Transfer of Knowledge in a Reinforcement Learning Setting for a Complex Environment

Progressive Networks in StarCraft II

8. of June 2018

Group DEIS1022F18
Aalborg University

AALBORG UNIVERSITY

STUDENT REPORT

Institut for Datalogi
Selma Lagerløfs Vej 300
9220 Aalborg Ø
Telefon (+45) 9940 9940
Fax (+45) 9940 9798
http://cs.aau.dk

Title:
PLACEHOLDER

Theme:
Specialization

Project Period:
Spring 2018

Project Group:
DEIS1022F18

Participant(s):
Andi Rosengreen Kjærsig Aaes
Kaare Bak Toxværd Madsen
Malthe Dahl Jensen

Supervisor(s):
Manfred Jaeger

Copies: 3

Page Numbers: 75

Date of Completion:
June 4th, 2018

Source Code:
Delivered separately as .zip file
(Private Repositories)

Abstract:

This project is a master thesis by a group on the 10th semester of the software education at Aalborg University.
The topic of this project surrounds using reinforcement- and transfer-learning on the complex environment of Starcraft II. We test a number of different agent architectures to find a candidate best suited for applying transfer learning.
To test if transfer is possible on Starcraft II, we use a network architecture proposed by Google DeepMind in 2016 called progressive networks[1], which allows us to leverage knowledge from multiple tasks when training on new tasks. At the same time progressive networks do not suffer from catastrophic forgetting, which allows us to approximate how much transfer is happening and where in the network it is occurring.

Signatures

Andi Rosengreen Kjærsig Aaes

Kaare Bak Toxværd Madsen

Malthe Dahl Jensen

Summary

The purpose of this project is to investigate if it is possible to utilize transfer learning from one task to another in a complex environment such as Starcraft II.

The report starts by investigating multiple different agent architectures to find the one best suited for grasping the complex environments, and thereby find the agent architecture with the best chance of learning features that can be transferred across environments. These agent architectures includes a standard A3C agent, a number of different agents that include some kind of memory encoding, and a modified version of the A3C agent that was introduced in our previous semester report[2] which we named SA3C that utilizes multiple location networks. Based on findings gathered by testing these agents against each other, we discovered that none of the agents were a significant improvement to the regular A3C, but were often more time and resource consuming. Because of this we decided to move forward with the regular A3C agent.

For testing transfer learning on the game of Starcraft II, we focus on a network architecture introduced by Google Deepmind in 2016 called progressive networks[1]. This network architecture makes it possible to leverage knowledge from an arbitrary number of source tasks when training on a new target task. At the same time progressive networks do not suffer from catastrophic forgetting, which allows us to calculate an approximation of how much the transferred knowledge is being leveraged for the target task, and make us able to investigate the details of and conclude on the tests.

We first test our implementation of progressive networks on two less complex environments namely CartPole[3] and Sonic the Hedgehog[4]. We do this in order to determine if our implementation of progressive networks is functional, before testing it on Starcraft II. Our findings in these tests shows us that we can transfer knowledge from fully-connected and convolutional layers on these low complexity games, even between different tasks.

After we concluded that our progressive networks are functional, we performed tests on Starcraft II to determine if transfer of knowledge is possible, and beneficial. We tested the progressive networks on three different minigames, leveraging knowledge from 1 and 2 minigames at a time. We found that it is indeed possible to leverage knowledge from one Starcraft II minigame to another, but that it was not always beneficial for the agent.

Some minigames allow for better transfer of knowledge between eachother, such as the FindAndDefeatZerglings and DefeatZerglingsAndBanelings minigames. Both minigames require the use of multiple friendly units, the ranged marine unit, to fight and destroy enemy units. Both minigames have the zergling unit as one of the enemy units, which is expressed as part of one of the input features for the screen. This seems to make a transfer learning agent able to leverage the constructed features from the convolutional layers.

We performed a proof of concept test on the DefeatRoaches minigame to verify that transfer was possible when the source task and target tasks are identical, but this test showed that previously learned knowledge is not always leveraged in the best case scenario.

We conclude that overall transfer learning in Starcraft II is possible when using progressive networks, and that more tests should be performed to further verify when it can happen, and if it reliably improves the convergence speed or average reward over time of the agent.

Preface

The Vancouver method is used for citations, where sources are indicated with a number in square brackets (i.e. [2]), and comma separation if using multiple sources. The title, author(s) and other relevant information is stated in the bibliography.

Several abbreviations and terms will be used throughout the report. The abbreviations are described on first time use in the report, but for good practice the most frequent abbreviations are stated here as well.

Abbreviations:

- Average Layer Sensitivity (ALS)
- Real-Time Strategy *game* (RTS)
- Application Programming Interface (API)
- Markov Decision Process (MDP)

Terms:

PySC2: A Python library for the Starcraft II API.

Agent: Software that can observe and actuate on an environment.

(Neural) Network: A set of neurons, ordered in layers, that has an input and output.

Episode: One game instance until player win, lose or time elapsed.

(Time) Step: One discrete observation of the environment.

Mini-game: An environment that is a subset of the full game, with one or multiple purposes based on the full game.

Contents

Chapter 1: Introduction

In this report we test if deep reinforcement learning, reinforcement learning using deep neural networks, benefits from transferring of knowledge gained from similar environments, to assist learning in problems with different goals, called transfer learning[5]. There are different ways of performing transfer learning, where Progressive networks and finetuning are two types of transfer learning that we focus on in the report.[1]

Transfer learning has been shown to improve agents when used in agents for solving various games for the Atari platform, as well as a labyrinth navigation task.[1]

These tests were made on environments with relatively few actions, in the low 10s, and require only simple strategies for completion of the task. We seek to determine if the discussed transfer learning types can be used in a more complex environment, where there are a vast amount of actions and where completing the tasks require relatively complex strategies.

Progressive networks in particular allows us to analyze how well previously learned knowledge for one task is being leveraged in another task, which makes us better equipped for discussing the impact of transfer learning in a complex environment.

With a more complex environment, the possibilities for knowledge transfer is more potent. Higher complexity in the tasks often translates into longer training periods and higher risk for sub-optimal policies stuck in local optimum. We investigate if the higher complexity of the environment, and the strategy to complete the task, impact the usage of knowledge. Previous study shows that transfer learning, progressive networks in particular, can utilize knowledge from tasks that have little to no similarities with the task at hand[1]. However in this report we will focus on determining if transfer learning is at all beneficial in a more complex environment than the relatively simple Atari games.

Google DeepMind and Blizzard Entertainment have made it possible to use a complex game based environment that fits our needs very well. It uses the computer game Starcraft II, with an API for coupling the game environment with a reinforcement learning agent using python. Starcraft II is a complex Real-Time Strategy (RTS) game, which can be split into sub-problems, called mini-games, these mini-games utilize the same actions and similar observable environments.[6]

The following sections seek to introduce the background necessary for understanding the work.

1.1 StarCraft II

Starcraft II is a RTS video game that has a large competitive player-base, where worldwide tournaments are held every year. The game is very hard to master and has a vast amount of strategies, game elements, and simultaneous management tasks, that a player needs to decide on and execute better than their opponent in order to win.

Starcraft II being a RTS game means that it does not have turns during which the player chooses which action to take, based on some game state. The game runs in real-time, and the player can choose to do an action at all times, thus a player might do better if he is faster at executing actions, but more actions per minute(APM) does not necessarily make a good player. In Starcraft II the

player plays against one or more players, and the point of the game is to destroy all units of the opponents in an environment with limited resources and space. At the start of a game, the player must build structures to make a base, then build armies to defend or attack the opponent. The only means to victory is through battle, so building units and structures for army production, resources and upgrades, fighting the opponent player with units, defensive structures, are all core elements of the game. Starcraft II is a very complex game, that requires multitasking, planning and execution of strategy, and micromanagement all under time pressure, as well as a vast knowledge of the purpose of all buildings, researches, and units, as some units, buildings, and upgrades can be essential for countering certain strategies of the opponent. Starcraft II has, as many other RTS games, a certain view-range or line of sight around all their units and buildings in the game, where the surroundings can be seen. This means the player cannot see what is going on in places where the player does not have units or buildings. Therefore the game is partially observable, making it harder for the player or agent to predict what the opponent is doing, and thus harder to counter the opponent's actions.

Starcraft II allows for creating and playing mini-games, where developers can create customized maps with many possibilities for changing units, game goals, visuals, game mechanics, and much more. In this report, we will be referring to mini-games as games that incorporate a smaller part of the regular player vs player game mode, but with less complexity. The mini-games used in this project will be similar to the normal player vs player game previously described, however we will change the goal of the game, to make a less complex environment. All mini-games will however be without the player vs player aspect of the game, but instead the goal will be to handle a specific aspect of the game as well as possible, which will be scored with rewards depending on the goal of the mini-game. In the normal player vs player game, the player has to make decisions early that can change the course of the whole game, and a lot of the strategies in the game are long term, and need to be maintained throughout the game. In the mini-games, the strategies are less complex and shorter lived then in the standard game. By comparison, the length of a standard player vs player game can often go beyond 30 minutes and require strategies for all aspects of the game, whereas the duration of the mini-games are between 5 and 15 minutes and require only strategies for few selected aspects of the game, such as combat, at once.

The combination of the hundreds of available actions, and the complexity of the decision making required for Starcraft II, makes the Starcraft II game a hard environment for even state-of-the-art reinforcement learning agents, and is therefore a very interesting environment for pushing the limits of reinforcement learning. Google DeepMind was unable to create a reinforcement learning agent that could play the full player vs player Starcraft II game[6], and as such decided to focus on the described mini-games.

Google DeepMind and Blizzard Entertainment have created a Python library for the Starcraft II API called PySC2, for connecting machine learning algorithms with the Starcraft II game environment. We will utilize this library throughout the project, to handle all interaction with the game environment. The library was created with a machine learning focus, so a few limitations on the observations has been made, so that working with PySC2 requires less computational power and helps agents converge faster. This is done by creating feature layers that extract information from RGB pixels with spatial information of the screen, instead of feeding raw pixel input to the agent. As an example, one feature layer represents units and their hp, and another feature layer describes the type of units. However this implementation also limits the agent's knowledge of the game, for example the agent has no way of telling what actions the units are doing based on a single set of feature layers, describing one frame of the game.

1.1.1 PySC2

The PySC2 library allows a reinforcement learning agent, created in python, to interact with the game of Starcraft II using an API. PySC2 is a library created with machine learning in mind, and the creators have constructed features to be used as representation for the state of the game.

The library currently only supports usage of the constructed features, but there are plans to allow for learning using the full RGB representation of the state. In this report we focus on learning with the features constructed by the library.

There are three sets of features, each representing a specific type of data currently observable in the current state by the player playing the game. These sets of features are *screen spatial features*, *minimap spatial features*, and *non-spatial features*. The screen and minimap spatial features represent the information available to the player through the primary screen, where the primary interactions happen, and the minimap where an overview of the complete game is available in low resolution. The game as seen through the library is displayed on Figure 1.1. The various screen and minimap spatial features are displayed on Figure 1.2 on the next page.

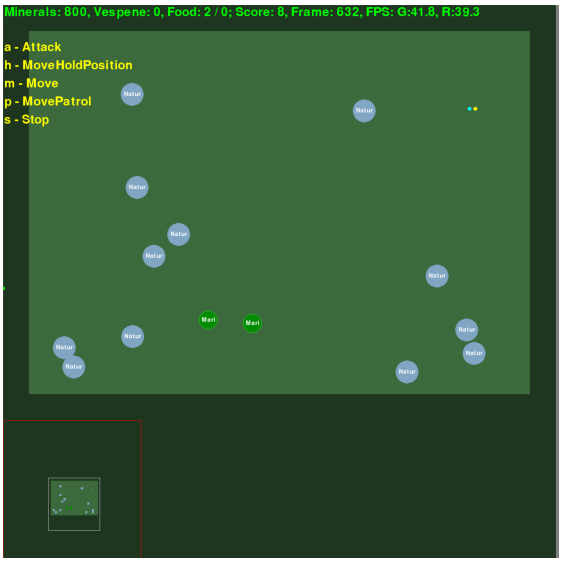


Figure 1.1: Screenshot of the screen in the middle, minimap at the lower left and some non-spatial features at the top

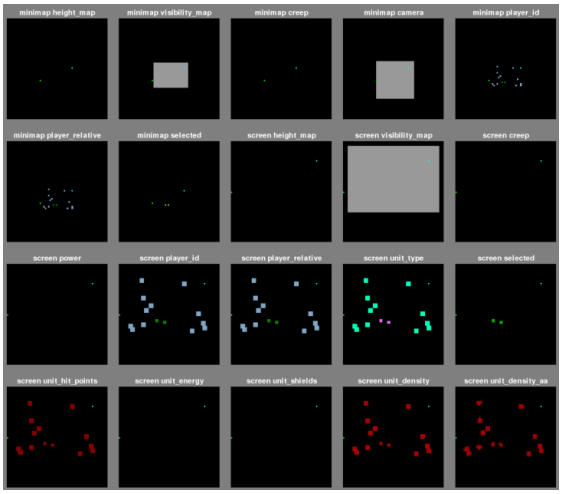


Figure 1.2: Screenshot of the 20 available spatial feature layers; 7 minimap layers and 13 screen layers

The non-spatial features represent data that is not represented visually, but is still relevant for a player, e.g. unit information and economy information.

Each of the features available are either categorical features, e.g. the *screen unit_type* feature layer that distinguishes different types of units, or scalar features, e.g. the *unit_hit_points* feature layer that gives a scalar value for the health in place of every unit present in the screen.

Some features may attain relatively large values due to the nature of the feature, such as the two previously described unit type and health features that contain values in the thousands. We perform preprocessing of features that may contain values in the hundreds or above, where we normalize them to be between 0 and 1. This normalization was done based on findings in the initial research paper for PySC2 by the deepmind team.[6]

All features are represented as arrays, where the size of the array depends on the data represented. Spatial features are 2D arrays of the same size as the screen, $64x64$ by default.

Through this PySC2 library we are able to retrieve the state of the game, represented by all the features, including all actions that can be executed. The library also handles interaction with the game through a step function, that allows for executing a single action in the current state. All features and methods available is described in the environment documentation.[7]

1.2 Deep Reinforcement Learning

Reinforcement learning is the act of placing an agent into an environment, allowing it to learn to act well in the environment in order to maximize some cumulative reward from or based on the environment. A reinforcement learning agent is not told which actions are good or bad, but is only given reward. In contrast, a supervised learning agent will learn based on labeled data that contains the correct answer.

We use reinforcement learning to grasp the complex environment of Starcraft II and the mini-games supplied with the Starcraft II Learning Environment[6]. The Starcraft II environment

makes a good platform for researching reinforcement learning, made more accessible by the team at Google DeepMind[6]. The DeepMind team mentioned that it is a very interesting domain, due to the complexity present in even simple mini-games that only contain few elements of the full game. We focus on deep reinforcement learning, where neural networks are used as function approximators.

When an agent is learning, it is presented with a choice between exploitation, exploiting what is known to make the best decision for optimal reward, or exploration, the act of forgoing immediate reward for gathering information that allows for potentially higher long-term reward.[8] Overfitting and underfitting in reinforcement learning is related to the exploration-exploitation trade-off. Overfitting occurs when an agent focusing on exploitation will become specialized in a subset of the full action-state space, and therefore will not be able to generalize well. Underfitting occurs when an agent is not exploring the action-state space well enough, limiting it to locally-optimal policies.[9] It is important to ensure that an agent is able to explore well in order to visit higher value states, without limiting the exploitation of the agent such that it may never be able to make the decisions that give higher reward in the newly explored state space.

A reinforcement learning problem being acted upon by a reinforcement learning agent, is described by a Markov Decision Process (MDP)[10, 11]. The Markov Decision Process is described in the next section, followed by a description of how an agent can learn the problem using various learning methods, in particular the actor-critic method, and the objective function for learning using an actor-critic method.

1.2.1 Markov Decision Process

A MDP satisfies the markovian property, meaning any decision to be made in the current state is only dependent on information available in the current state. The following contains a description of the elements of the MDP tuple:

- S - set of states for the environment
- A - set of actions for the agent
- $T(s'|\, s, a)$ - state transition probability function: $T(s'|\, s, a) = \mathbb{P}[S_{t+1} = s'|\, S_t = s, A_t = a]$
- $R(s)$ - reward function. Rewards are based on the current state s
- γ - discount factor for future rewards

Figure 1.3 contains a simple MDP. There are 11 different states with 3 terminal states that give reward. The set of actions for this MDP is $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$, and the reward is designated by the numbers in the colored states. A reward of 2 is received when the current state is the right-most state.

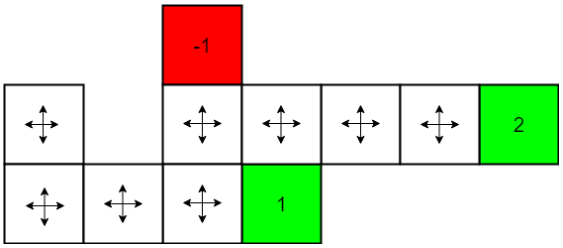


Figure 1.3: Example of a simple MDP with 11 states

For this example, the state transition probability function T will in this case have a probability of 1 for the direction of movement chosen, without some chance of moving in one of the other directions.

In order for an agent to act, it has to follow a policy $\pi(a|s)$, which is a probability distribution over actions given a state, as seen in Equation (1.1).

$$\pi(a|s) = \mathbb{P}[A_t = a \mid S_t = s] \tag{1.1}$$

The objective in reinforcement learning is to maximize the long-term future reward G_t seen in Equation (1.2), by finding and following the optimal policy π^* that always selects the action maximizing long-term future reward.

$$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \tag{1.2}$$

The example on Figure 1.4 shows the actions to be used in each state by following the optimal policy π^*. This example assumes a γ value of 1. The reward in this case is 2 when the terminal state with reward 2 is reached.

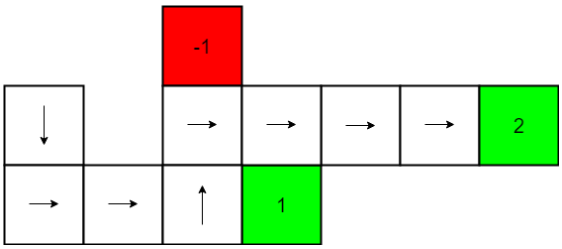


Figure 1.4: Example of applying an optimal policy to a MDP

There are 2 types of value functions that specify the expected long-term future reward; the state-value function $V_\pi(s)$, and the action-value function $Q_\pi(s, a)$. The state-value function of an MDP is the expected return from being in state s and following the policy π, as seen in Equation (1.3). The action-value function of an MDP is the expected return from being in state s, taking action a, and following the policy π, as seen in Equation (1.4).

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] \tag{1.3}$$

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \tag{1.4}$$

Each of these value functions can be converted to a Bellman equation, such that the succeeding states are decomposed into a single component, as seen in Equation (1.5) for the state-value function, and Equation (1.6) for the action-value function.

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s] \tag{1.5}$$

$$Q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma Q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \tag{1.6}$$

The discount factor γ is there to determine how much future rewards should be weighted. A high discount factor value means that all rewards are equal, and the highest reward should always be chosen. A low discount factor means early rewards are more important, and should be chosen over larger future rewards.

The example shown in Figure 1.4 on page 6 depicts a MDP with a discount factor of 1. However if a discount factor of 0.5 is used, the policy will change to favor closer rewards, as seen in Figure 1.5.

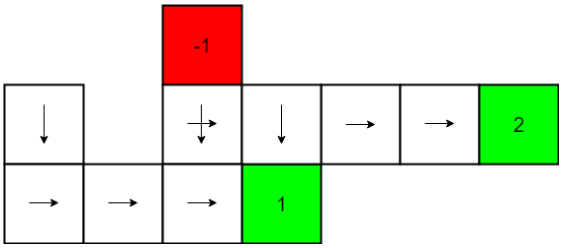


Figure 1.5: Example of applying an optimal policy to a MDP with lower value of γ

MDPs assume a markovian state and complete observability, but that does not fully describe the reinforcement learning problems present in Starcraft II, as the game includes partial observability where some states contain hidden information. Partially Observable Markov Decision Processes (POMDP) are used to model problems with partial observability, and this may model the problems better than a MDP.

Instead of introducing POMDP, we assume that complete observability of the state is not needed for sufficient use of the MDP model. We can instead view the partially observed states as a complete state of the game, which makes the markov assumption inaccurate due to states not containing all relevant information, as it is hidden in partial observability.

Instead we attempt to alleviate the problem with the markov assumption by introducing memory. Since the MDP model determines actions based on the current state only, it may be possible to help an agent get a better understanding of the current state, by remembering elements from previous observations through the use of memory, by encoding the previous 4 states and actions in the current state, and assuming this is the full state although that is not the case. This form of simple memory has previously been utilized in Atari games.[12]

Knowning the full state is extremely important in the full game of Starcraft II, as strategies and tactics differ based on the composition of units in the army of the enemy, which are partially observable. Because we do not focus entirely on the full game of Starcraft II, where this is most relevant, the simple encoding may suffice for the minigames where enemy army composition does not change over time.

1.2.2 Deep Neural Networks

In deep reinforcement learning, deep neural networks with multiple hidden layers are used due to the sheer size of the state-action space that makes it difficult to compute elements such as the action-value and state-value functions. Neural networks with multiple hidden layers are used to approximate a function using, typically, stochastic gradient descent optimization algorithms. If a value function is the approximation target for a neural network, the function

$V_\pi(s;\theta_V)$ signifies that the estimated value function output is influenced by the neural network parameters θ_V.

1.2.3 Learning Methods

There are two distinct areas of methods that can be used to perform reinforcement learning; model-free and model-based learning methods. In this report we focus on value based and parameterized-policy based learning, both of which are model-free learning methods. These types of learning focus on interacting with the environment itself, and compute a value function or a policy respectively.

Model-based learning is about learning a model of the environment, which is then used to find the optimal policy. This means the underlying elements of the MDP, such as the transition function and the reward function, are explicitly defined in the learned model. The model is then used to compute the optimal policy, possibly using Adaptive Dynamic Programming.[13]

Model-free learning is about sampling the environment to estimate either a value function or a policy. Model-free methods cannot, unlike model-based methods, predict the next state or reward.

Value-based learning is about estimating a value function, described in Section 1.2.1 on page 5. The learned value function is then used to derive the optimal policy. An example of value-based learning is Q-learning where action-values (Q-values) are estimated, and the policy is equivalent to selecting the action with the highest value, called the greedy policy.

The purpose of Q-learning is to compute the optimal Q-function seen in Equation (1.6) on page 6. Q-learning is off-policy, because the Q-function can be computed under any policy that is used. Q-learning assumes a greedy policy is followed, where the action with the highest action-value is chosen, but this includes no exploration, which is why epsilon-greedy is often used. Epsilon-greedy policies have a chance $1 - \epsilon$ to adhere to the greedy policy, or else explore by taking an action uniformly at random.

Policy-based learning is about estimating the policy, a set of action probabilities, using policy parameters. The policy parameters can then be improved using gradient ascent on the objective with respect to the parameters.[14, Section 13.1]

The learning method used in this report is a combination of value-based and policy-based methods, called actor-critic. We focus on an actor with a policy and a critic parameterized by a deep neural network.

1.2.4 Actor-Critic

Actor-critic methods are combinations between policy-based and value-based learning methods. The architecture for actor-critic can be seen on Figure 1.6 on the facing page, and is comprised of an actor that performs policy improvement on the policy parameters θ_π while ensuring exploration, and a critic estimating the underlying value function, that is used to improve the actor, and perform policy evaluation. Actor-critic methods are similar to policy iteration, but the reward- and transition-functions are unknown.[15]

The actor is supplied the current state, and outputs an action to be executed in the environment. After executing the action, the environment outputs the succeeding state, and the reward

obtained. The critic then receives the new state and reward, concluding a full experience. To perform policy evaluation, the critic estimates values for the current state and the new state. The actor then performs a policy improvement step, based on the value estimates of the critic.

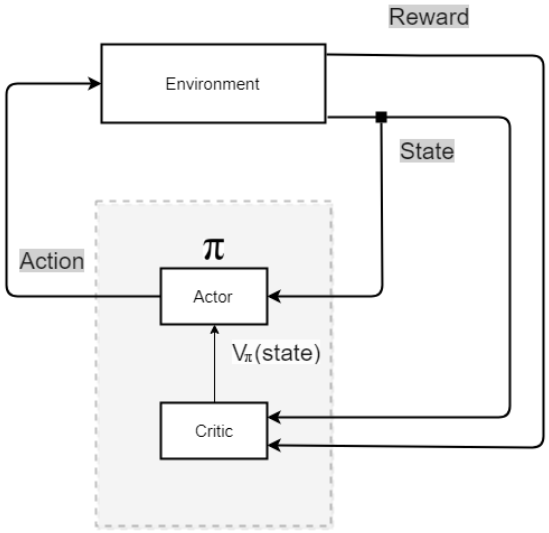


Figure 1.6: Visualization of the Actor-Critic architecture. The Actor performs policy evaluation using the value-estimate of the state, from the critic

The critics policy evaluation step utilizes bootstrapping[16].[14, Section 13.5] states that "Bootstrapping is updating the value estimate for a state from the estimated values of subsequent states" by some learning rate α:

$$V_{new}(s_t;\theta_V) = V_{old}(s_t;\theta_V) + \alpha\Big(r_t + \gamma V_{old}(s_{t+1};\theta_V) - V_{old}(s_t;\theta_V)\Big) \tag{1.7}$$

The 1-step return, shown in Equation (1.8), is used as an estimate of the value of a state s_t, based on the reward r_t received by entering the state, and the value estimate of the following state $V_\pi(s_{t+1};\theta_V)$. [14, Section 13.5]

$$R_\pi(s_t) = r_t + \gamma V_\pi(s_{t+1};\theta_V) \tag{1.8}$$

We know now that the critic is updated using bootstrapping, but updating the policy requires an objective function for the policy with respect to the policy parameters θ_π.

