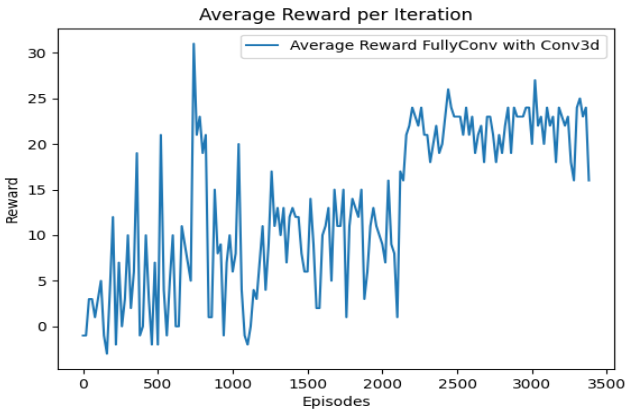

Figure 2 – Reward gained during the training process by the Fully Conv with a 3D Convolution

The x-axis represents the total number of episodes on which the agent was trained on. The y-axis represents the reward collected by the agent in an episode. Additionally, in Figure 3 we present a comparison between the convergence of all 3 different models on the *Find And Defeat Zerglings* mini game.

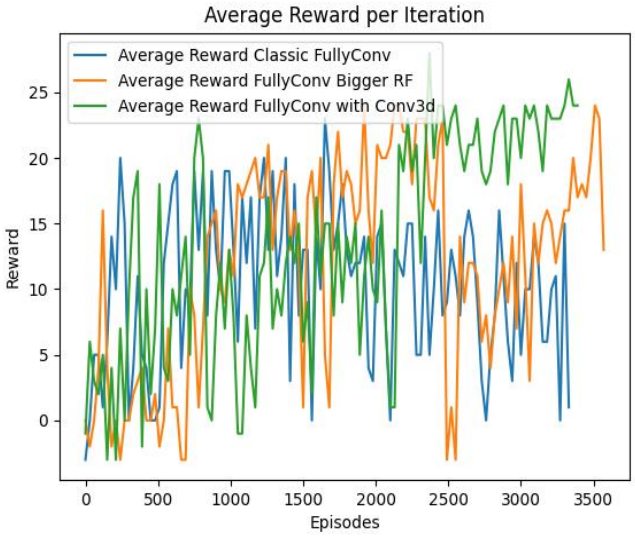

Figure 3 – Convergence comparison between all the 3 different model variations

The axes in Figure 3 have the same meaning as the ones in Figure 2. As it can be seen, for this specific mini game the network with a 3D convolution layer converges to a better reward value. To better compare the performance of the variations, we computed for each mini game a reward average over 100 episodes. Additionally, we computed the standard deviation of the rewards during these 100 episodes and also the maximum and minimum obtained rewards. All these values are presented in Tables 1 - 4. The tables also include for reference the best results obtained by the DeepMind team in their paper [8]. The bolded values represent the best values obtained by one of our agents for the respective mini game.

	Average Reward	***STD***	***Max Reward***	***Min Reward***
FullyConv (Ours)	*24.22*	*2.20*	*29.0*	*15.0*
FullyConv with bigger RF	***25.28***	*1.93*	*29.0*	*21.0*
FullyConv with 3D convolution	*25.20*	*2.48*	*30.0*	*15.0*
FullyConv (DeepMind [8])	*26.00*	-	*45.0*	-

Table 1 – Move To Beacon Mini Game Comparison

	Average Reward	***STD***	***Max Reward***	***Min Reward***
FullyConv (Ours)	***61.50***	*9.07*	*84.0*	*35.0*
FullyConv with bigger RF	*15.94*	*2.39*	*29.0*	*10.0*
FullyConv with 3D convolution	*26.05*	*7.75*	*39.0*	*17.0*
Heuristic Based Agent	*54.51*	*5.76*	*71.0*	*43.0*
FullyConv (DeepMind [8])	*103.00*	-	*134.0*	-