The objective function is the function that a learning algorithm in deep reinforcement learning is trying to optimize. The function describes the objective of the algorithm with respect to the elements of the MDP, and is used to determine changes to the parameters such that the objective function is maximized.

We are working with episodes of experiences, where each episode contains a finite amount of experiences in sequence. An experience is a tuple (s_t, a_t, r_t, s_{t+1}).

The policy objective function for the episodic case, seen in Equation (1.9) on the following page, is then equivalent to the true value of the start state s_0 assuming the policy π_{θ_π} is followed.[14, Section 13.2]

$$J(\theta_\pi) \doteq V_{\pi_{\theta_\pi}}(s_0) \tag{1.9}$$

In our case, the start state for every episode is not always the same, as Starcraft II introduces some randomization when starting a game, but the difference between each start in most mini-games and the full game is negligible.

The parameter update rule for improving the critic over a single step, to better approximate the true value function using parameters θ_V, is shown in Equation (1.10). The parameter update rule for improving the policy objective over a single step is shown in Equation (1.11). [14, Section 13.5]

$$\theta_V = \theta_V + \nabla_{\theta_V} V(s_t; \theta_V) * \alpha\Big(R_t - V_\pi(s_t; \theta_V)\Big) \tag{1.10}$$

$$\theta_\pi = \theta_\pi + \frac{\nabla_{\theta_\pi} \pi(a_t|s_t; \theta_\pi)}{\pi(a_t|s_t; \theta_\pi)} * \alpha\Big(R_t - V_\pi(s_t; \theta_V)\Big) \tag{1.11}$$

[14, P. 329] states that "the fractional vector $\frac{\nabla_{\theta_\pi} \pi(a_t|s_t; \theta_\pi)}{\pi(a_t|s_t; \theta_\pi)}$ is equivalent to the compact expression $\nabla_{\theta_\pi} \log \pi(a_t|s_t; \theta_\pi)$". Using the compact expression, the parameter update rule for improving the policy objective over a single step becomes:

$$\theta_\pi = \theta_\pi + \nabla_{\theta_\pi} \log \pi(a_t|s_t; \theta_\pi) * \alpha\Big(R_t - V_\pi(s_t; \theta_V)\Big) \tag{1.12}$$

1.2.5 Transfer Learning

A long standing goal of machine learning is continual learning, where agents can learn and remember a sequence of tasks while having the ability to transfer knowledge from previously learned tasks, to converge faster on new tasks[1].

Transfer learning focuses on being able to leverage previously learned knowledge to better/faster learn a new task. An example of the transfer learning principle can be seen on Figure 1.7 on the next page, where a model is trained on a source task, resulting in a trained model which we can try to extract knowledge from and apply to the target task[5].

As an example, we choose our source task as the supervised problem of detecting cars based on an image input. We start by training the model on the source task, giving us a model that is able to detect cars. Now lets say we have a target task of detecting trucks, which is a somewhat similar problem as detecting a car. We would now want to somehow leverage the knowledge stored in our source task model to solve our target task of detecting trucks.

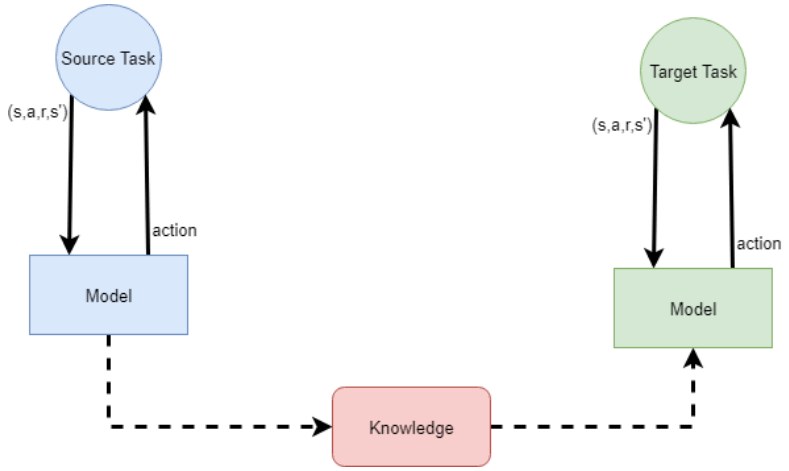


Figure 1.7: Example of transfer learning principle

How exactly knowledge is transferred depends entirely on the transfer learning technique being used. In this report we will be looking at two of the methods, namely Finetuning and Progressive Networks.

Finetuning is a simple method for using transfer learning between different tasks, by using the parameters from a neural network that has learned one task, and using those parameters instead of randomly initialized parameters when the agent starts learning a new task.

Progressive Networks is a network architecture that was developed by Google Deepmind in 2016[1]. Progressive Networks moves towards solving the continual learning problem. It leverages previously learned knowledge to learn new tasks, while avoiding loss of knowledge about previously learned tasks, also called catastrophic forgetting, by saving and freezing all parameters of the neural network, when moving on to the next task.

The action-state space of Starcraft II minigames and the full game share most of their elements, like unit types, minerals and so on, which are represented the same way across all games. The actions space is also the same across minigames, meaning that action 0 in one minigame will do the same as in another. This could mean that there is a potential to reuse features learned from one of the minigames in another. Both Finetuning and Progressive Networks have proven to work on the game related environment Atari 2600[1].

1.3 Problem Statement

After the introduction of relevant theory and information, which created a foundation to work upon, we now need to determine the problem that will be in the scope of this project.

The problem which we will be focusing on in this project, is an extension of the following problem statement, which was used in our previous project[2]:

"How can an agent be trained efficiently with reinforcement learning for a complex environment?"

We determined from this problem statement that none of the implemented reinforcement learning agents were able to fully grasp the minigames supplied with the Starcraft II learning en-

vironment[2, Chapter 5]. Even with a simplified action-state space, using few selected feature layers and actions, we were unable to reproduce results equivalent to current state-of-the-art[6] or human levels. Because we were unable to reproduce the results, we will in this project start by focusing on establishing our own baseline agents, using the full action-state space of the Starcraft II learning environment. After establishing baseline agents, our goal is to utilize transfer learning using the best performing agent, to give transfer learning the best foundation, to make the agent achieve a better average reward and converge faster, in the complex environment of Starcraft II.

This has inspired the following problem statement:

Can knowledge learned in a reinforcement learning setting from a complex environment, be leveraged to learn another task in a similar complex environment?

In order to analyze how previously learned knowledge is being leveraged, we perform transfer analysis. [1] introduces progressive networks, and a way of calculating if previous knowledge is leveraged for new tasks, without only monitoring convergence and average reward return. This is important as the convergence in the complex environment of Starcraft II is highly unstable, which we discovered in our last project[2]. Therefore in order to determine the amount of transferred knowledge from performance results, a very large amount of training tests would be needed, which is impractical with our limited resources.

Chapter 2: Agents

In this chapter we will determine which agent we will continue development on, from those developed in our previous project[2], and establish baselines for future agents. The contesting agents will be described, tested and chosen for further development.

We aim to find the best agent, to give the best foundation possible for transfer learning. For transfer learning, it is essential that the underlying agent can learn something meaningful from the environment. We also use the baseline to later measure the impact transfer learning in relation to performance metrics such as convergence speed, max reward, and average reward, in order to measure whether or not transfer learning has a non-negligible impact in a very complex environment such as Starcraft II.

Baseline Agents
We have previously tested different agents that could be used for transfer learning. The tests and findings can be read about in our 9th semester project[2]. These agents include a DQN, A3C and our own modified A3C agent, that we call SA3C. From these agents we will only continue with A3C and SA3C. We will not be using DQN because it performed very poorly in previous tests on the Starcraft II environment.

The A3C algorithm was used and confirmed to be the best among the algorithms tested by Google DeepMind in their research with the Starcraft II environment[6]. Based on the A3C algorithm used by the DeepMind team, we created a slightly modified version, that we named SA3C[2]. We will modify and test these two baseline agents from our last semester project. Both of these agents from the last semester project did not incorporate the use of all actions, all non-spatial features, all screen feature layers, and did not use any minimap feature layers. Some mini-games as well as the full game require the agents to have access to these, so changes had to be made for more general purpose agents. The new SA3C implementation has had further changes, which are described in Section 2.2 on page 20.

Along with the A3C and SA3C agents, we include a modified version of the A3C agent that incorporates memory, described in Section 2.3 on page 24.

2.1 A3C Agent

The Asynchronous Actor-Critic Agent (A3C) algorithm was chosen for this project due to the previous work done in relation to reinforcement learning in the PySC2 paper by the DeepMind team[6] and the work from our 9th semester project[2]. In both of these it was shown that the A3C algorithm was among the best of the tested reinforcement learning algorithms for Starcraft II, and therefore the implementation used in our previous work is used in this project. This section describes the various elements of the A3C algorithm, using an article on A3C[17] and the A3C paper[18] as sources.

The input to the network is the currently observed state s_t. The output is then the estimate of the state-value, $V_\pi(s_t; \theta_V)$, assuming the policy π_{θ_π} is followed, and a probability distribution $\pi(a_t|s_t; \theta')$ over all actions that are possible in the current state. Because Starcraft II requires

coordinate arguments for some actions, there is also a location parameter $\Pi_{l_t|s_t;\theta_\Pi}$ which is a probability distribution over all coordinates.

A3C is an actor-critic reinforcement learning algorithm that estimates the policy and value function using neural networks. As shown in Figure 2.1, the value estimate has the parameters θ_V, and shares only part of the neural network parameters with the policy and location estimates, namely the hidden layers with parameters θ. The neural network therefore has 4 sets of parameters; the shared parameters θ, the parameters θ_V for the value estimate, the parameters θ_π for the policy estimate, and the parameters θ_Π for the location estimate. It is in this section assumed that the parameters θ are always included when referring to the parameters of either of the estimates, e.g. the value function estimate $V_\pi(s_t;\theta_V)$ is equivalent to $V_\pi(s_t;\theta,\theta_V)$.

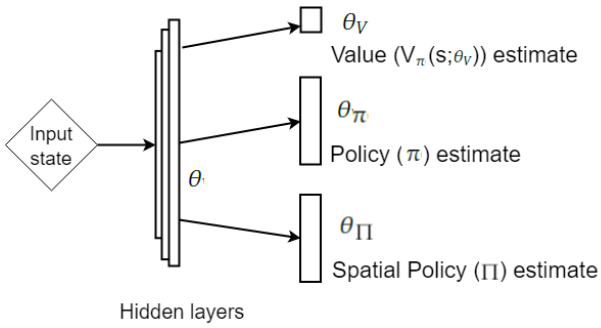


Figure 2.1: Parameters shared between outputs. All hidden layers are shared.

2.1.1 Approximating Episodic Advantage Actor-Critic

The policy, location, and value estimates make up the actor-critic of the A3C algorithm for Starcraft II. The critic is the estimate of the state-value function $V_\pi(s_t;\theta_V)$. The actor is the policy and location estimates, $\pi(a_t|s_t;\theta_\pi)$ which is a probability distribution over all actions and $\Pi(l_t|s_t;\theta_\Pi)$ which is a probability distribution over all coordinates in a ($64x64$) window.

The neural network used to approximate the state-value function, the location, and the policy requires a loss function that defines the objective in order to improve. The update rules for actor-critic, Equation (1.10) on page 10 and Equation (1.12) on page 10, are used as a basis for creating the loss function for the neural network. The loss, or error, is then used by an optimizer to compute and apply gradients for the network parameters that maximize the objective. The derivative of the loss should be equivalent to the update rules.

The neural network optimization algorithm used in this project is RMSProp[19], which is an optimizer that minimizes functions. Minimizing the loss is equivalent to maximizing the objective. The loss function for A3C is created based on the actor-critic update rules, such that the derivative of the loss gives a gradient equivalent to the update rule, and is used by the optimizer to minimize the error in the estimates of both the policy and the value function. The loss function consists of three parts; the the value loss, the policy loss, and the entropy regularization.

Advantage:
The value loss is a squared error loss of the advantage. Advantage is used as a measure of the value of actions relative to the value of the state, to distinguish how much better or worse an action is than what was expected. In the advantage function, shown in Equation (2.2) on the next page, the value of actions is estimated using the discounted n-step return R, seen in

Equation (2.1), which is calculated based on a sample of n experiences in sequence. The n-step return is an extension of 1-step return, described in Section 1.2.4 on page 8, to an n-degree bootstrapping.[18]

$$R_\pi(s_t, n) = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_\pi(s_{t+n}; \theta_V) \tag{2.1}$$

On Table 2.1 an example is shown for calculating the return for a sample of 3 experiences. The return is, in practice, usually calculated from the final timestep backwards to t_{start} in order to reduce the computation time. The reason becomes apparent in the Simplified Return column in the table, where the value of the return at timestep $t + 2$ can be saved and reused in the calculation of the return at timestep $t + 1$ and so on.

Timestep	Reward (r_t)	Return $R_\pi(s_t, n)$	Simplified Return $R_\pi(s_t, n)$
t_{start}	2	$r_t + \gamma * r_{t+1} + \gamma^2 * r_{t+2} + \gamma^3 * V_\pi(s_{t+3}; \theta_V)$	$r_t + \gamma * R_{t+1}$
t+1	1	$r_{t+1} + \gamma * r_{t+2} + \gamma^2 * V_\pi(s_{t+3}; \theta_V)$	$r_{t+1} + \gamma * R_{t+2}$
t+2	3	$r_{t+2} + \gamma * V_\pi(s_{t+3}; \theta_V)$	$r_{t+2} + \gamma * V_\pi(s_{t+3}; \theta_V)$

Table 2.1: Example of n-step Return calculation using $n = 3$

The advantage is used to evaluate the actor's policy, using the difference between actual return $R_\pi(s_t, n)$ and the critic, the estimated value following the policy, $V_\pi(s_t; \theta_V)$. The advantage function can be seen in Equation (2.2).

$$A(s_t, a_t; \theta_V) R_t - V_\pi(s_t; \theta_V) \tag{2.2}$$

Value Loss: The advantage is a calculated estimate of the actual observed value of executing the chosen actions in the experience batch, compared to the estimated value of following the policy, starting from current state. If the estimated state-value is higher than R, then the wrong action may have been selected, or the estimate may not be accurate yet. The squared error emphasizes the larger values, such that a very low or very high advantage is influencing the loss much more than close-to-expected advantage values. The value loss then becomes:

$$ValueLoss_{\pi(\theta_V)}(s_t) = [R_t - V_\pi(s_t; \theta_V)]^2 \tag{2.3}$$

The value loss is derived from the update rule, shown in Actor-Critic at Equation (1.10) on page 10. We derive the gradient from the loss, to show that the gradient of the loss is equivalent to the update rule. The derivatives assume that R_t is independent of the parameters θ_V. This is assumed because the loss is derived for true-gradient methods, where R_t is replaced with the true value function $V_\pi(s_t)$ which is constant. However we still use the true-gradient method's loss function in semi-gradient methods like the A3C, where a bootstrapping critic is used in the calculation of R_t.[14, Chapter 9]

The gradient is derived from the value loss in Equation (2.4):

$$ValueLoss_{\pi(\theta_V)}(s_t) = \frac{1}{2}[R_t - V_\pi(s_t; \theta_V)]^2 \tag{2.4}$$

The $\frac{1}{2}$ fraction is added to simplify the gradient, by removing the 2 after calculating the derivative of the square. The gradient of the loss is derived in Equation (2.5).

$$\begin{aligned} ValueGradient_{\pi(\theta_V)}(s_t) &= \frac{1}{2}\nabla[R_t - V_\pi(s_t;\theta_V)]^2 \\ &= \frac{1}{2} * 2 * (R_t - V_\pi(s_t;\theta_V)) * \nabla(R_t - V_\pi(s_t;\theta_V)) \\ &= (R_t - V_\pi(s_t;\theta_V))\nabla V_\pi(s_t;\theta_V) \end{aligned} \tag{2.5}$$

This gradient is equivalent to the value update rule displayed in Equation (1.10) on page 10, when applying the gradient using an optimizer that manages the learning rate.

Policy Loss:
The policy loss is the negative log-likelihood of the policy probability distribution multiplied by the advantage. The log-likelihood will increase the loss for actions with low probability, and make the result negative for minimization purposes. This helps with improving the probability for an action that was better than expected, according to the advantage, where a lower probability for the good action generates a large nudge, to help the policy to improve faster. The action a drawn from the policy π in the current state s_t is used for calculating the loss.

Because we have both an action policy, and a location, the loss is calculated for both estimates, and added together. Some actions in the action policy require a location that determines where the action should be used.

The policy loss does not include any learning rate term, and is negative. The previously mentioned optimizer minimizes rather than maximize the loss, and also manages the learning rate term rather than it being explicitly present in the loss. The loss for the policy and the location is shown on Equation (2.6) and Equation (2.7).

$$PolicyLoss_{\pi(\theta_\pi,\theta_V)}(s_t, a_t) = -\log \pi(a_t|s_t;\theta_\pi)(R_t - V_\pi(s_t;\theta_V)) \tag{2.6}$$

$$LocationLoss_{\Pi(\theta_\Pi,\theta_V)}(s_t, l_t) = -\log \Pi(l_t|s_t;\theta_\Pi)(R_t - V_\Pi(s_t;\theta_V)) \tag{2.7}$$

The location loss may influence the total loss a lot more than the policy loss, due to the granularity of the map. Currently we use a granularity of $64x64$ pixels to represent the observed values in the spatial features, but any other granularity can be used. The higher the granularity, the lower a probability will be assigned to many locations by $\Pi(l_t, s_t;\theta_\Pi)$. This may generate a very high loss when a location with a small probability is chosen, possibly making learning unstable.

The update rule, or gradient, that we want to obtain is shown in Equation (1.12) on page 10. We derive the gradient from the loss, to show that the gradient calculated based on loss is equivalent to the update rule. The derivatives assume that the advantage $(R_t - V_\pi(s_t;\theta_V))$ is independent of the parameters θ_π (and θ_Π for the location) and therefore constant.

The gradient of the policy, and location, loss is derived in Equation (2.8).

$$\begin{aligned} PolicyGradient_{\pi(\theta_\pi,\theta_V)}(s_t) &= \nabla\Big((R_t - V_\pi(s_t;\theta_V))\log \pi(a_t|s_t;\theta_\pi)\Big) \\ &= (R_t - V_\pi(s_t;\theta_V))\nabla \log \pi(a_t|s_t;\theta_\pi) \end{aligned} \tag{2.8}$$

The added policy loss and value loss are combined into a single loss function, allowing a single update for all the parameters θ_Π, θ_π, and θ_V:

$$Loss_{\pi(\theta_\pi,\theta_V),\Pi(\theta_\Pi,\theta_V)}(s_t, a_t, l_t) = PolicyLoss + LocationLoss + ValueLoss \tag{2.9}$$

Entropy:
The entropy regularization term was included as a way of improving exploration, by incorporating it into the policy[18]. The entropy regularization term is given by:

$$EntLoss_{\pi(\theta_\pi)}(s_t) = -[\beta H(\pi(s_t; \theta_\pi))] \tag{2.10}$$

H is the entropy of the policy and β is the entropy-weight hyperparameter for controlling the impact of the entropy regularization term. The entropy is calculated as $H(\pi(s_t; \theta_\pi)) = -[\pi(s_t; \theta_\pi) \cdot \log \pi(s; \theta_\pi)]$. The entropy term will be small when the policy probability distribution is concentrated around fewer actions, and largest when actions are equally probable.

The complete loss function for a single step t is seen in Equation (2.11).

$$Loss_{\pi(\theta_\pi,\theta_V),\Pi(\theta_\Pi,\theta_V)}(s_t, a_t, l_t) = PolicyLoss + LocationLoss + ValueLoss + EntLoss \tag{2.11}$$

We compute the loss in batches, where a batch is n number of experiences gathered in sequence. The loss computed is then averaged by the batch size n Equation (2.12). This is repeated until some number of total steps t_{max} have been performed. Π

$$Loss_{\pi(\theta_\pi,\theta_V),\Pi(\theta_\Pi,\theta_V)}(s_n, a_n, l_n) = \frac{1}{n} * \sum_{t=1}^{n} \Big(PolicyLoss + SpatialPolicyLoss + ValueLoss + EntLoss \Big) \tag{2.12}$$

2.1.2 *Asynchronous* Advantage Actor-Critic

The A3C algorithm is an algorithm that allows for multiple actor-learners to run concurrently, each with its own computing unit (such as a CPU core) and a different copy of the environment. The asynchronous actor-learners allow for more diversity in the experience, a (s, a, r, s') tuple, used for batch updates of the neural network, as each of the environments may be acted upon differently by each actor-learner. The diverse experience helps the algorithm not to overfit on repeating experience, due to the similarity coming from the nature of progress in games where each experience following another may be very similar.

Each actor-learner maintains a separate copy of the neural network, and then renews their copy every time training has been performed. The renewal happens by cloning a shared global neural network, that always has the latest and most updated network. The parameters of the global and local networks are referred to as θ and θ' respectively.

Each actor-learner copies the parameters of the global network to its local network when it starts learning, and gathers experience until the experience buffer (batch), that contains some number of sequential (s, a, r, s') experiences, is full or a terminating state is reached. The loss is then calculated from the batch of experience using the parameters θ'_π, θ'_Π and θ'_V of the local network, and used with the optimizer to calculate and apply the gradient to the parameters of the global network. After applying the gradients to the global network, the local network is

once again renewed, by cloning the global network. Afterwards the process repeats until some number of global steps have been performed by all the actor-learners combined.

The implementation of the A3C algorithm, used in the project, follows the pseudo-code present in Algorithm 1.

Algorithm 1 A3C - Pseudocode for each actor-learner used in this project

//Assume global network shared parameters θ_π, θ_Π and θ_V, and global shared step counter $T = 0$
//Assume local network parameters θ'_π, θ'_Π and θ'_V
//Assume experience buffer of size t_{max}
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradient: $d\theta_\pi \leftarrow 0$
  Reset gradient: $d\theta_\Pi \leftarrow 0$
  Reset gradient: $d\theta_V \leftarrow 0$
  Synchronize local parameters with global $\theta'_\pi = \theta_\pi$, $\theta'_\Pi = \theta_\Pi$ and $\theta'_V = \theta_V$
  $t_{start} = t$
  repeat
    Get state s_t
    Perform a_t according to policy $\pi(a_t|s_t;\theta'_\pi)$ and $\Pi(a_t|s_t;\theta'_\Pi)$
    Receive reward r_t and new state s_{t+1}
    Store experience (s_t, a_t, r_t, s_{t+1}) in experience buffer
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_V) & \text{for non-terminal } s_t \text{ // Return calculated from current state backwards} \end{cases}$
  for $i \in \{t-1, t-2, \ldots, t_{start}\}$ in experience buffer **do**
    $R \leftarrow r_i + \gamma R$ // *Reuse of Return calculation*
    ValueLoss: $L_V \leftarrow L_V + (R - V(s_i;\theta'_V))^2$
    PolicyLoss: $L_\pi \leftarrow L_\pi - \log \pi(a_i|s_i;\theta'_\pi)(R - V(s_i;\theta'_V))$
    LocationLoss: $L_\Pi \leftarrow L_\Pi - \log \Pi(l_i|s_i;\theta'_\Pi)(R - V(s_i;\theta'_V))$
    Accum. combined Loss: $L \leftarrow L + L_V + L_\pi + L_\Pi - \beta H(\pi(s_i;\theta'_\pi))$
  end for
  Calculate average loss $L_{avg} = \frac{L}{t_{max}}$
  Calculate gradient $d\theta_\pi$, $d\theta_\Pi$, and $d\theta_V$ of L_{avg} w.r.t. θ'_π, θ'_Π, and θ'_V respectively
  Perform asynchronous update of θ_π using $d\theta_\pi$, θ_Π using $d\theta_\Pi$, and of θ_V using $d\theta_V$
until $T >= T_{max}$

2.1.3 Network Architecture

There are important elements in the neural network architecture that was designed for use with Starcraft II and PySC2. This section will describe the elements of the architecture used. The architecture is heavily influenced by the one proposed by the deepmind team for their FullyConv A3C agent[6].

In Figure 2.2 on the next page the architecture is visualized. The diamonds represent input, the circles represent operations, and the arrows indicate the flow of information. It consists

of fully-connected dense layers, and convolutional layers[2, Section 2.3.6]. The information retrievable through the PySC2 library, described in Section 1.1.1 on page 3, has influenced the choice of the types of layers used. We feed the non-spatial features into a dense layer, because there is no inherent underlying structure in these features that we have to extract, besides the meaning of the values themselves. The spatial features are image-based, and so convolutional layers are used to detect an underlying structure in the data processed.

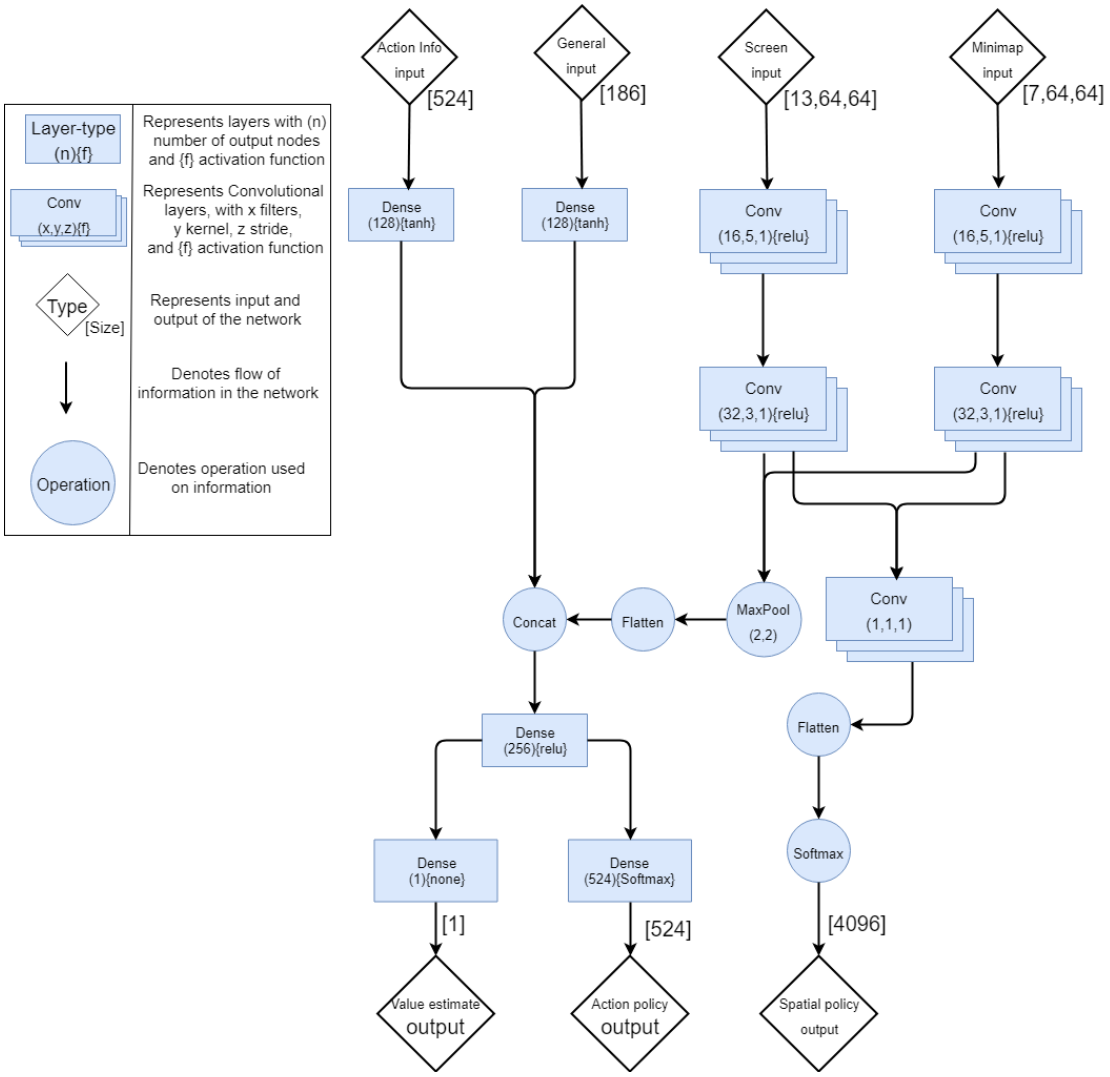


Figure 2.2: The neural network used in our A3C implementation

The action policy and state-value estimates are both utilizing the state-representation that combines all observed information, because these depend on all information found, but not particularly the spatial locality of the data. E.g. if resources, a non-spatial feature, are low, a resource miner should be selected and later ordered to gather resources, and if an enemy is spotted in the base, the army units should be selected and later ordered to attack the enemy. The choice for which action should be chosen is conditional on the situation, whereas the choice for where

an action should be used is location dependent.

Locations are extracted from spatial features as part of the spatial structure of the features, but the spatial structure is discarded once a fully-connected layer is used. Therefore a fully-convolutional network is used to determine the location output.

The minimap and screen spatial features are input separately into two convolutional layers detecting features on each input and later concatenated into a spatial state representation for the inputs of the location output and the state representation layers. Although the images for the screen and minimap are identical in size, they represent different categories of information, with some overlap in information between categories. Therefore there are two convolutional networks, one for the screen spatial features and one for the minimap spatial features.

2.2 SA3C Agent

The SA3C agent is based on the A3C, where the neural network architecture has been redesigned. The agent's neural network is redesigned so that the location output, instead of having a single output, has an output for each action that needs a location in the game. Thus the SA3C has many fully convolutional spatial networks, giving it the name Spatial-A3C.

The motivation for multiple spatial network outputs is that different action has very different interactions with the elements in the environment, and that means the agent needs to radically change location outputs, based on which action is used in the environment. As an example; a select action should not be used to choose enemies, and for an attack action, friendly units should not be chosen. Therefore coordinates chosen by the agent depends on which action the agent has chosen. This has proven useful in some of the Starcraft II mini-games according to our 9th semester project[2].

The idea initially came from our 9th semester project[2], where we implemented multiple neural networks in one agent, one for deciding an action, and then one for each action that requires a location, to chose the location for one specific action based on the raw spatial information from the environment, see Figure 2.3 on the next page. However this solution does not scale well, does not interact well with transfer learning, and was tested with a small subset of actions from the game. To give the agent a chance of mastering more complex mini-games, or even the full game, we would need to use around one hundred actions that require a location.

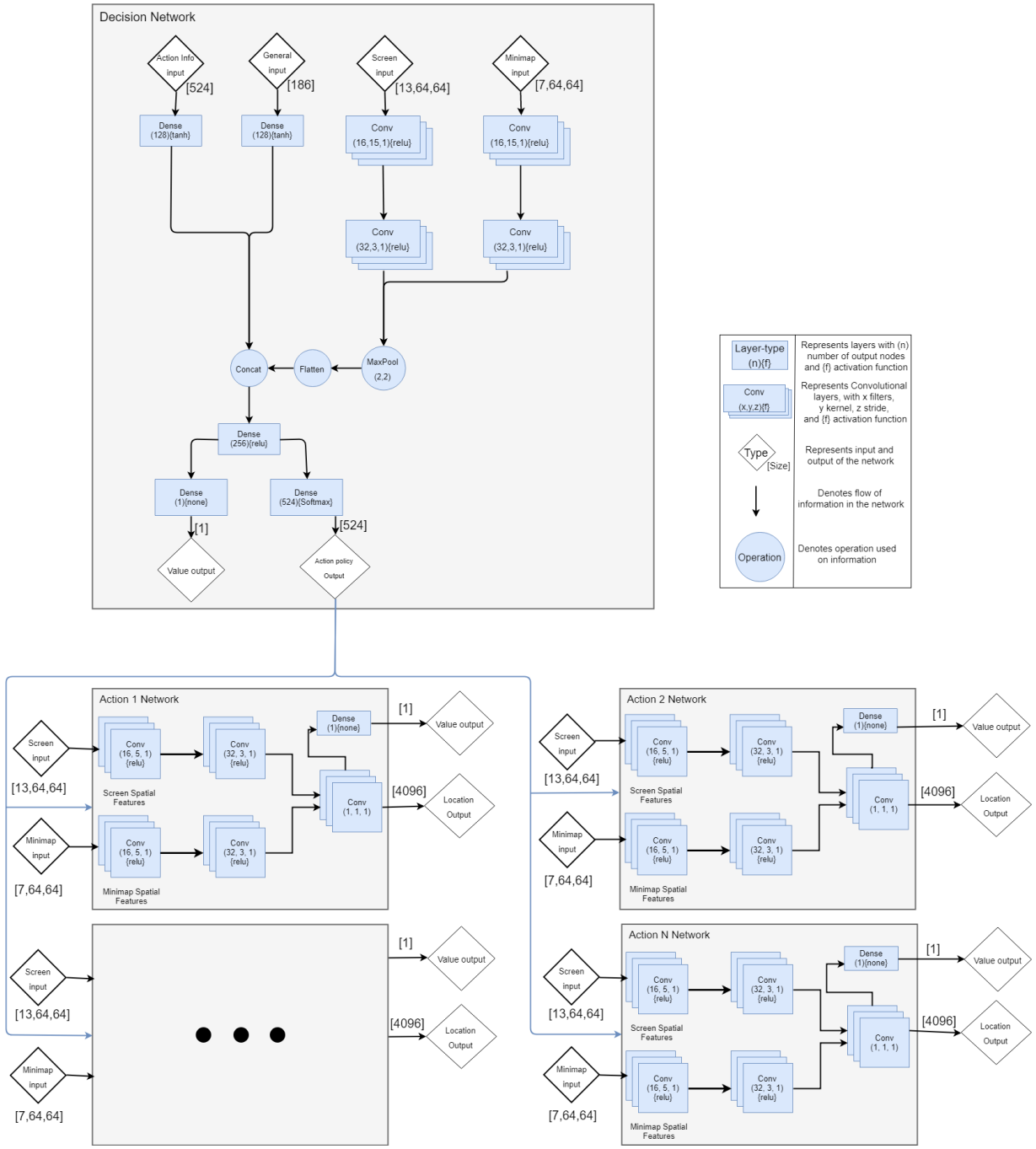


Figure 2.3: The old SA3C's neural networks, with multiple large fully convolutional neural networks. Arrows between networks describes the flow of the agent, no information is relayed from decision networks to action specific networks.

The old SA3C has, as mentioned, one fully convolutional network for each action in the game that requires a location. This network is only used and evaluated, when the specific action is

chosen by the decision neural network. In our old 9th semester project[2], we used 6 actions in the mini-games, resulting in about 4 times more network parameters than the standard A3C. However, we would like to scale that up to all actions, and with all actions available, there will be many times the parameters, and this is a noticeable and disfavorable amount for our agent.

2.2.1 Network Architecture

We decided to implement a revised version of the old SA3C idea, for a new and slightly different SA3C agent. The new agent aims to scale better, be transfer learning compatible and be able to handle upwards of a hundred actions that require a location output, with reasonable RAM and computational usage. This agent is, as the old SA3C, a network architecture modification, and uses all fundamental ideas from the A3C agent.

The idea is to reuse the convolutional layers of the main network, so that the action specific location networks do not need to do feature extraction from the raw input. This is done by relaying abstract spatial features, from the intermediate convolutional layer in the main network, and feed those to the action specific location networks, as can be seen by the dotted lines on Figure 2.4 on the facing page. The action specific location networks are only used when the main network's action policy chooses the specific action, and are only trained if there is at least one experience utilizing the network.

This modification of the architecture requires a change to the loss function to account for the loss of each of the action specific location networks. Each of the different networks has its own seperate loss, but all of them use both the policy loss and value loss as described in Section 2.1.1 on page 14. The loss for the main network in the new SA3C is shown in Equation (2.13). The main network does not have the location loss, because each of the seperate action networks manage the location.

$$Loss_{\pi(\theta_\pi,\theta_V)}(s_t, a_t) = PolicyLoss + ValueLoss + EntLoss \tag{2.13}$$

The loss for the action specific location networks is similar to the loss for the main network, except that the policy loss is swapped for the location loss since there is no action policy, as shown in Equation (2.14). Each of the action specific location networks have their own value estimate.

$$Loss_{\Pi(\theta_\Pi,\theta_V)}(s_t, l_t) = LocationLoss + ValueLoss + EntLoss \tag{2.14}$$

The old SA3C is trained to look at raw input, extract action specific features, and choose a location for the specific action. The new SA3C is trained to create abstract features that should be used for all actions, and then based on the universal abstract features determine the best location for each action using a smaller action specific location network.

Because we want to have all actions available, and have a chance to use the baseline agent with transfer learning, the old SA3C is not feasible. This means that we will only be going forward with the new version of the SA3C agent.

We expect to see that the SA3C agent will yield better results in mini-games where the agent needs to use actions with high diversity, especially in mini-games where the agent needs to use selection and attack actions in sequence, as was indicated in our 9th semester report[2].

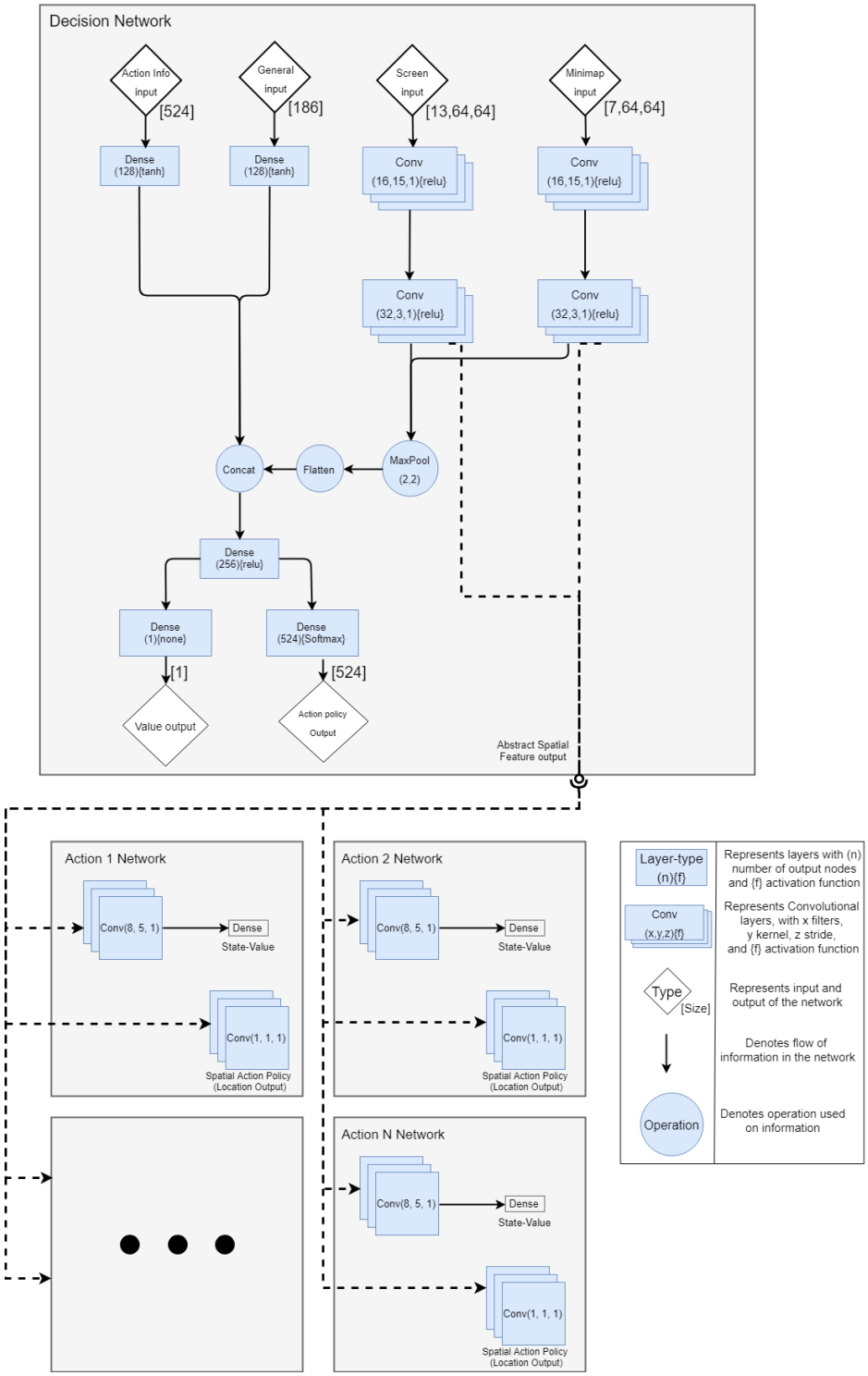


Figure 2.4: The new SA3C's neural networks, with multiple action specific location networks. The dotted lines describes a relay of abstract spatial features, from the main network, so the action specific location networks only receives input from the main network.

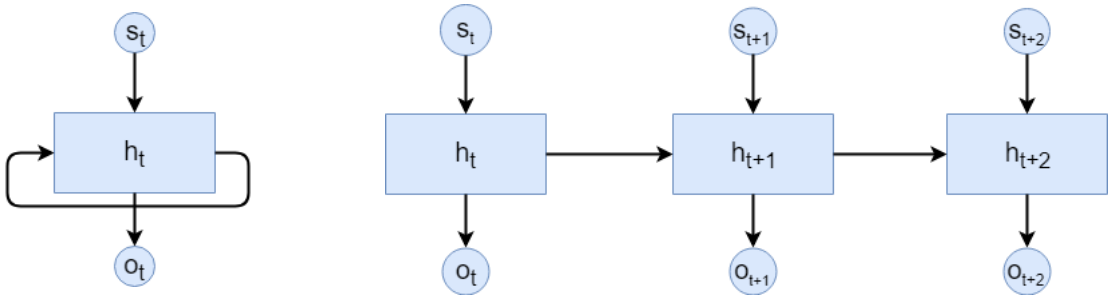
2.3 Memory Agents

We made an assumption Section 1.2.1 on page 5 that the complete state could be represented using memory to help alleviate the markov assumption. In this section some different types of memory will be looked at to determine how they may help alleviate the markov assumption for the reinforcement learning agents.

Memory in reinforcement learning agents has previously been used in atari games[12] and for Starcraft II [6] with noticeable improvements, but it is not always a clear winner.

In the atari games, the memory encoding was done by considering a sequence of observations as the representation for a state; the previous 4 frames was given as input as the current state. This type of memory, which we define as simple memory, does not require any alteration of the network architecture. Simple memory was introduced to give agents the opportunity to identify speed and direction of entities part of the state. This is also useful in Starcraft II, where units can be issued commands to move around. We extend this to include the actions used in the previous 4 frames, as many Starcraft II actions are compound actions where one action requires another action to be used beforehand; the select action has to be used before the move action. It is therefore useful to remember the used action.

The A3C FullyConv agent for Starcraft II, introduced in [6], was modified to include a Long Short-Term Memory (LSTM) unit in the FullyConvLSTM variant. The LSTM unit is a special kind of recurrent neural network (RNN), which is a network that is able to loop onto itself to allow for information to persist through steps in time. Figure 2.5 depicts a simple RNN, which takes an input and is able to pass along the information to the next timestep using a self-loop mechanism. For input s_{t+1} the RNN considers the input s_{t+1} and the information extracted from the previous timestep. The output o_{t+1} is then influenced by inputs over multiple timesteps represented in the hidden state h.



(a) Simple RNN

(b) Unrolled simple RNN

Figure 2.5: A simple recurrent neural network. s_t represents input from previous layers. The RNN passes the hidden information h_t to the next timestep h_{t+1}, and outputs a value o_t

LSTMs, and RNN in general, require an alteration of the architecture because they function as a layer rather than a modification of the input, as in the case with simple memory. The LSTM unit is useful because it is enables the agent to persist information for longer than the simple memory, because it is a type of RNN, and is especially good at considering dependencies over longer periods of time compared to a simple RNN due to the use of gates that control remembering input, forgetting memory and outputting information[20]. In Starcraft II this may be useful when trying to decipher which enemy units should be prioritized in a battle, as the LSTM may

help an agent towards being able to determine that friendly units die less often when a specific enemy unit type has been removed, allowing for countering strategies, or make the agent able to distinguish visited areas from non-visited areas to be better at scouting.

We focus on implementing agents that incorporate simple memory and the LSTM variant of a RNN, in order to determine if either is feasible as a baseline for Starcraft II.

2.3.1 Network Architecture

The architecture used for the memory variants are based on the architecture for the A3C agent. Due to the limited time available for testing, versions based on the SA3C will only be created if the A3C version performs well. The benefits should be very similar for A3C and SA3C.

The simple memory extension adds changes to the input. The non-spatial and spatial feature inputs are modified to include the features from the previous 4 frames, giving 4 times the size of the original input. The available actions non-spatial feature from the past 4 frames is not included, because actions that were available in the past are not relevant.

In order to remember the actions used in the previous frames, the available actions feature is extended to include the action IDs of the previous 4 actions used. The action memory does not include the location parameter if one was included. These changes can be seen at the top of Figure 2.6.

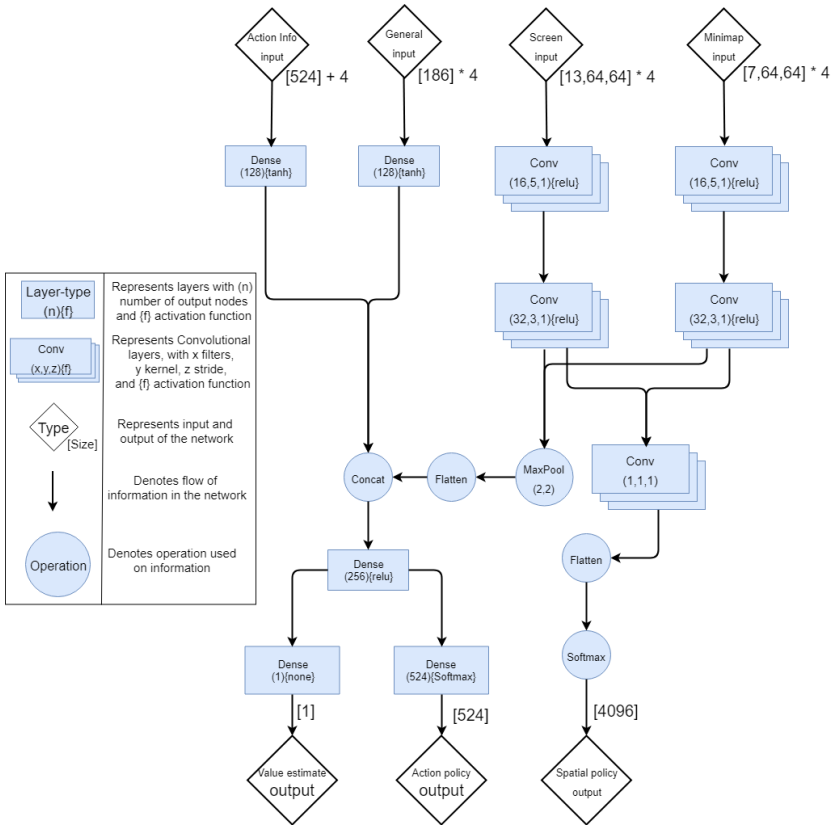


Figure 2.6: Architecture of the simple memory agent. Changes are made only to the input.

We decided to include to LSTM agents, one using a convolutional LSTM[21] and one using a conventional LSTM[22].

The conventional LSTM extension modifies the architecture to contain a LSTM layer after the fully-connected layer of size 256 representing the current state. The LSTM receives the full representation of the current state as input, and cycles through timesteps to determine temporal dependencies. The layer was implemented using the built-in LSTM layer available in tensorflow. The architecture for the LSTM version can be seen in Figure 2.7.

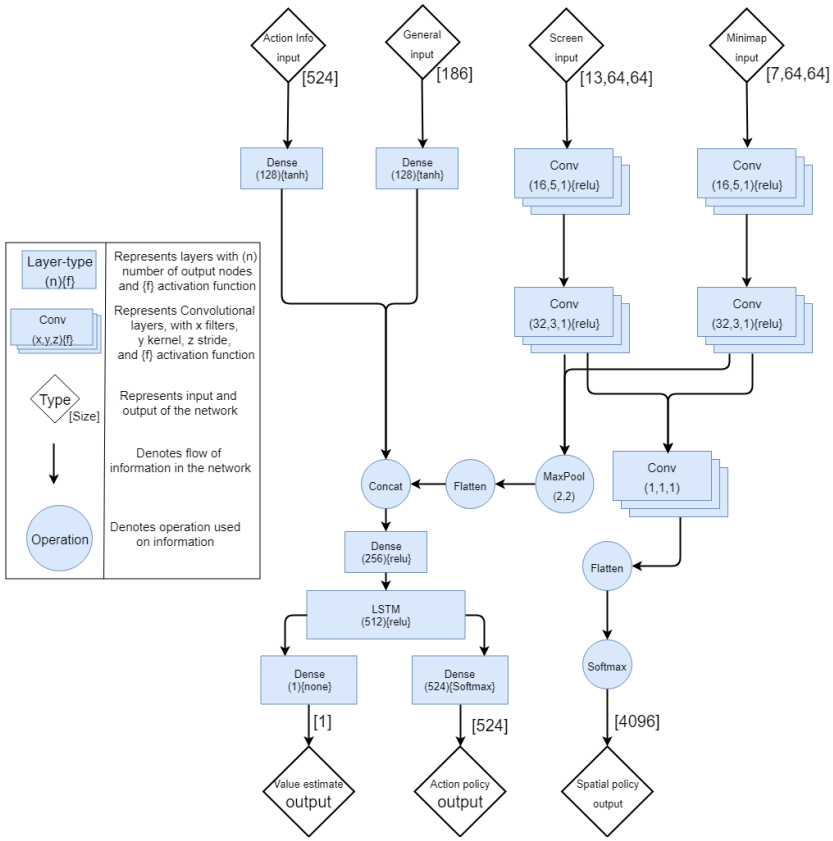


Figure 2.7: Architecture of the LSTM memory agent. A LSTM layer is inserted after the full state representation, just before the value-estimate and policy-estimate output layers.

The ConvLSTM extension modifies the A3C architecture to contain ConvLSTM2D layers instead of the second convolutional layers for both the screen and minimap inputs. The agent was implemented using the built-in ConvLSTM2D layer available in tensorflow. The architecture for the ConvLSTM version can be seen in Figure 2.8 on the facing page.

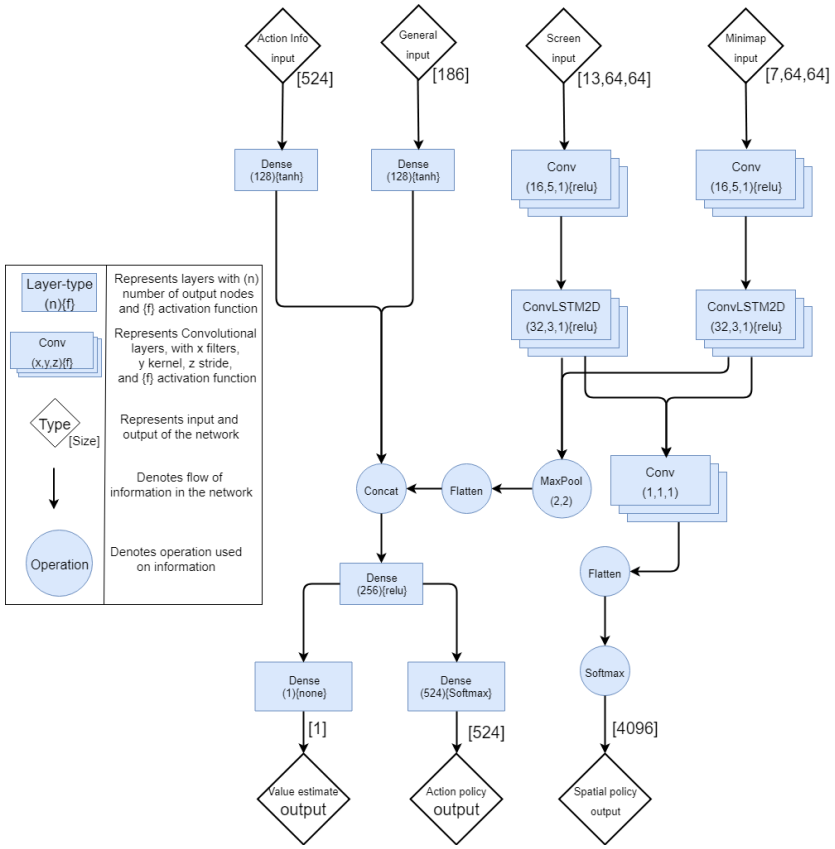


Figure 2.8: Architecture of the ConvLSTM agent. The second convolutional layers are replaced by ConvLSTM2D layers.