Table 2 – Collect Mineral Shards Mini Game Comparison

	Average Reward	***STD***	***Max Reward***	***Min Reward***
FullyConv (Ours)	*7.55*	*2.43*	*14.0*	*2.0*
FullyConv with bigger RF	*18.32*	*5.08*	*29.0*	*5.0*
FullyConv with 3D convolution	***22.06***	*3.81*	*40.0*	*6.0*
FullyConv (DeepMind [8])	*45.00*	-	*56.0*	-

Table 3 – Find And Defeat Zerglings Mini Game Comparison

	Average Reward	***STD***	***Max Reward***	***Min Reward***
FullyConv (Ours)	***23.20***	*18.55*	*121.0*	*-9.0*
FullyConv with bigger RF	*16.53*	*18.47*	*71.0*	*-9.0*
FullyConv with 3D convolution	*20.50*	*17.56*	*81.0*	*-9.0*
FullyConv (DeepMind [8])	*100.0*	-	*355.0*	-

Table 4 – Defeat Roaches Mini Game Comparison

As it can be seen from the tables, none of the experiments compare to the results obtained by the Deep Mind team, but they are still good. The only exception is the *Move To Beacon* mini game, where all the variations converged to similar values. As it can be seen from the *Table 2*, the *Fully Conv* agent trained by us performs better than the "hand crafted" one.

The tables also show that the variations perform similarly for the *Move To Beacon* and *Defeat Roaches* mini games. For the former one, both proposed variations perform slightly better than the classical *Fully Conv* one. The 2 mini games where there are more notable differences are the *Collect Mineral Shards* mini game and *Find And Defeat Zerglings.* For the first mentioned one, all the agents learn to collect a consistent number of shards, but the classical *Fully Conv* agent learns a better policy overall by a big margin. This can be explained by the fact that all 3 models where trained on approximately the same number of epochs, but the classic *Fully Conv* model has far less parameters than the other 2 models. Due to this property this model should converge faster than the other 2 ones.

An interesting observation can be made for the *Find And Defeat Zerglings*, where the classical architecture doesn't perform so well as the other 2 different variations, even if it has an advantage in convergence speed. Moreover, the variation that exploits both temporal information and a bigger *Receptive Field* generally performs better than the other simpler variation.

CONCLUSIONS

In conclusion, in this paper we showed that by replacing the *Asynchronous Actor Critic* algorithm with the *Advance Actor Critic* one in the proposed solution by DeepMind [8] the resulting agent still performs well on the tested mini games. Moreover, the experiments showcased in this paper are obtained using a lot less computational power than the ones performed in the original solution, proving that they can be replicated to some degree even on some more accessible computers. Lastly, the variations proposed to the architecture present a great potential on improving results on some more of the more complex tasks from the StarCraft game. This statement was empirically proven by the results obtained on the *Find and Defeat Zerglings* mini game. In the future, we would like to expand these solutions on the other 3 mini games that weren't described in this research report and on the early game phase of the game.

REFERENCES

1. Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.
2. Ji, S., Xu, W., Yang, M., & Yu, K. (2012). 3D convolutional neural networks for human action recognition. IEEE transactions on pattern analysis and machine intelligence, 35(1), 221-231.
3. Lee, D., Tang, H., Zhang, J. O., Xu, H., Darrell, T., & Abbeel, P. (2018). Modular architecture for starcraft ii with deep reinforcement learning. Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference.

4. Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. International conference on machine learning, 1928-1937.
5. Pang, Z. J., Liu, R. Z., Meng, Z. Y., Zhang, Y., Yu, Y., & Lu, T. (2019). On reinforcement learning for full-length game of starcraft. Proceedings of the AAAI Conference on Artificial Intelligence.
6. Samvelyan, M., Rashid, T., De Witt, C. S., Farquhar, G., Nardelli, N., Rudner, T. G., & Whiteson, S. (2019). The starcraft multi-agent challenge. arXiv preprint arXiv:1902.04043
7. Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction. MIT press.
8. Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Vezhnevets, A. S., Yeo, M., & Tsing, R. (2017). Starcraft ii: A new challenge for reinforcement learning. arXiv preprint arXiv:1708.04782.