2.4 Tests and Findings

This section contains documentation of our tests of baselines, where we compare results of the A3C, SA3C, Simple memory, LSTM and ConvLSTM agents. The purpose is to find the agent that is best equipped to handle a complex environment such as Starcraft II. In order to give transfer learning the best conditions, we modify the best performing baseline.

2.4.1 Testing Procedure

In the baseline tests the agents will be trained on two Starcraft II minigames, namely FindAndDefeatZerglings and DefeatRoaches, which can be seen on Figure 2.9 on the next page.

FindAndDefeatZerglings In this minigame the player starts by being in control of three ranged combat units(marines), in a large enough map to require movement of the screen in order to view the entire map. Enemy melee combat units(zerglings) are spawned at random locations in the map, whenever one is defeated the player receives a reward of +1. If one of the player's units are defeated, a penalty of -1 is awarded. The minigame has a time

(a) Screenshot of the FindAndDefeatZerglings minigame

(b) Screenshot of the DefeatRoaches minigame

Figure 2.9: Screenshots of the minigames being used

limit of 180 seconds and requires the player to move their units around the map exploring it, while trying to find enemy units and defeat them.

DefeatRoaches In this minigame the player starts with nine ranged combat units(marines), and is opposed by four tougher enemy ranged units(roaches). When an enemy is defeated the player is given a reward of +10, however if one of the player's units dies a penalty of -1 is given. If all enemies are defeated a new set of four enemy units are spawned, and the player is given 5 more marines as reinforcement. This minigame has a time limit of 120 seconds and requires the player to make smart engagements with its units.

The RMSProp[19] optimizer is used for all agents, and they will be tested on each minigame five times, each with different learning rates between 10^{-3} and 10^{-5}, using an entropy weight of 10^{-3} for all tests. The learning rate is decayed to half its initial value during the training period. The agents are tested with 8 workers each, unless the agent is unable to fit in memory when using 8 workers, as is the case for SA3C which we had to limit to 6 workers. The Agents are trained for approximately 50 million game steps. The hyper parameters ranges are chosen because these are the hyper parameters that Google Deepmind use in their extensive tests[6]. We also confirmed this range to be functional in our previous semester report[2].

To measure the performance of agents, we have chosen to use the best moving average over 100 episodes. This measure of performance suggest that when an agent is performing well over some steps, the agents model could be saved, extracted and tested without training and still yield results close to the point at which the agent was saved. This assumes that the randomness of the game and the agent does not cause major fluctuations to the performance of an agent when the model is constant. To confirm that our speculations of this measure of performance of an agent is true, we check the performance of an agent using the model at different points during the training period, to see if the agent consistently achieve similar average score as the loaded model. The performance test is depicted on Figure 2.10 on the facing page, where the X points represent the points at which a model has been extracted from the A3C agent, as well as the average reward of running the agent with these models, where we have run them for 300 episodes without training the models. Looking at the X points on Figure 2.10 on the next page, we can see that the moving average reward over 100 episodes, depicted with the solid green line, fairly accurately represents the performance of the agent at different points, when the model is extracted and run to get an average reward over 300 episodes, as the performance

of the extracted models very closely corresponds to the training graph.

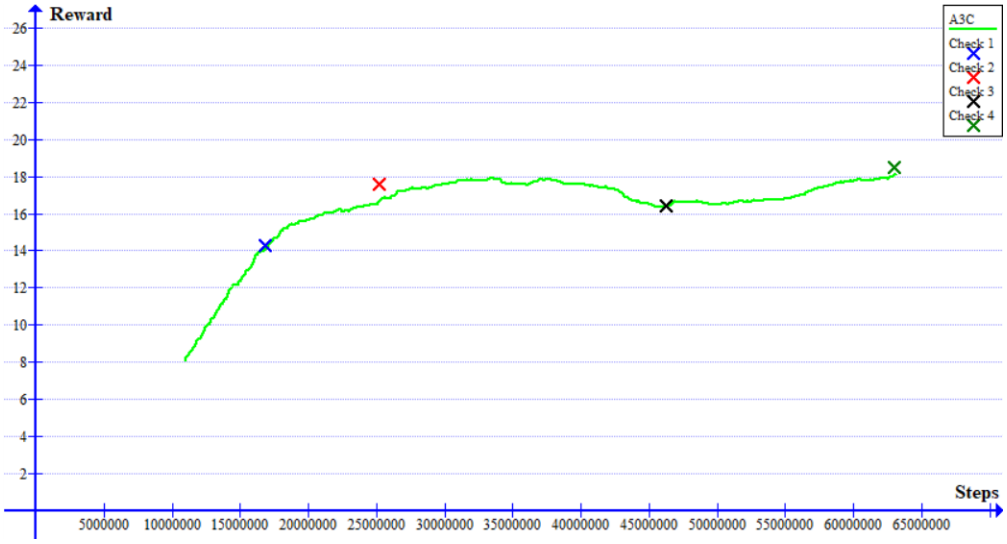


Figure 2.10: Training process for an A3C agent on the FindAndDefeatZerglings minigame showing moving average reward over 100 episodes as a function of game steps. The X points show the average reward over 300 episodes, for models extracted and frozen from the A3C agent at the given points.

The test for each agent with the best moving average over 100 episodes will then be shown on a graph while the rest will be shown on a table for each minigame with each test's best moving average and mean, describing how good the agent became, and how consistent the performance in the test was.

2.4.2 Results

We start by comparing the results of the various agents from the FindAndDefeatZerglings minigame, and afterwards compare the results from the DefeatRoaches minigame.

FindAndDefeatZerglings minigame
The tests for each agent that had the best running average over 100 episodes are pictured on Figure 2.11 on the following page. We see that for this minigame the best A3C agent is reliably better than all other agents, until the end where the best SA3C agent manages to obtain a similar best moving average over 100 episodes as the A3C. The differences are so small, that it is difficult to point out whether A3C or SA3C is better. Both are able to achieve equal best average reward over 100 episodes, so based on these tests both SA3C and A3C are equally equipped to handle the FindAndDefeatZerglings minigame.

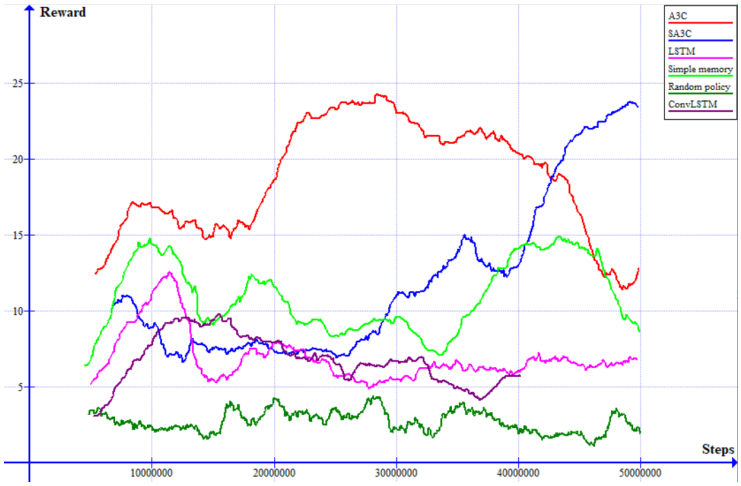


Figure 2.11: Training process on the FindAndDefeatZerglings minigame for A3C, SA3C, Simple memory, LSTM and ConvLSTM. Solid lines are running mean rewards over 100 episodes as a function of game steps for the run of each agent with the best running mean.

Table 2.2 lists the mean and the best running average over 100 episodes achieved by each agent for each test on the FindAndDefeatZerglings minigame.

FindAndDefeatZerglings					
Learning Rate	Metric	A3C	SA3C	Simple Memory	LSTM
0.001	Mean	7	10	6	0
	Best Moving Average	12	13	7	5
0.0005	Mean	18	15	8	4
	Best Moving Average	24	18	12	8
0.0001	Mean	14	14	11	7
	Best Moving Average	18	24	15	13
0.00005	Mean	12	11	10	4
	Best Moving Average	15	18	13	6
0.00001	Mean	12	12	8	3
	Best Moving Average	16	17	11	6

Table 2.2: Results for the different tests performed with each agent. For each test the learning rate was modified

For the LSTM variants, we see a considerably worse performance for both the LSTM and the ConvLSTM. We only performed a single test with the ConvLSTM agent due to the increase in wall-clock time required for a single test. We talk more in-depth about this later in Section 2.4.3 on page 32. Even the best LSTM agent had difficulty learning anything meaningful.

Based on the tests performed in the FindAndDefeatZerglings minigame, both the A3C and SA3C agents seem to be best equipped.

DefeatRoaches minigame

We decided to remove the LSTM and ConvLSTM agents from the test pool on the DefeatRoaches minigame, because they did not show promise in the FindAndDefeatZerglings minigame, and due to limited time and processing power we wanted to swap the time that would have been

spent on testing the LSTM variants on DefeatRoaches, with more time for testing transfer learning.

The DefeatRoaches results can be seen on Figure 2.12. We see that the SA3C is unable to reach a similar best moving average over 100 episodes as that reached by the A3C and Simple Memory agents around 3 million steps. The Simple Memory agent achieves slightly better best moving average over 100 episodes than A3C, but A3C achieves a more stable moving average.

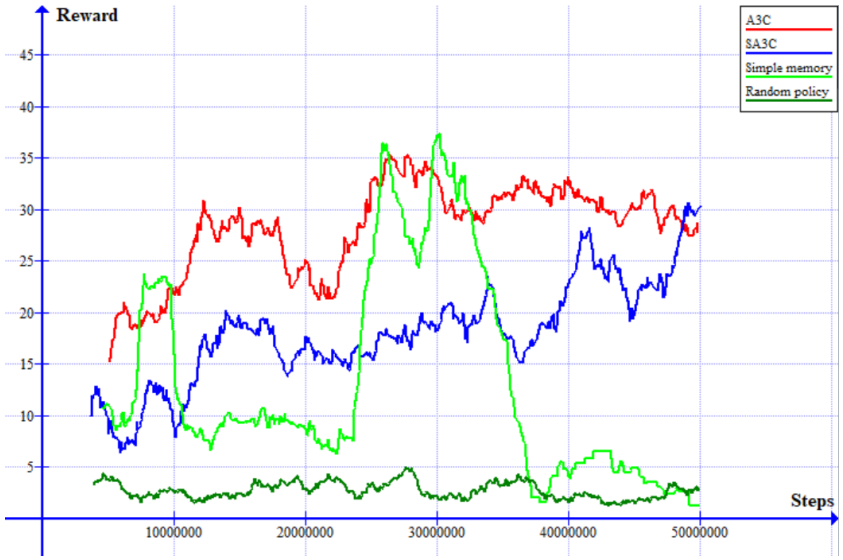


Figure 2.12: Training process on the DefeatRoaches minigame for A3C, SA3C, and Simple memory. Solid lines are running mean rewards over 100 episodes as a function of game steps for the run of each agent with the best running mean.

The simple memory agent performed much better in the DefeatRoaches minigame than in the FindAndDefeatZerglings minigame, as can also seen on Figure 2.12. It was able to achieve a slightly higher best moving average over 100 episodes than the A3C, but does not manage to stay stable and achieve a better mean over the span of the entire run. The simple memory agent seems to have more difficulty than the A3C in learning the minigame, and is therefore not deemed better equipped to handle the DefeatRoaches minigame.

These results are very similar to the findings from the FindAndDefeatZerglings tests, and show that both the A3C and SA3C agents are able to graph both minigames, unlike the other agents, however the A3C agent seems to achieve its best running average faster.

DefeatRoaches				
Learning Rate	Metric	A3C	SA3C	Simple Memory
0.001	Mean	12	5	5
	Best Moving Average	16	8	7
0.0005	Mean	28	13	8
	Best Moving Average	35	23	10
0.0001	Mean	22	17	16
	Best Moving Average	31	30	36
0.00005	Mean	17	15	11
	Best Moving Average	21	18	14
0.00001	Mean	11	13	2
	Best Moving Average	17	15	3

Table 2.3: Results for the different tests performed with each agent. For each test the learning rate was modified. The bold values are the ones displayed on Figure 2.12 on page 31

The A3C and SA3C agents were robust and show promise in both minigames, achieving a similar best moving average over 100 episodes, and are the two agents best equipped to handle the minigames out of all tested.

2.4.3 Test Discussion

In this section we point out some extra performance metrics to help us decide whether transfer learning should be applied to A3C or SA3C, and discuss issues we encountered during our tests.

Training time
During testing we noted the wall-clock time required for doing 50 million steps for each of the different agents. We decided to do this because we noticed a large difference in the time required for each agent to go through all 50 million steps. The average training time for 50 million steps for each agent is displayed in Table 2.4.

Agent:	**50M average training time (hours)**	**Relative to best training time**
A3C	64	1
SA3C	80	1.25
Simple Memory	92	1.44
LSTM	68	1.06
ConvLSTM	180	2.81

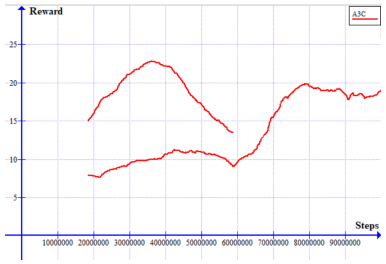
Table 2.4: Average wall-clock training time for each agent, in the Starcraft II environment.

The A3C agent has the best overall training time, taking 64 hours on average to go through 50 million steps. The A3C agent is also the least memory intensive agent, as all other agents expand upon the A3C architecture. Especially the SA3C agent is very memory intensive, unable to fit in the limited amount of memory available to us, unless only 6 workers are used.

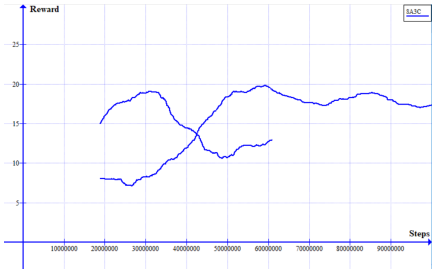
We mentioned that we only made a single ConvLSTM test because of the training time. The training time of the ConvLSTM agent was almost 3 times that of the A3C agent, and it was unable to perform better than any other agent, so since we were pressed on time for tests we found it reasonable to remove the remaining ConvLSTM tests from the testpool.

Unstable learning

During training in Starcraft II, we encountered varying results when we performed new tests with the same set of hyperparameters, showing that the network is extremely sensitive to the randomness present. Both random initialization of the network parameters and the minigame may influence this. This can be seen on Figure 2.13a and Figure 2.13b where two tests for the A3C and SA3C agents are shown. The two A3C agents were both run with identical hyperparameters and the two SA3C agents were run with identical hyperparameters, but in one test the SA3C is quite a bit worse than the other SA3C. Looking at the A3C tests on Figure 2.13a we can see something similar where one run starts out considerably better than the other.



(a) Two A3C runs on FindAndDefeatZerglings



(b) Two SA3C runs on FindAndDefeatZerglings

Figure 2.13: Training process on the FindAndDefeatZerglings minigame with identical hyperparameters.

Because there is such a large discrepancy between tests of the same set of parameters, it is difficult to conclude which agent is the best suited for the Starcraft II minigames, as more tests could skew the favor towards either of the A3C or SA3C agents. Doing more tests would give a better foundation for making a conclusion, but due to our time-constraints that is not viable for us to pursue.

Sometimes an agent would crash during training, and never recover again. This did not happened to all agents we tested, and if it happened we would restart the test if the agent had not yet reached the 50 million steps. Figure 2.14 on the next page shows a run that was saved, from an A3C agent test where this happened before 50M steps was reached. Looking at the behaviour of the agent at this point, we observed upon a reload of the model, after the crash, that the agent no longer attacked the enemy units, but instead never moved its own units. This might be because the agent has learned that it is best to avoid the enemies all together instead of receiving a negative reward for losing units to the enemy.

Figure 2.14: Training process on the DefeatRoaches minigame. A3C agent crashes and does not recover.

Through out testing all the agents, we have found out that with a specific set of hyperparameters, the A3C agent were more stable. Furthermore we have made small changes to the agent, that also improved the performance and stability, for example slightly changing the entropy so that it has one entropy for location output and one for action policy. This could also be done to the A3C's policy loss, so that they could be scaled in a manner that either favor one or the other or made them equally impactful. For the transfer learning, the agent has these small changes and we will only be using the best set of hyper parameters, as we do not have the time or computational power to go through multiple hyperparameters.

2.4.4 Test Conclusion

Both the A3C and SA3C agents were able to improve on both minigames during the training period. As can be seen on Figure 2.12 on page 31, results for the DefeatRoaches minigame, the running average reward was better for the A3C for most parts of the training, and only in the end did the SA3C start to perform a bit better. For the DefeatRoaches minigame, the A3C's best and worst run both perform better than its counterparts from the SA3C, and the Simple Memory agent was able to achieve a similar running average reward as the A3C agent, however in the FindAndDefeatZerglings minigame the SA3C looks more stable, and ends up with a better running average reward at 50 million steps than the A3C, where the Simple Memory agent is unable to grasp the minigame as well as A3C and SA3C.

Both the A3C and SA3C agents are able to grasp the minigames at similar performance levels, and therefore both agents could be potential candidates as a foundation for using transfer learning. However since the performance level was very close and the fact the SA3C requires a bit more resources and was slower than A3C, we will continue to work with the regular A3C, because we believe that transfer learning will exacerbate the issues with memory and time.

Chapter 3: Transfer Learning

In this chapter we introduce the main topic of this report, Transfer learning. Transfer learning is the act of gaining knowledge from one or more source tasks, that can be leveraged in a target task to improve performance. Transfer learning can be done in multiple ways, each with their own advantages and disadvantages.

We select a transfer learning method that we implement and test. The tests are performed to help us conclude on the problem statement, seen in Section 1.3 on page 11, on whether transfer learning can be utilized in a complex environment such as Starcraft II.

3.1 Transfer Learning Methods

In this section we introduce two methods used for applying transfer learning. The first method, finetuning, is the most wide-spread transfer learning method, and is relatively simple to implement, test and use. The second method, progressive networks, is a relatively new transfer learning method that both leverages transfer of knowledge between tasks as finetuning, and avoids loss of knowledge from previously learned tasks, called catastrophic forgetting, which finetuning does not.

Finetuning is a simple method for transferring knowledge between different tasks, by using the parameters from a neural network that has learned one task, and using those parameters instead of randomly initialized parameters when the agent starts learning a new task, this is illustrated in Figure 3.1. Finetuning is a well established and widely used method, because it works with many tasks, is simple to implement and has been proven to yield good results[23, 24].

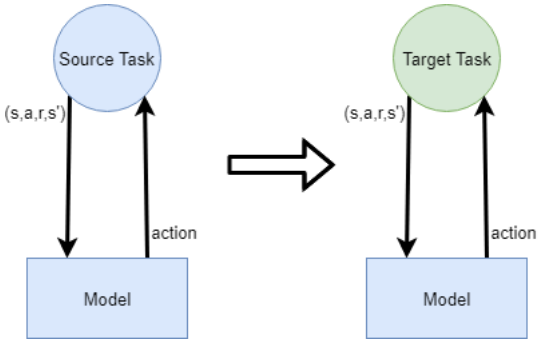


Figure 3.1: Example of Finetuning principle, where the tasks change, but the model stays the same.

When applying finetuning, the output layer of the model is changed to represent the target task, as shown in Figure 3.2 on the next page. After changing the model so the parameters are from a previously trained neural network, but the output layer is newly initialized and possibly changed in size depending on the target task, the model is now trained or "Finetuned" on the target task. If transferable aspects exists between the two tasks, the hope is that the training time of the target task is reduced or has better performance.

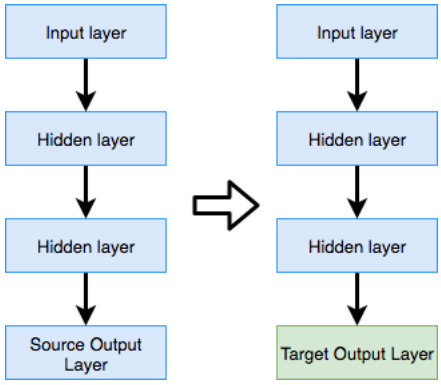


Figure 3.2: Applying finetuning on a neural network, the output layer is still initialized normally, while the rest of the parameters is initialized as the old model that has been trained.

Finetuning is susceptible to catastrophic forgetting. When finetuning on the target task, the learning process may overwrite the features learned on one or more source tasks, making it unable to leverage them.

In order to evaluate finetuning, several tests have to be performed to compare average performance measures before and after applying finetuning. In other words, only the performance graph of the training period can be analyzed to determine if finetuning is useful or not. In the tests of baseline agents, we have determined that the performance graphs in form of reward while training, can be varying. This means that multiple tests for average performance would be needed to determine how finetuning impacted the training period, which is not feasible with our level of computational power.

Progressive networks is a network architecture that was developed by Google Deepmind in 2016[1]. Progressive Networks moves towards solving the continual learning problem, where the focus is on learning continuously and adapt to every task in an incremental development of knowledge.

Progressive Networks leverages previously learned knowledge to learn new tasks, while avoiding catastrophic forgetting, by saving and freezing all parameters of the neural network used for a source task, and creating a random initialized copy of the neural network when moving on to a new target task. This architecture splits the network into columns, where the newest column is the network learning the target task, and the previous columns are frozen networks that have learned the source tasks.

The paper also introduces transfer analysis, described more in-depth in Section 3.2.1 on page 40, which allows for analysis of transfer between columns in the progressive network. Instead of limiting the analysis of transfer to the performance graph, the network itself is analyzed to determine how much the output depends on each layer in the network. Using this transfer analysis, it is possible to determine if transfer is happening without running a lot of tests, as is required when evaluating finetuning.

Due to our limited time and computing resources, we will focus on transfer learning with progressive networks in this project. We will implement progressive networks, perform tests and analyze them to determine if transfer of knowledge between tasks is happening.

3.2 Progressive Networks

This section describes the various elements of progressive networks, using the progressive networks paper[1] as the source.

Lets say we have a problem where we have to solve the three games shown in Figure 3.3. One could assume that some of the aspects in these games might overlap and knowledge about it be reused. For instance this could be that in Infinite Mario the agent figures out that it should collect coins, avoid enemies or something else entirely, which could perhaps be reused in Pac Man.

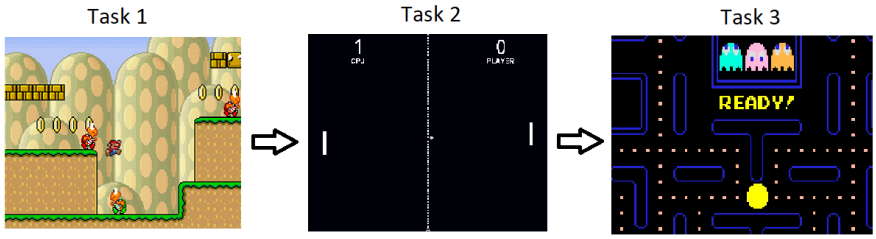


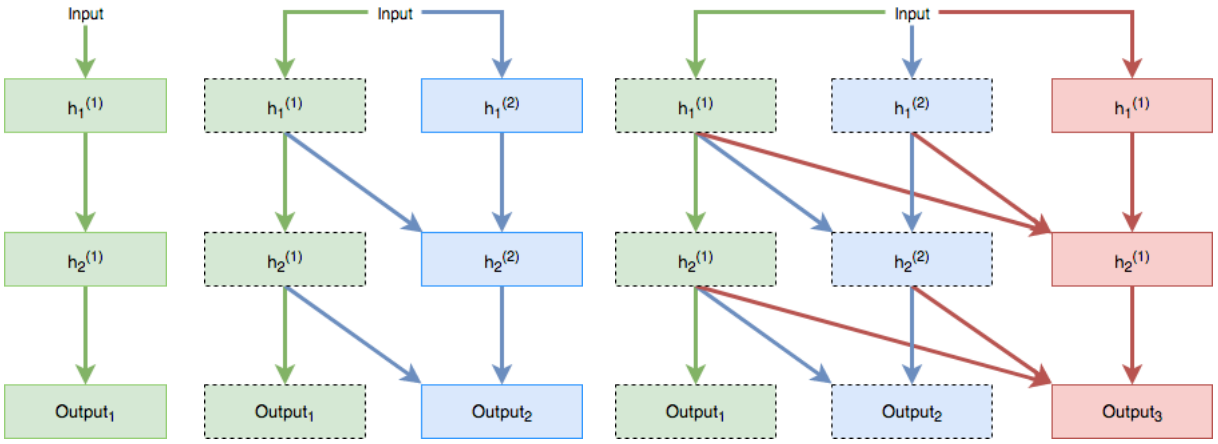
Figure 3.3: Screenshots of the three games, Infinite Mario, Atari Pong, and Pac Man

A progressive network starts out with one column as seen on Figure 3.4a on the following page, which is a deep neural network having hidden activations $h_i^{(i)}$, where $i \leq L$ and L is the number of layers in a column. $\theta^{(1)}$ represents the parameters of column one. We start by training the parameters $\theta^{(1)}$ on the first task(Infinite Mario), until convergence. Now we have some "knowledge" stored in the first column which we want to leverage when learning our second task(Atari Pong), but at the same time we do not want to lose the obtained "knowledge" involving the first task, in case we want to train on a third task, we want to leverage the "knowledge" from both the first and second task. To avoid loosing any knowledge about the first task, the parameters $\theta^{(1)}$ are frozen and a new column with random initialized parameters $\theta^{(2)}$ is instantiated. In order to leverage the "knowledge" from the first column, the definition of the hidden activations for the $i'th$ layer of the second column h_i^2 is changed to receive input from both the previous layer of the current column and of the previous column, corresponding to $h_{i-1}^{(2)}$ and $h_{i-1}^{(1)}$. The two column network can be seen on Figure 3.4b on the next page, where each column is a different color and use the connections of the arrows that match in color. Now the parameters $\theta^{(2)}$ can be trained, while hopefully being able to use "knowledge" from the first column. Doing the same again for the third task(Pac Man) we will get a three column network as seen on Figure 3.4c on the following page. Now our three column network has a column for each task that is able to do that task, however the network is not able to choose which column to use for a task by itself. One way to do this would be to label the tasks and columns. As we accumulate an increasing number of columns by training on more tasks, we will have more "knowledge" to leverage when training on new tasks.

This can be generalized to k tasks as follows:

$$h_i^{(k)} = ReLU((W_i^{(k)} h_{(i-1)}^{(k)} + B_i^{(k)}) + \sum_{j<k} h_{(i-1)}^{(j)} U_i^{(k:j)}), \tag{3.1}$$

where $W_i^{(k)} \in \mathbb{R}^{n_{(i-1)} \times n_i}$ is the weight matrix of the i'th layer of column k and n_i refers to the



(a) One column **(b)** Two columns **(c)** Three columns

Figure 3.4: Figure showing a one, two and three column progressive network, dotted borders means that the parameters are frozen

number of nodes of layer i, $B_i^{(k)}$ is the bias of the i'th layer of column k, and $U_i^{(k:j)} \in \mathbb{R}^{n_{i-1} \times n_i}$ are linear lateral connections from layer $i-1$ of column j, to layer i of column k. The linear lateral connections are trainable weight matrices, connecting the hidden activations of columns to subsequent columns. As mentioned $U_i^{(k:j)} \in \mathbb{R}^{n_{i-1} \times n_i}$ which means that the weight matrix $U_i^{(k:j)}$ is of size $n_{i-1} \times n_i$. The connection is done by multiplying the hidden activations $h_1^{(1)}$ with the weight matrix $U_2^{(2:1)}$. This is done for the hidden activations of each layer $h_{i-1}^{(<k)}$ in previous columns and the lateral connections $U_i^{(k:j)}$ are summed and added to the original layer output before being activated using a ReLU activation function.

The Finetuning method is limited to only leverage knowledge from one trained network at a time. Using the Finetuning approach you could however still make use of more tasks, by first training the network on a task(1), followed by another task(2) and then a third task(3). One of the problems with this approach could be that some of the features from task(1) could be overwritten during learning task(2), while they might still have been useful for task(3). Overwriting the features learned in previous tasks is catastrophic forgetting, which limits the transfer of knowledge to multiple target tasks.

Progressive networks do not have the same problem as they avoid catastrophic forgetting, and can make use of arbitrarily many source tasks. Each new column in a progressive network is intialized with random values, and slowly learn to use or not to use the previously learned knowledge using its lateral connections. Since each lateral connection only moves forward in the network and parameters for all previous columns $\{\theta^{(j)}; j < k\}$ are kept frozen, the outcome of earlier columns do not change and hence we avoid catastrophic forgetting.

In practice we use a slightly different approach also proposed in the progressive network paper[1], which makes use of adapters. Progressive networks with adapters are the only networks that they use and test in the paper [1]. The linear lateral connections are replaced with adapters, which feature a scalar to scale all inputs from previous columns at once and an extra dense/convolutional layer.

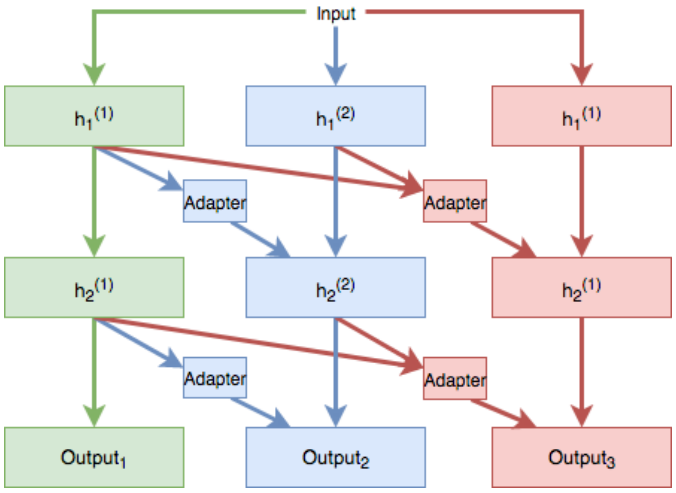


Figure 3.5: Visual representation of a progressive network with three columns

An overview of an adapter in a three column network with two lateral connections can be seen of Figure 3.6. This figure will be used as an example throughout the remainder of the section.

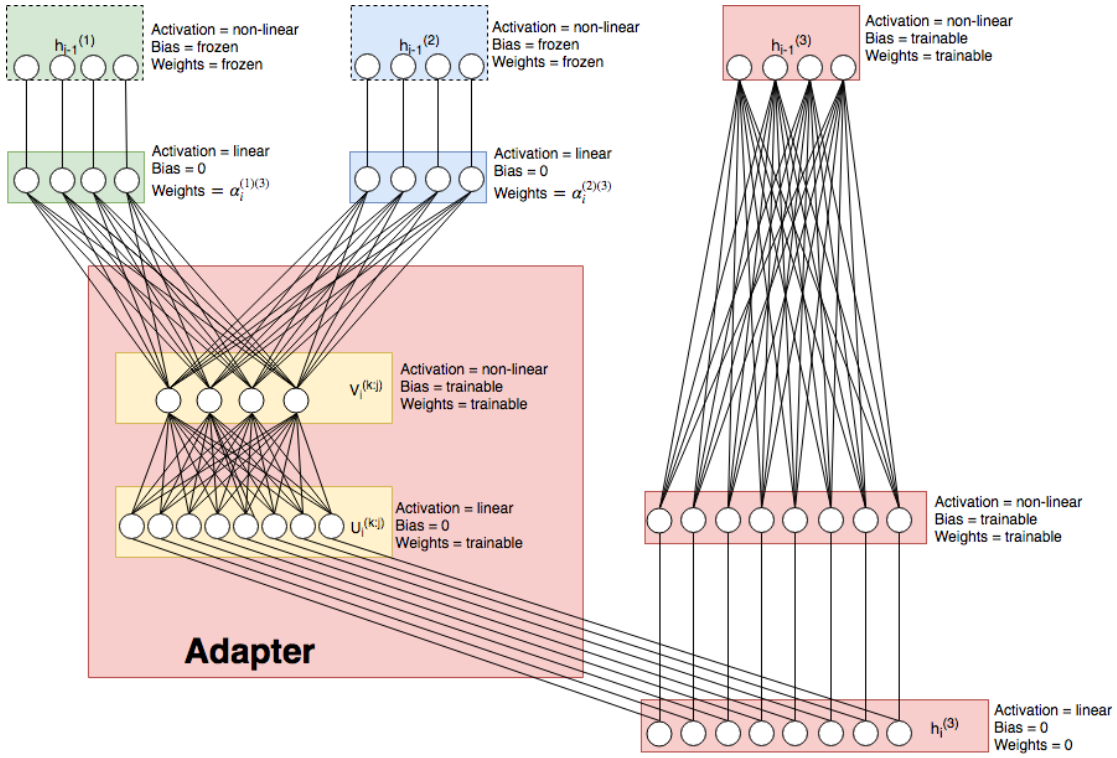


Figure 3.6: Figure showing an example of node connection in a three column network with a dense layer of size 4 to another of size 8

The input activations to the adapter is defined as $h_{i-1}^{(<k)} = [h_{i-1}^{(1)} \dots h_{i-1}^{(j)} \dots h_{i-1}^{(k-1)}]$, which has a

dimensionality equal to the number of hidden activations for the layer $i-1$ of all preceeding columns $n_{i-1}^{(<k)}$. On Figure 3.5 on page 39 the input activations for adapters corresponds to the diagonal arrows going to them, and on the example on Figure 3.6 on page 39 our input activations are $h_{i-1}^{(1)}$ and $h_{i-1}^{(2)}$. In the adapter we start by multiplying the input activations from each column with a separate trainable scalar $\alpha_i^{(j)(k)}$, with i being the layer that the adapter belongs to, j being the column from which input activations are multiplied with the adapter scalar, and k being the column which the adapter belongs to. The adapter scalar $\alpha_i^{(j)(k)}$ has the purpose of adjusting for different scales of different inputs. In the example on Figure 3.6 on page 39 the multiplication is represented as an un-trainable layer, with bias of 0, a linear activation function, and weights being set to $\alpha_i^{(j)(k)}$. The weights that the input activations are multiplied with changes in accordance with how $\alpha_i^{(j)(k)}$ is trained. We then concatenate each of the scaled activations, giving us the scaled input to the layer $V_i^{(k:j)}$. $V_i^{(k:j)}$ is a dense layer with a non-linear activation, it also results in a projection from the size of the concatenated scaled inputs $n_{i-1}^{(<k)}$ to the size of the preceeding layer n_{i-1}. We then add the information of $V_i^{(k:j)}$ to $h_i^{(k)}$(on the example this is $h_i^{(3)}$) via a linear lateral connection $U_i^{(k:j)}$, which is a trainable weight matrix, with a linear activation function. Hence the hidden activations for a layer i in a column k can be defined as seen on Equation (3.2).

$$h_i^{(k)} = f\Big(W_i^{(k)} h_{i-1}^{(k)} + B_i^{(k)} + U_i^{(k:j)} g\Big(W_{V_i^{(k:j)}} \alpha_i^{(<k)(k)} h_{i-1}^{(<k)} + B_{V_i^{(k:j)}}\Big)\Big) \tag{3.2}$$

Both f and g in Equation (3.2) refer to any non-linear activation function.

For convolutional layers the projection $V_i^{(k:j)}$ is done with a 1x1 convolutional layer with number of filters being equal to the number of filters for the input to the adapter h_{i-1}.

When measuring the knowledge transfer of transfer learning methods, it can be hard to see if previous knowledge is actually leveraged or if one agent performs better due too pure coincidence.

3.2.1 Transfer Analysis

Progressive networks keeps all the previous learned parameters frozen while training on new tasks, which makes it possible to analyze how much the agent's output depend on the previously learned parameters. This might be useful for explaining a potential increase in convergence or performance of the agent, or to find out if the previous knowledge is being leveraged or if it is just ignored while training on a new task. Transfer analysis in this explicit way cannot be done for finetuning, as the parameters are changed, and it is hard to tell how an agent leveraged the knowledge. Finetuning can be evaluated on the reward returns, however as we have showed, in Starcraft II this is not always feasible because of the highly unreliable reward returns that we get from the agent while training.

The paper for Progressive Neural Networks[1] proposed a measured perturbation method and a calculated output-sensitivity of the layers, for analyzing the networks with multiple columns.

Perturbation
Perturbation tests are simple in nature, but is a slow way of analysing the network. The tests are done by injecting noise into one layer in the network, and then by measuring the drop in

performance determining how reliant the agent is on the specific layer. The actual measure that we can use for our network, will require that we use noise with variance similar to the activation, then increasing the noise until we get 50% drop in rewards averaged over a couple of episodes. After the test, the increase in noise before the 50% drop will be our measure of how much the agent depends on the specific layer.

We will not be using the perturbation method, because the nature of the analysis method does not work well with our environment. Our agent's average reward varies a lot in the Starcraft II environment, the amount of episodes needed for an average reading of reward would have to be a large number. That and the fact that we would need a multiple average reward score per layer, and we have a lot of layers, means that it would require quite a lot of work, and computational power.

However the paper also introduced a calculated value, that approximates that of the perturbation method. Since that value is calculated and not measured, it means that we would not be required to run the agent for long to get an average reward measure.

Average Layer Sensitivity
The calculated approximation of the perturbation test is a method proposed by Rusu et. al.[1], the paper also proves that it yields similar results to the perturbation test. We will call this value, that describes the estimated output-sensitivity with respect to a specific layer, the Average Layer Sensitivity (ALS).

When calculating ALS, we use first order partial derivatives, a derivative of a function $f'(x)$ represents the amount of change in the output of the function as x changes. For a function with multiple variables $f(x, y)$ a first order partial derivative means that all but one variable is kept constant, for example if $f(x, y) = y^2 * x$ and we take the partial derivative w.r.t. x, then y will be kept as a constant value, the partial derivative of $f'_y(x)$ would be $f'_y(x) = y^2$, so the rate of change of the function $f(x, y) = y^2 * x$ when x changes, is y^2.

To calculate ALS as the outputs sensitivity to some activations in a layer, we need to take the partial derivative of the policy output in our neural network $\pi(s_t, \theta)$. The policy output is a vector that depends on the current input to the network and the parameters of the output. The partial derivatives is calculated with respect to normalized activations in a single layer i in a specific column k in our network $\hat{h}^{(k)}_{(i)}$. We do a normalization of the activations so that the scales of the derivatives are comparable across layers and columns. Thus a single partial derivative of the j'th output in $\pi(s_t, \theta)$ w.r.t the a'th normalized activation of $\hat{h}^{(k)}_{(i)}$ would be:

$$\frac{\partial \pi(j|s_t, \theta)}{\partial \hat{h}^{(k)}_{(i)}(a)} \tag{3.3}$$

The next step is to calculate the sensitivity of the output w.r.t. all the activations in a layer. Therefore we define one policy output's sensitivity to a full layer by creating a gradient vector, including all partial derivatives of the j'th policy output w.r.t. all the normalized activations of a layer, where the amount of activations in $\hat{h}^{(k)}_{(i)}$ is n. We define a vector, that describes exactly that:

$$\nabla_{\hat{h}^{(k)}_{(i)}} log(\pi(j|s_t, \theta)) = \left(\frac{\partial log(\pi(j|s_t, \theta))}{\partial \hat{h}^{(k)}_{(i)}(1)}, \frac{\partial log(\pi(j|s_t, \theta))}{\partial \hat{h}^{(k)}_{(i)}(2)}, \cdots, \frac{\partial log(\pi(j|s_t, \theta))}{\partial \hat{h}^{(k)}_{(i)}(n_i)} \right) \tag{3.4}$$

Now to get a single value for the j'th policy output's sensitivity w.r.t. the full layer, instead of a vector, we take the dot product of the vector with itself:

$$LS_{(i)}^{(k)}(\pi(j|s_t,\theta)) = \nabla_{\hat{h}_{(i)}^{(k)}} log(\pi(j|s_t,\theta)) \bullet \nabla_{\hat{h}_{(i)}^{(k)}} log(\pi(j|s_t,\theta)) \tag{3.5}$$

Giving us essentially the sum of squared partial derivatives of one output $\pi(j|s_t,\theta)$ w.r.t each index in the vector $\hat{h}_{(i)}^{(k)}$. Which will describe the sensitivity of the policy's j'th index w.r.t. the normalized hidden layer, we call this value the layer sensitivity score of the policy's j'th index w.r.t. a layer i at column k ($LS_{(i)}^{(k)}(\pi(j|s_t,\theta))$).

Until now every partial derivative has been for a single input, but before we can make our final Averaged Layer Sensitivity(ALS), we would need to calculate the sensitivity based on an averaged state distribution. We then need to sum all the Layer sensitivities for each policy output, as can be seen on Equation (3.6):

$$ALS_{(i)}^{(k)}(\pi(\theta)) = \mathbb{E}_{\rho(s)}\left[\sum_j LS_{(i)}^{(k)}(\pi(j|s_t,\theta))\right] \tag{3.6}$$

In practice to get an average state distribution, we average ALS scores for all steps in multiple episodes, until the change in the averaged ALS score is negligible.

The ALS score as calculated in Equation (3.6) describes the full policy's average sensitivity of layer i in column k. The ALS is a sum of squared partial derivatives of all policy outputs w.r.t. all normalized activations of nodes in a specific layer, giving an ALS score for each layer. The ALS calculation is essentially the same as Progressive Networks paper[1] suggests, they just call it Average Fisher Sensitivity (AFS) instead. We chose to rename AFS to ALS because the authors of the paper made the name based on small technicalities that were not at the core of the score. We believe Average Layer Sensitivity better describes the purpose of the score. In Section 3.4.1 on page 54 we will determine if the ALS values are as expected, to prove that ALS is correctly constructed and implemented.

3.3 Progressive Network Implementation

This section will describe how we implemented progressive networks as an extension to be used with Tensorflow. The "layers" submodule of Tensorflow allows the user to create neural networks by instantiating different types of layers and connecting them, while encapsulating the inner workings of the layers. An example of this can be seen on Listing 3.1, where we first instantiate a convolutional layer and define its different parameters such as the input, number of output etc, followed by a fully connected layer which uses the output of the convolutional layer as its input.

```
1 inpConv = layers.conv2d(input,
2                                num_outputs=16, #Number of filters
3                                kernel_size=5, #Size of conv window
4                                stride=1,
5                                activation_fn=tf.nn.relu)
6
```

```
dense1 = layers.fully_connected(layers.flatten(inpConv), #input
                                        num_outputs=128,
                                        activation_fn=tf.tanh)
```

Listing 3.1: Tensorflow constructing layers example.

Tensorflow contains many variants of layers that can be used, but no layers for progressive networks. Because of this our progressive network implementation is an implementation of a convolutional and a fully connected layer which can be used with Tensorflow just as its internal layers. For comparison, an example of how to instantiate a progressive network similar to the one in Listing 3.1 on page 42, but with two columns can be seen on Listing 3.2.

```
totalColumns = 2

inpConv = progConv2d(input,
                     numOutputs=16,
                     kernelSize=5,
                     stride=1,
                     activation=tf.nn.relu,
                     totColumns=totalColumns)

dense1 = progFc(layers.flatten(inpConv),
                  numOutputs=128,
                  scope="dense1",
                  totColumns=totalColumns,
                  activation=tf.tanh)
```

Listing 3.2: Progressive networks extension constructing layers example.

In order to explain how these layers are computed, psuedocode for a fully connected progressive layer can be seen in Algorithm 2 on the next page. The progFc procedure shown in the pseudo code creates layer i for all columns k, meaning that each use of *progFc* will compute the layers as shown by each black arrow in Figure 3.7 on the following page. The first call to *progFc* or *progConv2d* will compute the top layer for each column, and then the next call will compute the subsequent layer for each column. We can do this as new layers do not not depend on layers deeper into the network, but only the preceding set of layers.

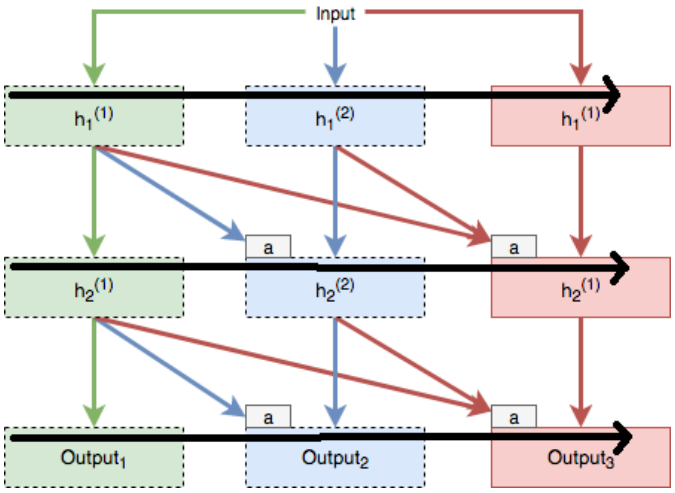


Figure 3.7: Visual representation of a progressive network with three columns, black lines indicate flow of layers are instantiated

In the algorithm we start by initializing a dense layer for the first column with the input being the hidden activations of the previous layer of the same column $h^{(1)}_{(i-1)}$. If the current layer is an input layer or from the first column, we activate it as shown on line 8 and the calculation of the layer is complete as the layer needs no adapter. If on the other hand the layer is not an input layer and not from the first column, we compute the adapter which returns a linear lateral connection $U^{(k:j)}_{(i)}$ and add it to the previous layer, before applying a non-linear activation function. This loop is continued until layer i has been computed for all columns.

Algorithm 2 Progressive fully connected layer - Pseudocode for constructing a fully connected progressive network layer

1: **procedure** PROGFC
//For more transparency between the formal description and the psuedocode, "i" will refer to the layer number of the progressive layer currently being created
2: **for** $h^{(k)}_i$ in $\{h^{(1)}_i \dots h^{(k_{max})}_i\}$ **do**
3: Initialize Dense layer with $h^{(k)}_{(i-1)}$ as input

$$h^{(k)}_i \leftarrow Dense(input = h^{(k)}_{(i-1)}, numOutputs = n^{(k)}_i)$$

4: **if** $h^{(k)}_i$ is *NOT* input layer *AND* $j > 0$ **then**
5: Adapter from $h^{(<k)}_{(i-1)}$ to $h^{(k)}_i$: $U^{(k:j)}_{(i)} \leftarrow adapter(h^{(<k)}_{(i-1)}, n^{(k)}_i)$
6: $h^{(k)}_i \leftarrow Activation(h^{(k)}_i + U^{(k:j)}_i)$
7: **else**
8: $h^{(k)}_i \leftarrow Activation(h^{(k)}_i)$
9: **end if**
10: **end for**
11: return $\{h^1_i \dots h^{k_{max}}_i\}$
12: **end procedure**

The adapter is computed as shown in the psuedocode on Algorithm 4 on page 47, which returns a linear lateral connection $U_i^{(k:j)}$ for all preceding columns $< k$ to current column k. Here we start for each input(each preceding column) $h_{(i-1)}^{(j)}$ instantiating an adapter scalar $\alpha_i^{((j)(k))}$ and multiplying the input with it. Then we create a dense layer $V_i^{(k:j)}$ which is our projection layer from $n_{(i-1)}^{(<k)}$ to $n_{(i-1)}$. This layer is activated and then we create a new dense layer without bias and with a linear activation which is our lateral connection $U_i^{(k:j)}$ and give the activated $V_i^{(k:j)}$ as input. The output of $U_i^{(k:j)}$ is returned as the adapter output.

Algorithm 3 Adapter for fully connected layer with fully connected input - Pseudocode of an adapter in a fully connected layer

1: **procedure** ADAPTER
2: **for** $h_{(i-1)}^{(j)}$ in $\{h_{(i-1)}^{(1)} \dots h_{(i-1)}^{(k-1)}\}$ **do**
3: Instantiate adapter scalar for the input $h_{(i-1)}^{(j)}$ to column k, $\alpha_i^{(j)(k)} \leftarrow random(0.01, 0.05)$
4: Multiply input hiddens with adapter scalar $h_{i-1}^{(j)} \leftarrow h_{i-1}^{(j)} \times \alpha_i^{(j)(k)}$
5: **end for**
6: Initialize projection layer $V_i^{(k:j)} \leftarrow Dense(input = h_{(i-1)}^{(<k)}, numOutputs = n_{(i-1)})$
7: Activate projection layer $V_i^{(k:j)} \leftarrow Activation(V_i^{(k:j)})$
8: Initialize lateral connection $U_i^{(k:j)}$ as a dense layer without bias $U_i^{(k:j)} \leftarrow Dense(input = V_i^{(k:j)}, numOutputs = n_i^{(k)}, excludeBias = True)$
9: return $U_i^{(k:j)}$
10: **end procedure**

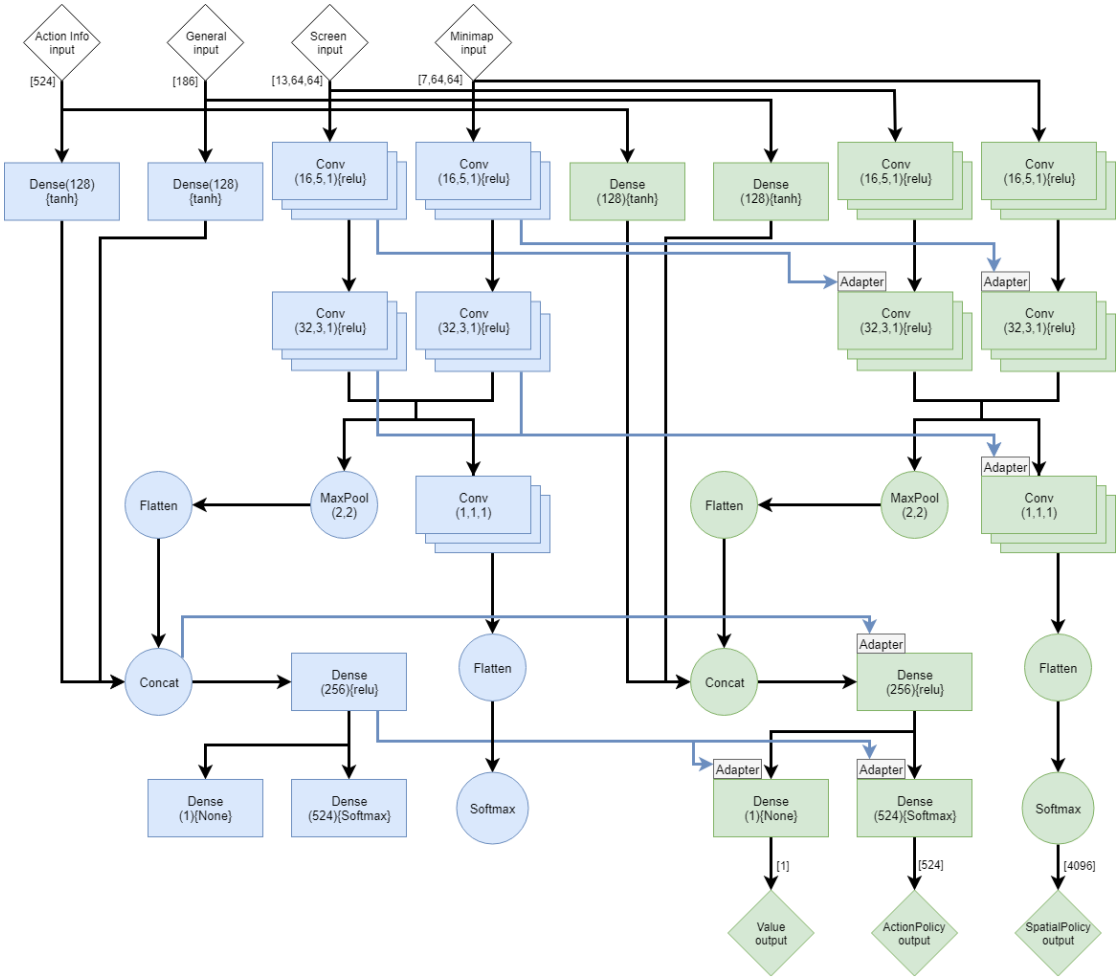


Figure 3.8: Visual representation of our progressive StarCraft II network with two columns

The progressive network paper[1] does not explicitly state how to proceed if a layer receives input from both convolutional and fully connected layers. If we look at the architecture of our Starcraft II network for the A3C agent with two columns, which can be seen on Figure 3.26 on page 63, we can see that the dense layer preceding our value and action policy output layer receives input from both a convolutional and a fully connected layer. A three column example of our network architecture can be seen on Figure A.1 on page 75. We handle this case as seen on the psuedo code Algorithm 4 on the facing page. Just like when we only have fully connected input, we start by multiplying the input activations with seperate adapter scalars. Then we split the inputs $h_{(i-1)}^{(<k)}$ into two groups, one containing all input activation maps received from convolutional layers $C_{h_{(i-1)}^{(<k)}}$ and one containing input activations received from fully connected layers $F_{h_{(i-1)}^{(<k)}}$. Instead of having one projection layer $V_i^{(k:j)}$ we instantiate one for convolutional inputs $C_{V_i^{(k:j)}}$ and one for fully connected inputs $F_{V_i^{(k:j)}}$. Both of these projection layers are activated separately with a non-linear activation function and the convolutional projection layer is flattened, before concatenating the activations of $C_{V_i^{(k:j)}}$ and $F_{V_i^{(k:j)}}$, giving us

$V_i^{(k:j)}$. We then initialize our linear lateral connection $U_i^{k:j}$ giving the activations of $V_i^{(k:j)}$ as input.

Algorithm 4 Adapter for fully connected layer with both convolutional and fully connected input

procedure ADAPTER
  for $h_{(i-1)}^{(j)}$ in $\{h_{(i-1)}^{(1)} \dots h_{(i-1)}^{(k-1)}\}$ **do**
    Instantiate adapter scalar for the input $h_{(i-1)}^{(j)}$ to column k, $\alpha_i^{(j)(k)} \leftarrow random(0.01, 0.05)$
    Multiply input hiddens with adapter scalar $h_{i-1}^{(j)} \leftarrow h_{i-1}^{(j)} \times \alpha_i^{(j)(k)}$
  end for
  Split input hiddens $h_{(i-1)}^{(<k)}$ into convolutional $C_{h_{(i-1)}^{(<k)}}$ and fully connected input $F_{h_{(i-1)}^{(<k)}}$
  Initialize convolutional projection layer $C_{V_i^{(k:j)}} \leftarrow Conv(input = C_{h_{(i-1)}^{(<k)}}, numOutputs = C_{n_{(i-1)}}, kernelSize = 1, stride = 1)$
  Activate convolutional projection layer $C_{V_i^{(k:j)}} \leftarrow Activation(C_{V_i^{(k:j)}})$
  Initialize dense projection layer $F_{V_i^{(k:j)}} \leftarrow Dense(input = F_{h_{(i-1)}^{(<k)}}, numOutputs = F_{n_{(i-1)}})$
  Activate dense projection layer $F_{V_i^{(k:j)}} \leftarrow Activation(F_{V_i^{(k:j)}})$
  Flatten convolutional projection layer $C_{V_i^{(k:j)}} \leftarrow Flatten(C_{V_i^{(k:j)}})$
  Concatenate output of convolutional and dense projection layer $V_i^{(k:j)} \leftarrow concat(C_{V_i^{(k:j)}}, F_{V_i^{(k:j)}})$
  Initialize lateral connection $U_i^{(k:j)}$ as a dense layer without bias $U_i^{(k:j)} \leftarrow Dense(input = V_i^{(k:j)}, numOutputs = n_i, excludeBias = True)$
  return $U_i^{(k:j)}$
end procedure

3.3.1 ALS Implementation

This section describes technicalities of the implementation of ALS, that had to be overcome in order to implement it using tensorflow. The section does not contribute new relevant knowledge w.r.t. ALS. The implementation was mostly straight forward, however tensorflow did complicate the implementation, and that is what this section is about.

We decided to train an agent's model normally, then save the model and transfer it into another agent made for calculating ALS. The ALS agent does not train, but acts according to the model's policy to get the estimated state distribution needed to calculate ALS. The ALS agent has one change to the network; all the layers have softmax activations, since this gives us the normalized activations that we need in $\hat{h}_{(i)}^{(k)}$. otherwise the ALS agent and the agent used to train the model, are identical, only change is that ALS is calculated for each step, and that the agent does not call any training function.

Tensorflow's functions and graph creation for optimizing the process was not easy to understand and work with. The code for calculating ALS could not be debugged at run-time, only when the environment was setting up and the graph created. We encountered multiple other problematics, but the most impactful where using conventional control structures with tensorflow's internal graph. Even though the graph had multiple complications some very simple, for example just handling variables and operations in the graph, we will only describe how we handled the

control structures.

Tensorflow graph problematics and optimizations
The first implementation of ALS revealed a problem. While the first couple of ALS values were a success, it was only tested on a network with less than 50 parameters in the model. When the first ALS implementation was tested on one of our Starcraft II models with hundreds of thousands of parameters, it took several hours to create parts of the internal graph, before crashing due to memory limitations.

Tensorflow creates a graph of all the code that is used for interacting with the neural network model, and all training and ALS calculation is done through this graph. The graph is a smart way of constructing the code, as it maps dataflow in the code, and this gives the possibility for distributed computation, parallelism, and portability. The graph is also optimizing the processes by only computing the parts of the graph needed for the output the surrounding code needs from the graph. e.g. if the surrounding code wants the location output, the graph only go through the nodes in the graph that are strictly necessary for computing that output, this cuts off computation of loss function, gradients and the rest of the neural network, even though the graph also includes these.

Nodes in the graph describe operations, and edges between them describe the dataflow. Variables that persists through multiple runs of the graph can be created, used and manipulated through operations in the graph. One problem with this graph is that it cannot optimize on conventional control structures like if-statements or loops. An if-statement is constructed at compile-time, and will impact the structure of the graph rather than create an edge in the graph. When loops are created in the graph, they also impact the structure of the graph in the same manner, constructing the graph as if the loop was unrolled instead of described in a loop. This can potentially make large graphs, that take a lot of memory and time to create. An example of this is creating one node for each gradient needed to compute ALS for each layer in the network..

Instead of the conditional if-statement, tensorflow graphs use a function called *cond* which takes one boolean, followed by a function that is to be called if the boolean is true, and another function that is called if the boolean is false. An example of this can be seen in Algorithm 5 on the next page on line 3 where the function is called. The graph used by tensorflow limits the possibilities with these functions; they cannot change variables in the graph and they need to return tensors of the same shape. The reason for this is that when the graphs are constructed, the shapes/sizes of all edges is accounted for.

Since we want to be able to decide whether we want to calculate or not calculate ALS, we need to return the same shaped tensor even if we don't calculate the ALS. This led us to simply return the input array as it had the same shape as the *ALS*, as can be seen on Algorithm 5 on the facing page. This is useful if we want to calculate the ALS for each third step the agent takes, since experiences are sequential, thus close states does not change much, this might be a faster way of estimating the state distribution.

The loop control structure used in Algorithm 5 on the next page is inefficient, as previously explained, they can potentially create a huge amount of nodes and thus use up huge amounts of memory and time. Instead tensorflow has a function called *While_loop* that takes a list of all variables, a predicate that returns true if the loop should continue, and a loop body. An example of the *While_loop* can be seen on Algorithm 6 on the facing page. When constructing the graph, tensorflow will determine if the predicate can be pre-calculated and thus determine if computations can be parallelized. This function creates a single node in the graph and reuses that, instead of creating multiple nodes. This loop is also optimized by tensorflow to incorporate parallelism and calculation reuse.

Algorithm 5 Tensorflow graph conditional control structure

```
1: CalculateALSBool = True
2: WantedLayers = [0, 0][1, 1] // We want layer both 0 and layer 1 in 2 columns
3: ALS = tf.cond(tf.equal(CalculateALSBool, tf.constant(True)), CalculateALS(WantedLayers),
   DoNothing(WantedLayers))
4:
5: procedure CalculateALS(WantedLayers)
6:     for each layer ∈ WantedLayers do
7:         ALS = CalcLayerALS(layer)
8:     end for
9:     Return ALS
10: end procedure
11:
12: procedure DoNothing(WantedLayers)
13:     Return WantedLayers
14: end procedure
```

Algorithm 6 Tensorflow whileloop

```
1: procedure CalculateALS(WantedLayers)
2:     loop_Vars = [tf.variable(0, int), tf.TensorArray(size=WantedLayers.length(), float)]
3:     cond = lambda i, a: i < WantedLayers.length()
4:     body = lambda i, a: (i + 1, a.write(CalcLayerALS(WantedLayers[i])))
5:     i, ALSList = tf.while_loop(cond, body, loop_vars)
6:     return ALSList
7: end procedure
```

We needed to use the *while_loop* shown in Algorithm 6. With the *while_loop* the graph takes under a minute and virtually no additional memory, as opposed to several hours and crashing in need of more memory.

3.4 Tests and Findings

In this section we will test our progressive networks, and we seek to determine if transfer learning can be used in a complex environment such as Starcraft II. Before testing the progressive network on Starcraft II we verify that the implementation is functional, and that the progressive networks leverages transferred knowledge. This is done through simpler environments, namely the Gym library's CartPole[3] and Sonic the Hedgehog[4] environments.

3.4.1 Proof of concept

For proof of concept we will be using environments with less complexity than the Starcraft II environment. The reason for this is because the reward graphs of Starcraft II does not converge in a reliable way, however in the less complex environment, the convergence is reliable and smooth, making it possible to compare differences in the agents' performance, from a graph of smoothed average reward.

We use the Gym library for these tests, which is a toolkit for developing and comparing reinforcement learning algorithms. This library allows us to test the algorithms on multiple different environments with very little reconfiguration.

In this section we will describe the two environments, CartPole and Sonic, show that progressive network architecture gives an improvement in convergence, and analyze the results with an ALS analysis.

CartPole

CartPole is a simple game with two actions, move left or right. The environment returns four numbers, which describe the pole's positioning. The pole is always on top of the cart, so when moving the cart either left or right, the lowest point of the pole is also moved in that direction. The cart is fixed vertically, so that it can only move horizontally, the pole on top however is affected by some gravity physics that makes the pole fall downwards with the upper most end of the pole. An image of the CartPole environment can be seen on Figure 3.9.

The CartPole game does not have any spatial features, since the statespace is described by four float values, this makes the CartPole agent use a simple network, consisting of two fully connected hidden layers with 24 and 48 nodes.

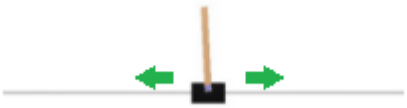


Figure 3.9: The CartPole environment, keep the pole affected by gravity from falling by moving the cart underneath it at all times, actions include move left and right, denoted with green arrows.

The test consist of three different agents that have all been run for 32 million steps in the same task. We chose to use the same base task for all our agents, as the transfer of knowledge is always greatest when the source task is the same as the target task. Progressive networks start with randomly initialized column for the target task, and do not immediately use the previous column(s), so it will not be good at the game instantly. On the other hand, finetuning only has to relearn the output layer to perform well, so those agents should improve quicker than the other agents.

The agents are using the A3C algorithm, each running with 16 workers. The three agents have different amounts of columns in the network, where the 1-column agent is essentially just an A3C agent. The 2-column agent was created by using the converged network from the 1-column test as the first column, and similarly for the 3-column agent which uses the converged 2-column agent's network for the first and second columns.

On Figure 3.10 on the next page the results can be seen for the test. The reward in CartPole is equivalent to the number of steps the pole has not fallen, maxed out at 200 steps after which the episode resets. The graphs show the mean score after each episode, that the agents with the 1, 2 and 3-columns gained through 32 milion game steps. To show a more detailed view of the results, they are also plotted on Figure 3.11 on the facing page, with 2 slim lines and a shaded area between them, showing rewards between first and third quartile for each network setup over 5 runs.

It is clear that on average, Figure 3.10, the baseline A3C agent (1-column) is slightly worse than the progressive network transfer learning agents (2 and 3-columns). The transfer learning agents are relatively slow at learning in the beginning, but can converge to a better policy than the 1-column agents. The tests clearly show that rewards are higher when using our implementation of progressive networks, with all columns trained on the same simple game.

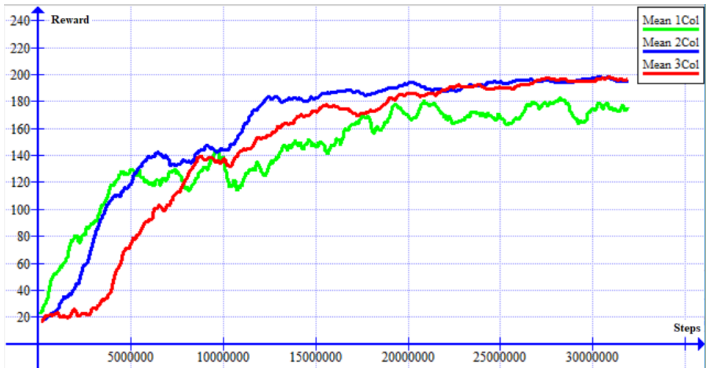


Figure 3.10: Graph showing mean reward as a function of steps on the CartPole game, for 1, 2 and 3-column networks

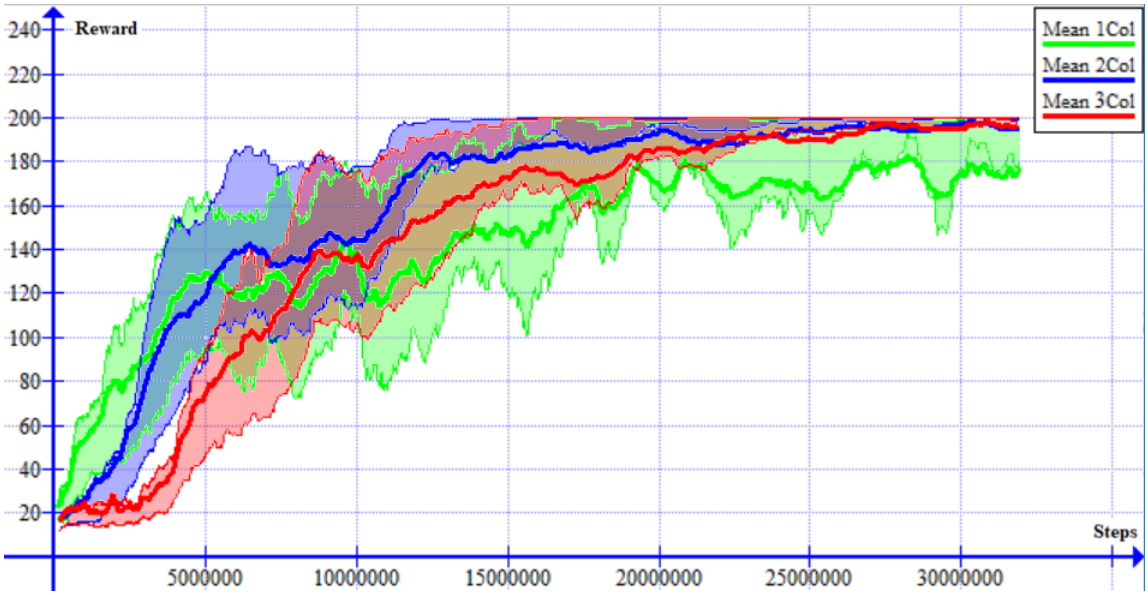


Figure 3.11: Graph showing reward graphs for 1, 2 and 3-column progressive networks, bold lines represents means, and coloured areal is the first to third quartils of all runs with the respective network

These results suggest that we leverage previously trained networks well, since we can see better convergence across all 2-3-column tests. The previously trained columns are trained on the same task, so the columns should be highly usable. We have tested the 1, 2 and 3-column networks 5 times each, and they all suggest that the 2-column network works better than 1 and 3-column networks for the CartPole game. This indicates that we can leverage previously learned knowledge, when a column is trained on the same task ensuring knowledge can be

leveraged.

The slower convergence right at the start of the 2 and 3-column networks could possibly be due to the increased amount of trainable parameters. This could also be the reason why the 2-column networks usually performs better than 3-column networks in this case. The 3-column networks becomes as good as the 2-column networks, but they are slower at the start.

There are some 2-column tests that differ slightly in their convergence speed. It would be interesting to know why there is a difference between them, we will attempt to determine why during the ALS analysis in Section 3.4.1 on page 54.

Sonic

Sonic is a more complex game from the Genesis console with 9 discrete actions, including movement and jump. This environment returns pixel information in three channels(RGB) to the agent, from a screen of size 224x320, giving us an input of size 224x320x3. We preprocess the screen input by converting the image to grayscale and resizing it to 92x92 pixels, giving us an input of 92x92x1. This environment is a platform game where the player controls a character(Sonic) and has to make it to the end of the map by moving right while avoiding or destroying obstacles on the way. The agent is rewarded with +100 whenever it destroys an enemy, and also receives +1 reward for each pixel it moves Sonic towards the goal. A screenshot of the game can be seen of Figure 3.12.

Figure 3.12: The SpringYardZone level of the Sonic environment

The emulator running Sonic the Hedgehog is not able to run multiple environments at once, hence the A3C algorithm is not optimal for testing on this game as we are not able to use multiple workers to get diverse experience. We can however still test our Progressive Network library using another agent. For this we used an open source DQN algorithm[25], which is better when limited to a single environment as it trains on diverse experience by using experience replay[26].

Figure 3.13: The GreenHillZone level of the Sonic environment

We start by training the DQN agent with 1-column on act 1 of the SpringYardZone level shown on Figure 3.12 on page 52 followed by training the 2-column on the first act of the GreenHillZone level shown on Figure 3.13, using the SpringYardZone 1-column as the first column. These two levels have a huge visual gap between them while some of the core elements like the controlled character stay the same. The two different levels also incorporate different mechanics required to complete them. The results of doing this can be seen on Figure 3.14.

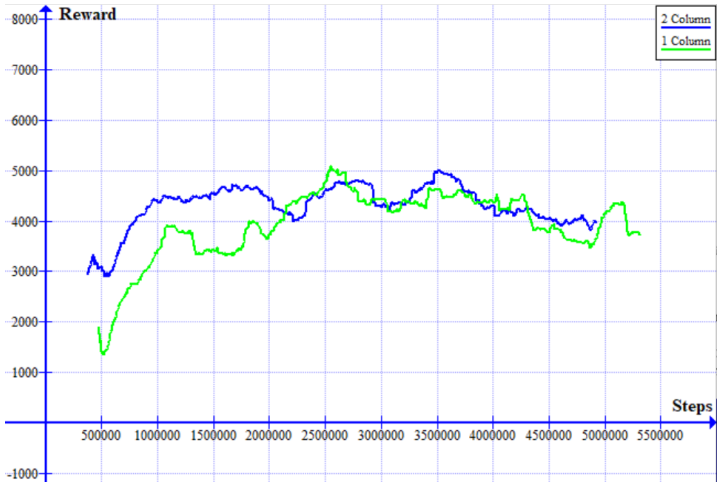


Figure 3.14: Graph showing moving average of the reward as a function of game steps on GreenHillZone act 1. The green line is a 1-column network. The blue line is a 2-column network, where the first column has been trained on SpringYardZone act 1

For the next test we train on act 3 of the SpringYardZone, using a 1 column network, and a 3 column network, which can be seen on Figure 3.15 on the next page. The 3 column network uses the previous 2-column network for the first 2 columns.

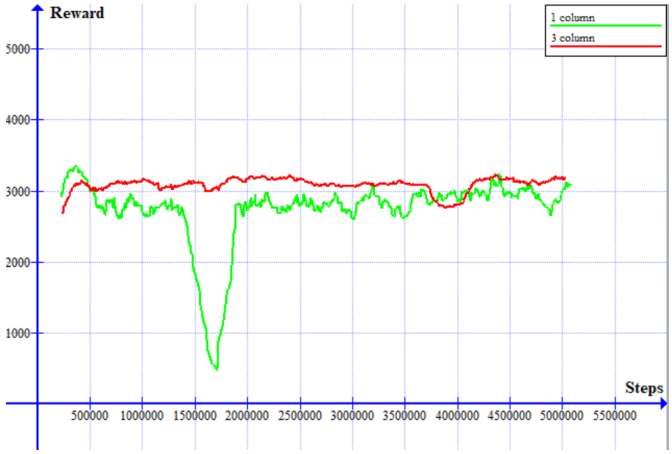


Figure 3.15: Graph showing moving average of the reward as a function of game steps on SpringYardZone act 3. The green line is a 1-column network. The red line is a 3-column network pre-trained on GreenHillZone act 1 and SpringYardZone act 1

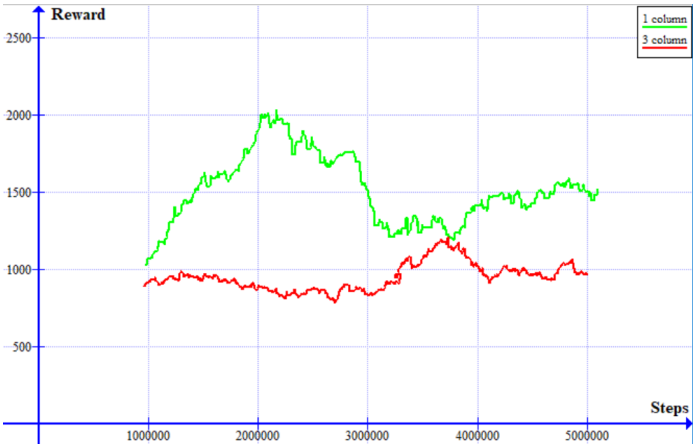


Figure 3.16: Graph showing moving average of the reward as a function of game steps on GreenHillZone act 3. The green line is a 1-column network. The red line is a 3-column network pre-trained on GreenHillZone act 1 and SpringYardZone act 1

For both of these tests the multiple column networks seem to have an edge, having a slightly higher moving average throughout most of the training period on both tests. For the test shown on Figure 3.14 on page 53 the two column network seems to be learning a bit faster than the 1 column network, however since we do not have more tests than are shown, all of this could also be due to pure coincidence. We do not know at this point if the agent is actually able to use the progressive convolutional layers, only that they do not break the agent. We can instead use ALS analysis to find out if the agent actually leverages knowledge from the previous columns.

ALS analysis

The ALS analysis is in two parts, CartPole where we seek to verify the ALS score itself and verify transfer is happening, and Sonic where we will analyze the network to find out to which

extent previously trained models are leveraged in different environments. Sonic was chosen for network analysis because the network was similar to Starcraft II, since it has image based input, and that we can train on different levels and see transfer to similar but different tasks. The analysis seeks to prove that ALS scores gives a meaningful representation of the level of transfer happening between columns, and that progressive networks are capable of leveraging transfer of knowledge in an image based game such as Sonic.

CartPole
We chose to use the agents from the CartPole game to verify the ALS score, since the reward curves from the game through learning are smooth and consistent, and convergence happen within an hour of training. It would require much more data and time to make these tests on one of the Starcraft II mini-games, since the convergence is very spiky, slow, and inconsistent, as discussed in Section 2.4.3 on page 32.

The first test done to verify the ALS score, was to compare the ALS score with the actual rewards earned while one layer is missing. This test is meant to verify that high ALS scores translate into a reduction in rewards if the layer was taken out, and vice versa if the ALS score is low.

We first calculate the ALS score for the neural network. Calculating the ALS score is only done once, to determine the output's dependence on a layer.

To verify the ALS scores, we compare ALS scores to an observed average drop in reward when removing a layer present in the neural network. We modify the output of a layer, by replacing the activations with very small random numbers, to remove any correlation between the modified output and the input.

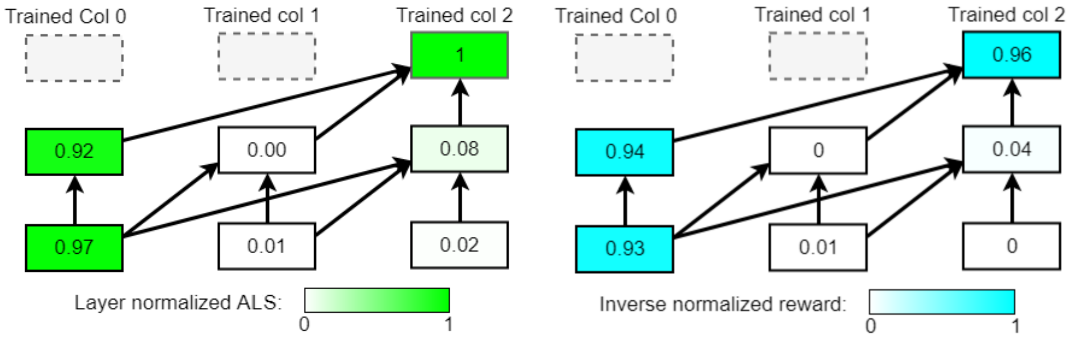
After removing a layer, we run the agent for 200 episodes, to calculate a ratio of drop in reward, we call it the inverse normalized reward. We repeat this process for each layer in the network, so we know how much the absence of each layer will influence the average reward. This allows us to depict the impact of removing the layer, by showing the inverse normalized reward for each layer, similar to the ALS score that depicts the output's dependence on each layer.

It is obvious that calculating the inverse normalized reward for each layer is tedious compared to the ALS score, as the agent has to be started, run to get an average reward sample, stopped, and reprogrammed for each layer in the neural network, instead of once for the ALS score.

We make this test to validate what would happen if layers are removed. If the ALS score depicts how much the output depends on a specific layer, we expect to see the reward go down for a layer that the output somewhat depends on. If a layer has a very low AFS score, the policy does not depend on the contributions of that layer and it should have very limited impact on the reward if it is removed.

The inverse normalized reward for each layer can be seen on Figure 3.17b on the following page. At each layer in the network, the inverse normalized rewards are stated. The colouring are for swift overview and easy comparison, the higher the number the more colour. A score of 0 is equivalent to no change in the reward achieved before and after the layer is removed, and a score of 1 is a reduction in average reward per episode to 0.

The Calculated ALS scores can be seen on Figure 3.17a on the next page, where the ALS scores are positioned in their respective layer. It is obvious that there are similarities in the two scores, indicating that the ALS score represents an accurate estimate of the sensitivity of the output w.r.t. each layer. Removing a layer with a high ALS score gives a lower reward, and removing a layer with close to 0 ALS score gives virtually no change in reward.



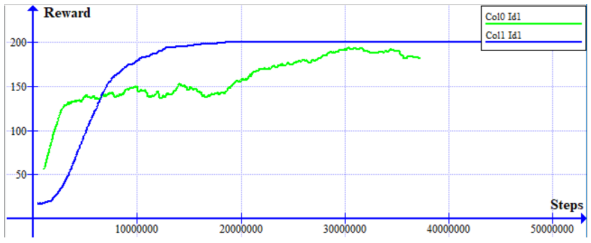
(a) Shows the sensitivity in the output based on change in one layer, normalized over layers in all columns.(ALS)

(b) Shows the drop of reward over the maximum theoretical reward, when replacing the respective layer with noise.

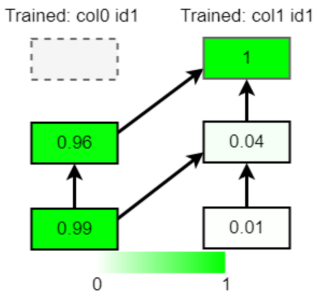
Figure 3.17: ALS and simple layer dependence.

The second test on the CartPole game, compares the learning graphs with the ALS scores, as we argued in section Section 3.4.1 on page 50, if the reward graphs consistently converge in a certain way, while comparing tests of various progressive networks, it has to correlate to the changes in the network and thereby the previously gained knowledge. This implies that if the 2-column networks generally get better rewards and faster convergence, the 2-column progressive networks must leverage the first column to some extent. If that is true, our hypothesis is that the ALS analysis must show usage of the first column in the 2-column networks on the CartPole task, because in the CartPole task we observe both faster convergence and convergence to a higher average when using multiple columns. The single column tests all show similar convergence trends, so we hypothesize that if a 2-column network does not leverage knowledge from the previous column, it should have a similar convergence trend to a 1-column network.

We take one of our 2-column agents and calculate each layer's ALS score, with results for one of the 2-column test shown on Figure 3.18, where the reward graph for the specific run of the 2-column network are shown on Figure 3.18a. This 2-column network has the 1-column network, displayed as col0 in the reward graph, as the first column. The ALS analysis of the 2-column network is shown on Figure 3.18b.



(a) Agents' earned rewards(y) at game step/time(x). Green line one run of a 1-column network. Blue agent is one run of a 2-column network.



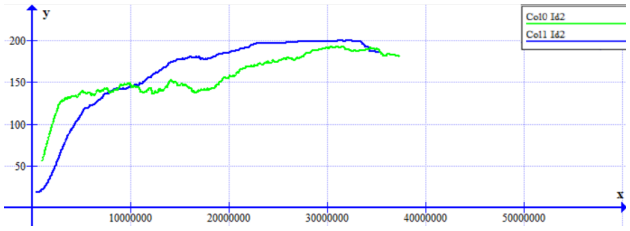
(b) ALS figures over the plotted 2-column progressive network.

Figure 3.18: Progressive network analysis example 1, a closer look into one of the runs of the 2-column agent.

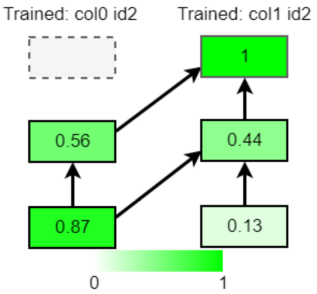
Our hypothesis seems to be validated; we get high usage of the previous column and when

the reward graph for the 2-column network shows faster convergence to a higher average. The reason why an agent that purely uses another column can still perform better than an agent with only that one column, could be because it has vertical connections with fully connected layers, which can make interpretations on the previously learned features, meaning that it can add knowledge through these connections and not only through the newest column.

Another interesting 2-column network to analyze, is displayed on Figure 3.19, in this run the reward graph of the 2-column network is more similar to the 1-column network. As mentioned above, we hypothesised that this could be due to the new column of the 2-column network relearning the features of the game, rather than leveraging the knowledge of the previous column. Figure 3.19b displays the ALS analysis.



(a) Agents earned rewards(y) at game step/time(x). green agent is only one column. Blue agent is 2-columns(with col 0 has frozen parameters from the green agent).



(b) ALS figures over the plotted 2-column progressive network.

Figure 3.19: Progressive network analysis example 2, a closer look into one of the worst runs of the 2-column agent.

The ALS analysis shows that there is a lower degree of dependence of the previous column in this test, compared to the previous 2-column ALS analysis, but the network still leverages knowledge from the first column. This corresponds well to the faster convergence to a higher average reward per episode, although the convergence rate is much slower than in the previous example from Figure 3.18 on page 56.

The ALS analysis has been performed with the model extracted from the last possible part of the graph, meaning that even after the 2 column has run on max score for a while, the progressive network still leverages the previously learned knowledge instead of rewriting it all.

Sonic

We perform the ALS analysis on our two column network to analyze how the progressive neural networks leverages the previously learned data in Sonic. We want to confirm that Progressive networks can leverage knowledge from convolutional layers, since we could not prove that in the CartPole test. The Sonic test should also conclude if it is possible to leverage knowledge gained from another task that is similar but different from the target task.

The first Sonic ALS analysis is performed on a 2-column network trained on act 1 of the GreenHillZone, with the first column being pre-trained on act 1 of the SpringYardZone. We analyze the trained agent at three different points during the training of the 2-column agent, we call each point a slice. The slice is an ALS analysis performed on the network using a model extracted from a specific point during training, the slice analysis takes multiple slices to see how the usage of transferred knowledge changes through training. On Figure 3.20 on the next page the three slices we perform ALS analysis on can be seen, marked with vertical lines of different color corresponding to the color of the figures shown in Figure 3.21 on the following page.

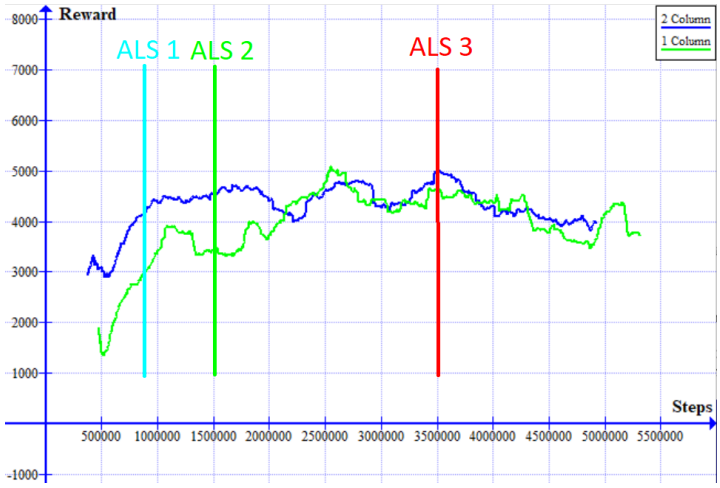


Figure 3.20: Figure showing points at which ALS is calculated for a 2 column network on GreenHillZone act 1

Looking at Figure 3.21 we can see that at the first point the agent only leverages the two first convolutional layers of the first column in the network, and has learned to do even more so at the second point. Then at the third point the agent has replaced most of the knowledge from the second layer of the first column with new features. This suggests that the agent starts relearning features because it deems it better than to use the old features of the first column.

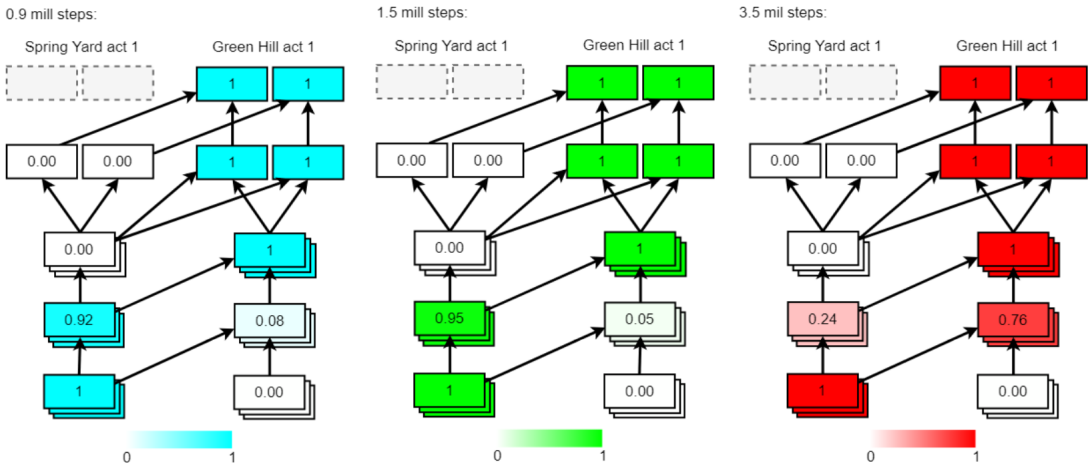


Figure 3.21: ALS figures, showing 3 different points in the sonic agents training period, 0.9M, 1.5M and 3.5M. The input layer is omitted, only convolutional and dense layers are shown. The reward graph over the training period can be seen on Figure 3.20

We also performed another ALS analysis on the second 3 column agent, which can be seen on Figure 3.22 on the facing page, and the analysis was performed after the agent finished training. The 3-column agent trained on Green Hill act 3 used the 2-column agent as the two first columns. The graph over the rewards gained while training can be seen on Figure 3.16 on page 54, in general the 3-column agent converges fast to a stable average reward, with performance worse than that of a 1-column network on the same Sonic act.

The analysis shows that a high amount of transfer of knowledge between levels and acts does not guarantee the agent equal or better performance compared to a 1-column agent. This could also explain why the 2-column slice analysis show that the agent slowly begins to learn its own features instead of persistently using the transferred knowledge. It is plausible that the agents yield lower rewards and slowly ignore transferred knowledge, because the features transferred are generally weak. In the start it could be slightly easier to learn meaningful features by leveraging transferred knowledge, than by improving the randomized features in the new column, but the transferred knowledge utilized in the start may prove to be bad or unreliable for the new task in general.

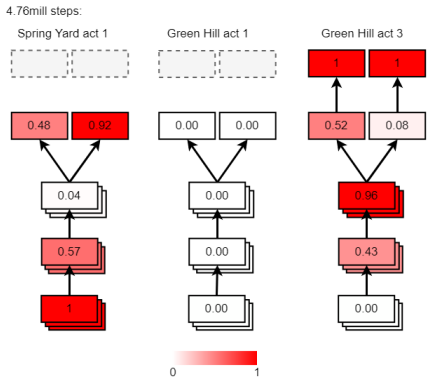


Figure 3.22: ALS figures at the final training point for the 3-column network trained on GreenHillZone act 3, the pre-trained columns are the 2-column Sonic agent. Connections between columns are omitted, though the layers are still connected, as normal. The reward graph for this 3-column network can be seen at Figure 3.16 on page 54.

The ALS analysis of the Sonic game shows that the 2-column agent had low amounts of knowledge transfer compared to the ALS analysis on the CartPole game, as seen on Figure 3.21 on page 58. The 3-column agent had, after the complete training period, a high amount of transferred knowledge. The lower knowledge transfer on the 2-column agent could be due to our reinforcement learning algorithm's overall level of performance on the games. At the last point in the learning process, the agent sometimes ran directly into the first obstacle and died, even though it is the simplest of obstacles that it has encountered the most. This could mean that the agent's learned features are not good enough for stable and useful transfer, driving the agent to leverage less and less of the previous column's learned features, as the agent learns what it decides are better features.

The ALS analysis for Sonic proves that an agent can leverage knowledge from progressive convolutional layers, and because the previously learned knowledge is generated from another level in Sonic, it also proves that we can leverage knowledge from similar but different tasks. Furthermore the 2-column test could indicate that the agent had faster convergence because of the transfer of knowledge, since the agent converges faster and utilizes the previously learned knowledge, however the 3-column agent that leverages the transferred knowledge and performs poorly, so it is not always the case that the agent achieves faster convergence or better results even when leveraging transferred knowledge.

Test Conclusion

The main purpose of this test was to prove that our progressive network and ALS implementation was functional, which was done using the two games Cartpole[3] and Sonic The

Hedgehog[4].

The network for the Cartpole game only makes use of dense layers, and its input only consists of non-spatial features, and the strategy required to master the game is relatively simple. The tests on the game show us that we can leverage information gained from the same task, when using only dense layers in the progressive network. The potential of transfer is greatest when the first column is trained on the same task as the target task, allowing almost complete usage of the first column.

We can also conclude that the implementation of the ALS score gives meaningful results, allowing us to analyze if there is a transfer of knowledge in our progressive network agents that utilize multiple columns. In this test the progressive networks utilize the previously learned features even after converging. The results show when we utilize transfer learning, the convergence is happening faster, and it can result in even better performance than agents trained without transfer learning. This suggests that the use of progressive networks allows for faster convergence to a higher level of performance by using the previously learned knowledge, and at the last point in the training, the progressive networks are still leveraging some transferred knowledge.

The Sonic game requires a much more complicated strategy to master while the input consists only of pixel information, and our network is a combination of convolutional and dense layers. This game allowed us to test our progressive convolutional layers and transfer between similar but different tasks from a more complex environment compared to Cartpole. Based on our findings for these two tests, we can conclude that our progressive network implementation is in fact able to leverage knowledge from previous columns and in most cases it improves performance, even when transferring knowledge from other tasks with different visuals. However in Sonic the results were only improved two out of three times, and the the progressive networks one time slowly learn to leverage less and less of the transferred knowledge, as documented on Figure 3.21 on page 58, where ALS is calculated from different points in the training. This trend suggests that the progressive network leverages the knowledge to get faster learning in the start, but does not deem much of the knowledge usable enough to refrain from relearning features, however more analysis is needed to conclude anything definitively. The progressive networks are not guaranteed to improve performance, and more tests are needed to conclude on whether the transfer learning agents are able to achieve a higher performance than 1-column agents in games other than CartPole, as more tests may prove there are agents with other hyperparameters that perform much better which may also allow for more useful transfer of knowledge.

3.4.2 StarCraft Test

Based on our tests for Cartpole in Section 3.4.1 on page 50 and Sonic in Section 3.4.1 on page 52, we know that the A3C agent is able to utilize both our progressive dense layers and our progressive convolutional layers. In this section we will test our implementation of progressive networks on Starcraft II. The goal is to determine if we can transfer knowledge in a complex environment and potentially achieve higher rewards or faster learning. The tests are all performed with a specific set of hyperparameters which we found made the agents a bit more stable than in the tests shown in Section 2.4 on page 27, like having two different entropy's, one for the action policy and one that is slightly lower for the location policy.

For this test we use three different Starcraft II minigames, including DefeatRoaches, FindAndDefeatZerglings which are described in Section 2.4.1 on page 27, and DefeatZerglingsAndBanelings

which is similar to DefeatRoaches except that there are two different enemy types, which require different tactics to defeat.

Due to time constraints all agents are tested only once with the optimal hyperparameters found in Section 2.4 on page 27. This means that we will not be able to tell necessarily if using more columns is an improvement in general, but we should still be able to find out if we are able to transfer knowledge using ALS analysis. Again the learning rate of the agents is decayed to half its initial value during the entire training period, and we use 8 workers for the test. We train the agents for approximately 50 million steps, using RMSProp optimizer.

Proof of concept

We start by making a proof of concept test on Starcraft II, which is depicted on Figure 3.23. Here we have training graphs showing moving average reward over 100 episodes on the DefeatRoaches minigame, where we have a 1-column network and a 2-column network where the first column is also trained on the DefeatRoaches minigame. This test is similar to the tests performed on CartPole, to determine if transfer is happening when the target task is the same as the source task. This 2-column network should be able to use the features learned in the first column as it is the same minigame, assuming that the transfer of knowledge is possible in Starcraft II.

Figure 3.23: Graph showing moving average reward over 100 episode as a function of steps on the DefeatRoaches minigame, for a 1 and 2-column network pre-trained on the same minigame.

Figure 3.23 shows slightly faster learning for the 2-column network at the beginning of the training period, but starts lacking behind the 1-column network after a fairly short time. In order to investigate what is happening and figuring out if transfer is occurring, we perform an ALS slice analysis of the model of the 2-column network at three different slices on the graph. The three chosen slices where we perform the ALS analysis are depicted on Figure 3.23, and the ALS analysis itself is depicted on Figure 3.24 on the following page. Looking at the ALS analysis we can see that the first slice at 6 million steps leverages some knowledge from the first column, and at 13 million steps it leverages the knowledge of the first column even more. If we look at the third slice it uses slightly more of the first column.

This shows us that it is possible to transfer knowledge on the Starcraft II environment via progressive networks, even though it does not necessarily improve the performance of the agent if we measure by highest moving average reward over 100 episodes.

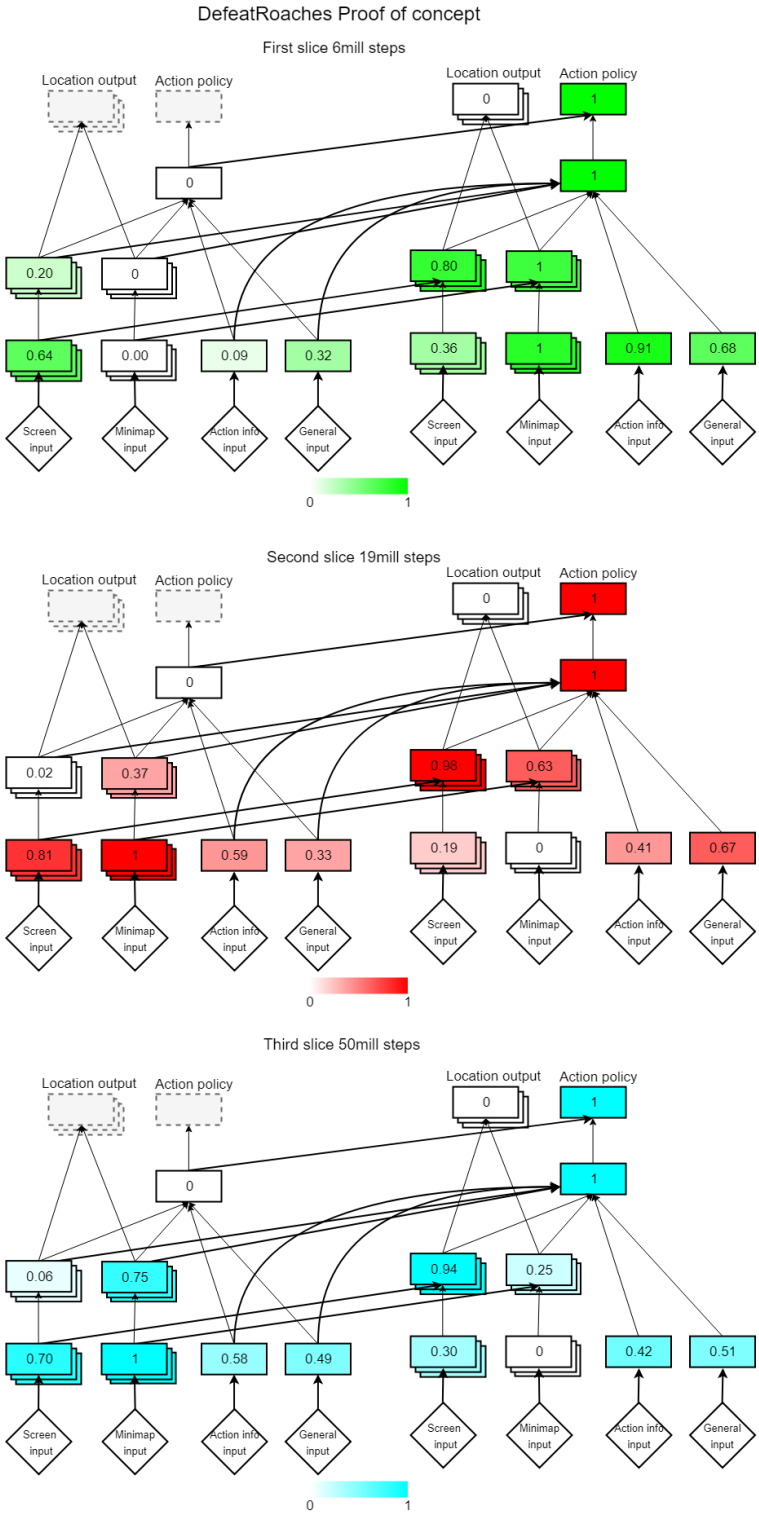


Figure 3.24: ALS figures over the proof of concept in Starcraft II, the 2-column agents training graph with slice indicators can be seen at Figure 3.23 on page 61

DefeatZerglingsAndBanelings test

The first minigame was tested on DefeatZerglingsAndBanelings minigame, and is shown on Figure 3.25. Here we have a 1-column network, and a 2-column network which has its first column trained on the DefeatRoaches minigame.

Figure 3.25: Graph showing moving average reward over 100 episode as a function of steps on the DefeatZerglingsAndBanelings minigame, for a random policy agent and a 1-column and a 2-column network with its first column trained on the DefeatRoaches minigame.

We performed an ALS analysis on the 2-column network seen in Figure 3.25. The ALS analysis can be seen on Figure 3.26.

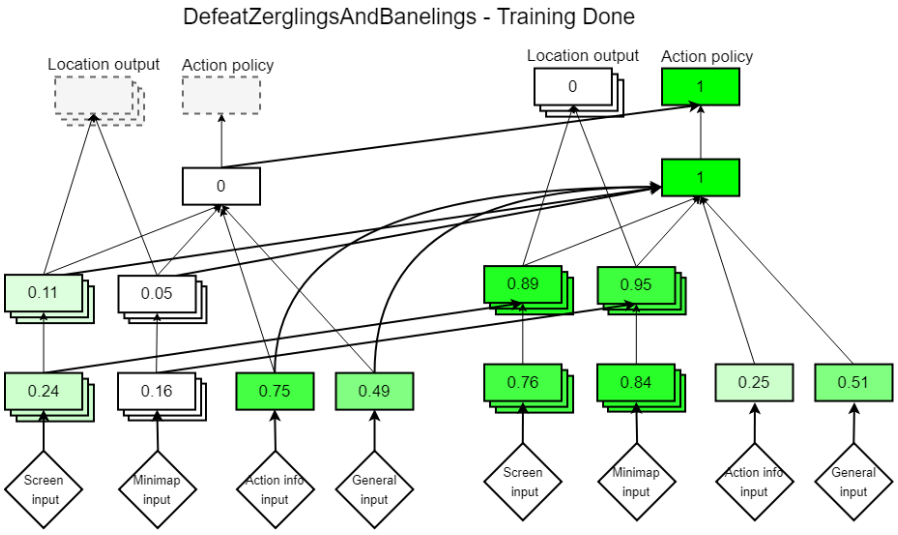


Figure 3.26: ALS figures for the 2-column progressive network after training, the training run is plotted on Figure 3.25. Tested on DefeatZerglingsAndBanelings and first column trained on DefeatRoaches

The ALS analysis shows little transfer of knowledge in the convolutional layers, and a larger amount of transfer in the non-spatial features.

We perform an in depth analysis of the transfer of knowledge happening, and we try to relate the transfer to the different minigames' mechanics and how they relate to each other: The features that can be extracted from the minimap input are not very relevant for this minigame, so it makes sense that it does not rely on this knowledge from the first column. The features extracted from the screen input are more relevant, as it contains information relevant to combat, so here it makes sense that more of this knowledge is leveraged. The strategies required in the two minigames is also quite a bit different. On the DefeatRoaches minigame it is important to focus attack specific units in order to reduce the damage output of the enemies as fast as possible, while it is much more important to split up the friendly units into small groups to avoid area damage from specific enemy units in the DefeatZerglingsAndBanelings minigame.

The action info input and the general input can be used to determine if a unit is selected, which unit is selected, and which actions are available. The reason that these are leveraged heavily may be due to the importance of knowing which unit, or if a unit, is selected in Starcraft II. If a unit is selected, the next action should probably be some kind of move or attack action, especially on these minigames where there is little need to reselect units. This is relevant because the DefeatZerglingsAndBanelings and DefeatRoaches scenarios have similar mechanics where the agent has to move around its own units and attack the enemy.

FindAndDefeatZerglings test

The second minigame we performed tests on in Starcraft II is shown on Figure 3.27 and depicts the FindAndDefeatZerglings minigame. Here we have a 1-column network, a 2-column network with its first column trained on the DefeatZerglingsAndBanelings minigame, and a 3-column network which uses the 2-column network trained on the DefeatZerglingsAndBanelings minigame seen above, for the previous columns.

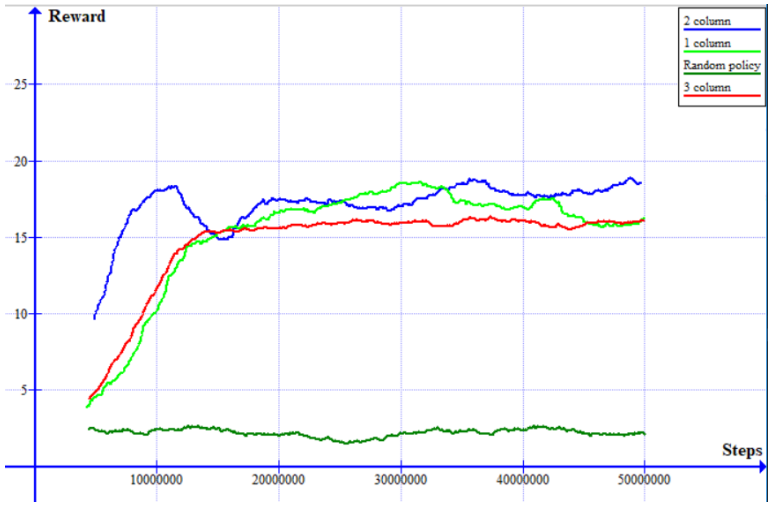
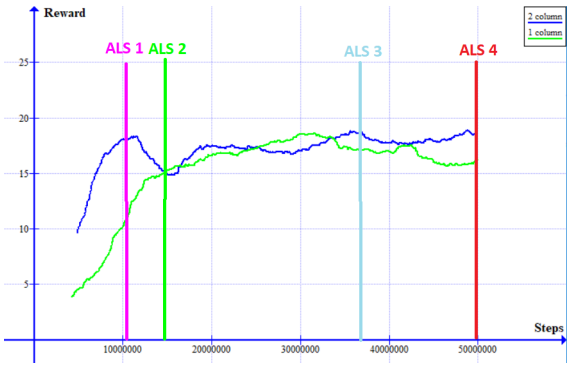


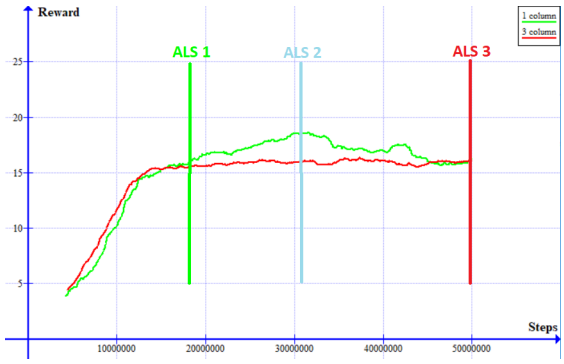
Figure 3.27: Graph showing moving average reward over 100 episodes as a function of steps on the FindAndDefeatZerglings minigame, for a random policy agent and a 1-column, 2-column and 3-column network, where the 2-column agent is pre-trained on the DefeatZerglingsAndBanelings minigame, and the 3-column network uses the 2-column network trained on the DefeatZerglingsAndBanelings minigame

Looking at Figure 3.27 on page 64 we can see that the 2-column agent seems to be learning faster at first, while it ends up with a mean reward somewhat similar to the 1-column agent. The faster learning at the start might not be due to transfer, but we can analyse the model with ALS analysis to see if the transferred knowledge is leveraged. The 3-column agent also seems to have slightly faster learning than the 1-column agent at first, but starts lacking behind after about 15 millions steps, however it ends up with about the same moving average reward over 100 episodes as the 1-column agent.

We performed a slice ALS analysis for both the 2-column and the 3-column agents from the FindAndDefeatZerglings tests. The 2-column agent slices we analyzed can be seen on Figure 3.28a, and the 3-column agent slices we analyzed can be seen on Figure 3.28b.



(a) Figure showing points at which ALS is calculated for the 2-column network depicted on Figure 3.27 on page 64



(b) Figure showing points at which ALS is calculated for the 3-column network depicted on Figure 3.27 on page 64

Figure 3.28: Figure showing points at which ALS is calculated for the 2 and 3-column networks depicted on Figure 3.27 on page 64

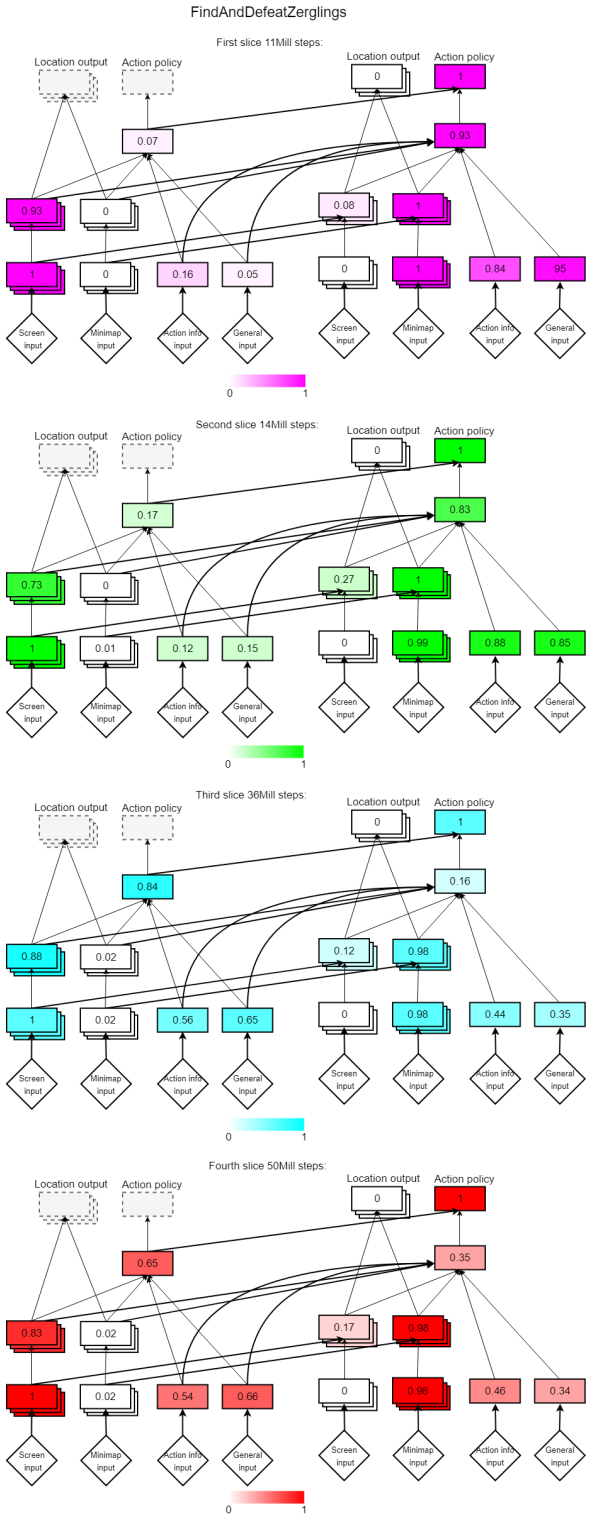


Figure 3.29: ALS per layer over the plotted 2-column progressive network in Figure 3.27 on page 64

By studying the ALS analysis of the 2-column agent depicted on Figure 3.29 on page 66, we can see that it leverages knowledge from all layers except the minimap convolutional layers. Throughout learning it manages to leverage more and more of the transferred knowledge, but never uses any of the features from the minimap convolutional layers.

The first column is trained on the DefeatZerglingsAndBanelings minigame, which does not require the player to use the minimap as the entire map is in view of the screen. It therefore makes sense that the minimap features are relearned by the agent, because only the new task, FindAndDefeatZerglings, requires usage of the minimap as part of the mechanics.

The FindAndDefeatZerglings minigame relies heavily on the ability of the agent to utilize the minimap input as only part of the map can be observed at any time on the screen input, while the minimap can be used to see what part of the map the screen is focusing on, and also to change the view of the screen.

It also makes sense that most layers other than the minimap layers can still be used as the agent in both minigames uses a specific unit type(marines) and have to defeat a specific enemy type(zerglings), on the FindAndDefeatZerglings minigame the agent has to look for the enemies around the map as well. For the fourth slice, which is at the end of the training period, the agent has relearned a bit more for other layers as well, but still leverages much knowledge from the first column.

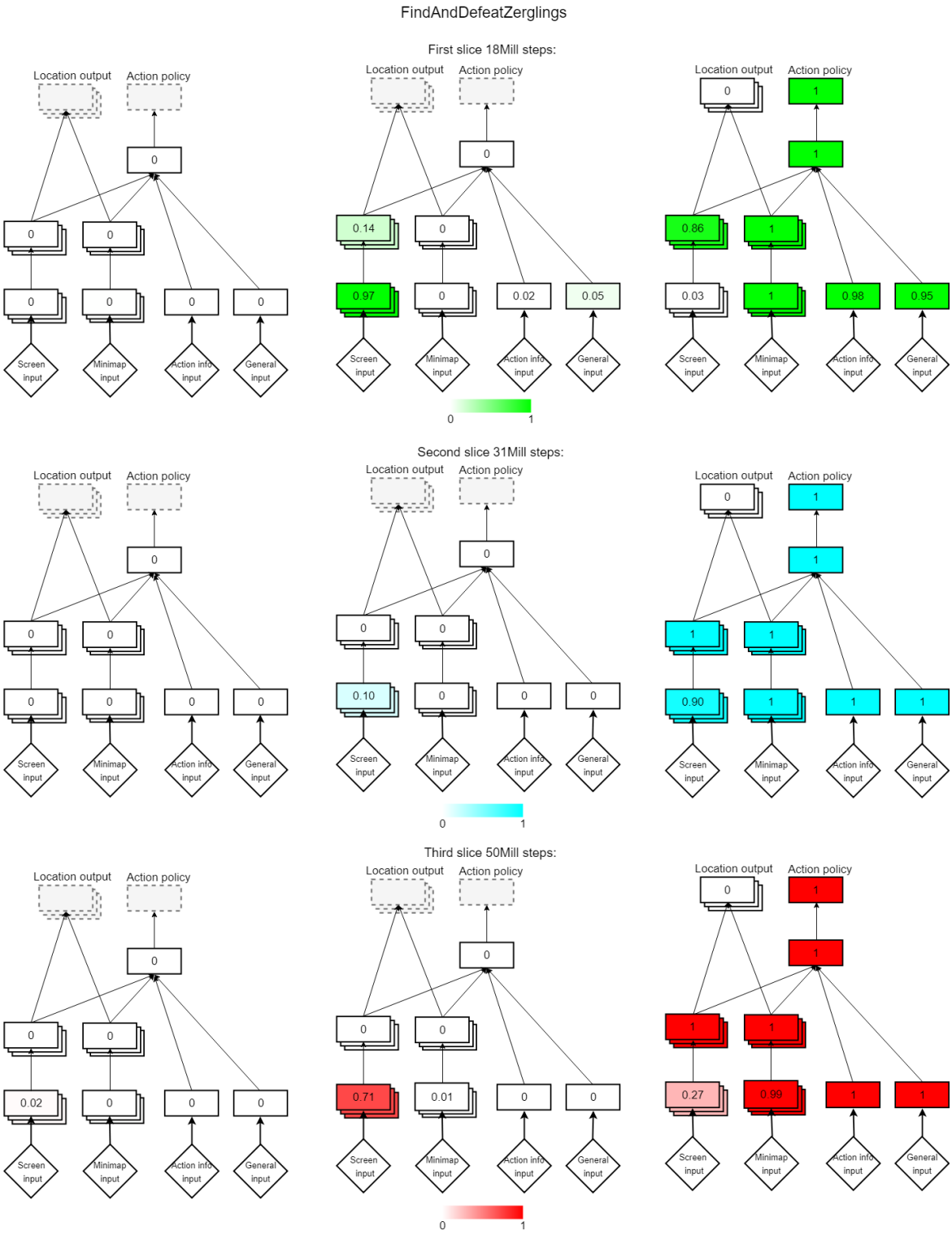


Figure 3.30: ALS per layer over the plotted 3-column progressive network in Figure 3.28b on page 65

The ALS slice analysis of the 3-column network for FindAndDefeatZerglings can be seen on Figure 3.30 on page 68. In the first slice the agent leverages some knowledge from the first convolutional layer, after the screen input, of the second column which is the DefeatZerglingsAndBanelings minigame, however it does not leverage knowledge from the first column that was trained on the DefeatRoaches minigame. This could once again be due to the FindAndDefeatZerglings and DefeatZerglingsAndBanelings minigames having more in common regarding unit types. Unlike the 2-column agent, the 3-column agent has relearned even the screen features at the second and third ALS slices.

The ALS slice analysis Figure 3.30 on page 68 shows that the agent does not really use much of the transferred knowledge, compared to the 2-column network also trained on the FindAndDefeatZerglings minigame. This could be due to the amount of parameters, as they increase with every new column, or it could be because the agent was unable to leverage the features in this test. More tests have to be made to determine the cause for the 3-column agent relearning the features. The agent starts by leveraging some of the screen convolutional features, then it learns to use its own features by the second slice at 31 million gamesteps, and then again at the third slice it relearns to leverage the features from the first convolutional layerr of the second column. This suggests that it has trouble determining if it is useful to use the previous features.

Test Conclusion
Ideally we would have wanted to do more tests on each minigame but due to time constraints and the amount of time it takes to train the agents we had to make do with the tests shown in this section. Even so we were able to gather interesting results, that indicate transfer learning is possible and sometimes, but not always, beneficial.

On Table 3.1 the training times for 1, 2, and 3-column networks in the Starcraft II environment can be seen. Here we can see that the training time increases as we add more columns which also makes sense as there are more parameters to be trained. This could pose a problem when expanding to even more columns as the training time per episode will keep increasing, however it could also result in faster convergence in which case it might not pose a problem.

# of columns	Training time (hours)	Relative to best training time
1	64	1
2	86	1.34
3	110	1.83

Table 3.1: Table showing Average wall-clock training time for different amounts of columns, in the Starcraft II environment.

Based on the tests performed on the DefeatZerglingsAndBanelings minigame and the proof of concept test on the DefeatRoaches minigame, we see that multiple columns do not always seem to improve learning significantly. Transfer of knowledge is still observed on these minigame tests, but it does not guarantee faster convergence or higher average reward even when using a previous column trained on the same minigame. This may also be because of the instability in the training on the Starcraft II environment, which is discussed in Section 2.4.3 on page 32. It is possible the first columns do not contain any meaningful features that could be leveraged for the subsequent tasks, but the proof of concept test performed on the DefeatRoaches minigame is an exception, where one could rightly assume that most if not all pre-trained features could be leveraged. The tests for the FindAndDefeatZerglings minigame, depicted on Figure 3.27 on page 64, indicate that given the right circumstances the agents can be improved on Starcraft II by transferring knowledge, where especially the 2-column agent performed well.

Chapter 4: Evaluation

This chapter will evaluate the project and conclude upon whether and how well the project solved the problem statement. We will also present some future work that might be relevant.

4.1 Conclusion

Based on our previous work with reinforcement learning and the Starcraft II environment we decided to work with the following problem statement for this project.

Can knowledge learned in a reinforcement learning setting from a complex environment, be leveraged to learn another task in a similar complex environment?

To create the best foundation for transfer learning, five tests were conducted with different variations of the A3C reinforcement learning algorithm. The best performing agent on the complex environment Starcraft II was chosen. We determined the A3C, also used in the initial Starcraft II tests by DeepMind[6], to be the agent best suited of the tested agents, and modified it to be transfer learning compatible.

For transfer learning we have used the network architecture progressive networks to determine if knowledge can be leveraged in a complex environment. Progressive networks are ideal for this, as it avoids catastrophic forgetting and thereby allows us to calculate an approximation of whether or not an agent leverages previous knowledge, and where the agent leverages the previously learned knowledge. Progressive networks are also ideal since they can leverage knowledge learned from multiple tasks while solving one task, which is Starcraft II has great potential to produce and utilize knowledge from multiple tasks, as the environment is similar across minigames, and minigames can be constructed as needed.

We tested our progressive network solution on three different domains. The first two domains were Cartpole[3], and Sonic The Hedgehog[4]. These were mainly used to prove that progressive networks and our implementation of them was functional and that we could perform ALS analysis on them. Based on these tests we found that our progressive network implementation was in fact functional, and able to leverage knowledge from previously learned tasks.

When testing on the high complexity game of Starcraft II, the progressive networks showed faster convergence to a higher average reward for one of three tests. We determined that faster convergence or a higher average reward is not always achieved even though the agents were able to leverage knowledge from different columns on all the minigames, as they did not all converge faster or to a higher average reward.

Based on our findings, we can conclude that it is indeed possible to transfer knowledge on a complex environment such as Starcraft II, but that this does not always improve the end result, and there is definitely potential for future work using progressive networks.

4.2 Future Work

We have concluded that transfer learning is possible in complex environments. More tests are needed for further conclusions, so more comprehensive testing is needed, and other things that could be interesting to see would be how longer training times affect the agent, and it could also be interesting to test the progressive networks on another kind of complex environment;

Longer & More Resourceful Training We would like to work with longer training times, if the base agent can be improved and better features learned, there might be a better possibility for transferring knowledge.

In the SC2LE paper from Google Deepmind[6], they tested every agent with 100 different hyper parameters settings, where we started out with 5 sets of hyper parameters, and narrowed it down in the final results to one set of hyper parameters. Gooogle Deepmind also ran every test for 600 million steps instead of our 50 million steps, and they used 64 workers, where we used 6-8. This means that we train our agents less and with less diverse experience than DeepMind, and since their average reward results were higher then ours, there could be more or better features for the agents to learn given more time. However the agents we made did get much better than the random agents, meaning they did learn and they are functional, but could potentially be trained even further.

New Complex Environments It would be interesting to see how the progressive networks would perform in other complex environments. Starcraft II is very complex in the sense that there are many actions with different impact on the games, and high precision locations that could have very different outcome depending on the imprecision in the location output. However the tasks of most of our mini-games are quite simple, for example: pick a marine, move to a point. It would be interesting to see what changes, if the action space and input space were simpler but the strategy required was more complicated and how that would influence the transfer happening in progressive networks which we have tested in this report.

Bibliography

[1] DeepMind. *Progressive Neural Networks*. 01-03-2018. URL: https://arxiv.org/pdf/1606.04671.pdf.

[2] Opstad et al. *SC2AI - Reinforcement Learning in StarCraft II*. 01-03-2018. URL: http://projekter.aau.dk/projekter/da/studentthesis/sc2ai--reinforcement-learning-in-starcraft-ii(59829e57-446c-4890-9b03-dc0bb6a959a6).html.

[3] OpenAI. *Gym*. 22-04-2018. URL: https://gym.openai.com/envs/CartPole-v1/.

[4] sonicretro. *Sonic the Hedgehog Genisis -Sonic Retro*. 22-04-2018. URL: http://info.sonicretro.org/Sonic_the_Hedgehog_Genesis.

[5] Sebastian Ruder. *Transfer Learning - Machine Learnings next frontier*. 22-04-2018. URL: http://ruder.io/transfer-learning/.

[6] Blizzard DeepMind. *StarCraft II: A New Challenge for Reinforcement Learning*. 16-09-2017. URL: https://deepmind.com/documents/110/sc2le.pdf.

[7] DeepMind. *PySC2 Environment*. 16-09-2017. URL: https://github.com/deepmind/pysc2/blob/master/docs/environment.md.

[8] David Silver. *Lecture 9: Exploration and Exploitation*. 23-04-2018. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/XX.pdf.

[9] Vinyals et. al. *A Study on Overfitting in Deep Reinforcement Learning*. 23-04-2018. URL: https://arxiv.org/pdf/1804.06893.pdf.

[10] Katerina Fragkiadaki. *Markov Decision Processes*. 12-03-2018. URL: https://www.cs.cmu.edu/~katef/DeepRLControlCourse/lectures/lecture2_mdps.pdf.

[11] David Silver. *Lecture 2: Markov Decision Processes*. 12-03-2018. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf.

[12] Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 19-10-2017. URL: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[13] Xiaoli Z. Fern. *Reinforcement Learning*. 23-04-2018. URL: http://web.engr.oregonstate.edu/~xfern/classes/cs434/slides/RL-1.pdf.

[14] Sutton and Barto. *Reinforcement Learning: An Introduction*. 26-04-2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[15] Nahum Shimkin. *Reinforcement Learning – Basic Algorithms*. 30-04-2018. URL: http://webee.technion.ac.il/shimkin/LCS11/ch4_RL1.pdf.

[16] Milica Gasic. *Actor-critic methods*. 01-05-2018. URL: http://mi.eng.cam.ac.uk/~mg436/LectureSlides/MLSALT7/L5.pdf.

[17] Arthur Juliani. *Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C)*. 15-10-2017. URL: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2.

[18] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *arXiv preprint arXiv:1602.01783* (2016).

[19] Kevin Swersky Geoffrey Hinton Nitish Srivastava. "Overview of mini-batch gradient descent". English. 9-10-2017. 2016. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[20] deeplearning4j. *A Beginner's Guide to Recurrent Networks and LSTMs*. 10-01-2018. URL: https://deeplearning4j.org/lstm.html.

[21] Tensorflow. *Class ConvLSTM2D*. 03-06-2018. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/ConvLSTM2D.

[22] Tensorflow. *Class LSTM*. 21-05-2018. URL: https://www.tensorflow.org/api_docs/python/tf/keras/layers/LSTM.

[23] Juan C. Caicedo Angie K. Reyes and Jorge E. Camargo. *Fine-tuning Deep Convolutional Networks for Plant Recognition*. 01-05-2018. URL: http://ceur-ws.org/Vol-1391/121-CR.pdf.

[24] Tajbakhsh et. al. *Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?* 01-05-2018. URL: https://arxiv.org/pdf/1706.00712.pdf.

[25] sonicretro. *devsisters/DQN-Tensorflow: Tensorflow implementation of human level control, through Deep Reinforcement learning*. 22-04-2018. URL: https://github.com/devsisters/DQN-tensorflow.

[26] Daniel Takeshi. *Going Deeper Into Reinforcement Learning: Understanding Deep-Q-Networks*. 23-10-2017. URL: https://danieltakeshi.github.io/2016/12/01/going-deeper-into-reinforcement-learning-understanding-dqn/.

Chapter A: Transfer

A.1 Three column progressive network

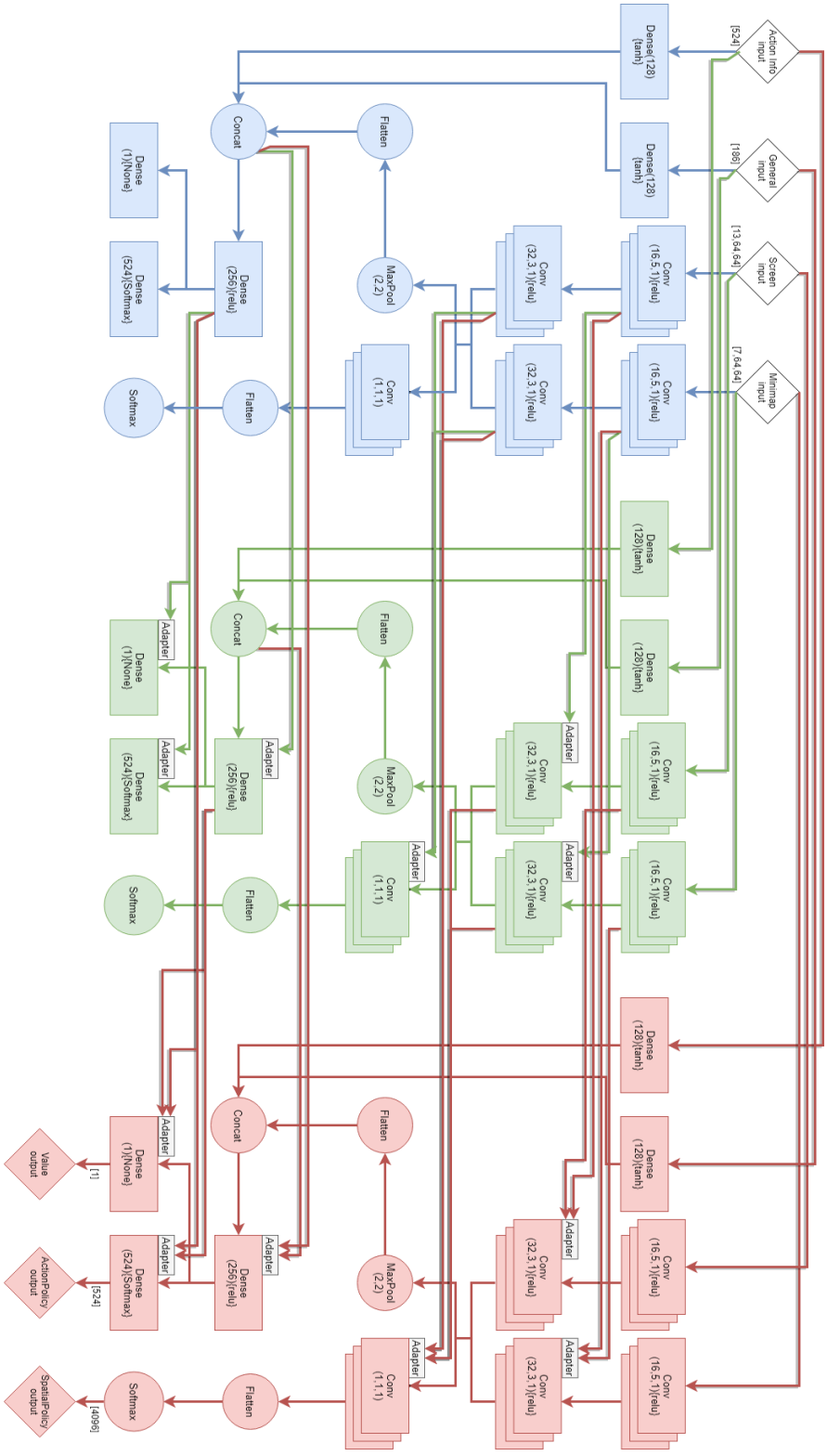


Figure A.1: Visual representation of our progressive StarCraft II network with three columns