Adapting FeUdal Networks for StarCraft II

Daniel Palenicek *
TU Darmstadt
daniel.palenicek@gmail.com

Marcel Hussing *
TU Darmstadt
marcel.hussing@gmail.com

Abstract

Sparse rewards and large time scales pose significant challenges for reinforcement learning. In this paper, we tackle those problems using a hierarchical deep reinforcement approach on the StarCraft II (SC2) video game environment. Hierarchical reinforcement learning offers methods that deal particularly well with sparse rewards and long-term planning as they provide the ability to solve a problem by splitting it into multiple sub-tasks. We adapt the existing Feudal Networks (FuN) [15] algorithm to the domain of SC2 and show that the algorithm used can be scaled up to large state-action spaces while giving a decent speed-up compared to other methods as an effect of the built-in intrinsic motivation. Additionally, our research focuses on the idea of transfer learning, since most of the learned subtasks should be reusable in different situations. We show first preliminary results that give a promising direction for further exploration in the area of transfer learning in SC2.

1 Introduction

Recently, video games have been used in reinforcement learning to simulate environments that provide similar challenges to real-world problems. The advantage of video games is that they are essentially simulations and therefore, reproducible at near zero cost. This makes for a virtually unlimited data supply which is crucial for current deep methods as they are usually very sample inefficient. Another advantage of video games is that they have well-defined rules and reward accreditations. With the rise of deep reinforcement learning, solving the learning process of video games has become one of the most important challenges in AI research. It is believed, that solving these complex environments efficiently is a great step towards *general artificial intelligence*. Therefore, we want to focus our studies on the StarCraft II Learning Environment (SC2LE) [16], which provides a new challenge for researchers with so far unsolved complexity. The SC2LE features several mini-games, each of which incorporates one or more isolated sub-tasks, that have to be solved in the full game. Currently, even those relatively simple mini-games pose a great challenge to modern algorithms and methods, since they consist of very large state-action spaces and require the ability for long-term reward accreditation and complex planning.

In our previous work, we showed that most of the mini-games can be learned by a neural network and two of them even at human performance, using a synchronous version of the Asynchronous Advantage Actor-Critic [7] as first proposed by DeepMind [16]. We were able to achieve results comparable to DeepMind's and based on these identified common issues that current learning approaches share [17]. Specifically, the more complex mini-games, DeepMind was not able to learn with their algorithm, require learning with sparse rewards, more efficient exploration and handling multiple objectives simultaneously [17].

One topic which has been receiving more attention lately is hierarchical deep reinforcement learning, as it promises to tackle some of the problems mentioned above. The general idea is to train a hierarchical policy that is able to subdivide the overall problem into several abstract sub-problems, which internally can be learned to be solved independently. By learning different spatiotemporal levels in a policy, hierarchical deep reinforcement agents are able to resolve sparse reward tasks, as the top-level hierarchy operates on a larger timescale and is not predicting low-level actions. The

*Joint first authors. Ordered alphabetically by first name.

lower level hierarchies can take into account whether they have reached the set goal and transform this intrinsic motivation into a learning signal. As a side effect, this usually aids to guide exploration as the agent will more quickly behave less randomly, and rather focus on the goals created by the hierarchy. We hope that by using a hierarchical approach to deep reinforcement learning, we are able to speed-up the learning process while also tackle more complex mini-games in SC2. FuN [15] is an end-to-end, fully differentiable hierarchical deep reinforcement learning approach. The network consists of two logical main parts. First, the manager which abstractly learns a distribution over sub-policies based solely on the reward obtained from the environment. Using these sub-policies the manager then predicts state dependent goals, which have to be achieved. The second part of the network is the so-called worker, which receives the environment's observations and the manager's goal and executes primitive actions in the environment. Its learning signal consists of both the reward from the environment and the reward given for reaching the manager's goal.

In this paper, we show, that FuN can be scaled to the larger SC2LE with very few adaptions to the network*, but the algorithm itself might not be stable enough to solve all of the challenges mentioned before for such complex environments.

2 Related Work

Video games for reinforcement learning Video games, as a particular form of simulation, have been used in reinforcement learning research for a long time. They offer multiple benefits over, for example, learning on physical robots directly. Their main benefits being reproducibility, cost efficiency, much higher simulation speeds and well-structured reward functions. Currently, one of the most popular environments is the Atari Learning Environment (ALE) [3] as it offers an extensive collection of video games with different types of discrete action spaces and raw pixel observations that all have unique challenges. As algorithms increase in performance, researchers look at more complex environments. Thus, people have been starting to use real-time strategy games (RTS) such as the SC2LE [16] as a challenge for AI research. Another recent example is the just-released Deep RTS game environment which is trying to find a middle ground in complexity between ALE and SC2LE while delivering much faster execution speeds to accelerate learning [18].

Deep reinforcement learning With the increasing popularity of neural networks, they also made their way into reinforcement learning. Researchers started using deep neural networks to represent value functions, Q-functions and policies directly [8, 4, 7]. In the latter case, weights of the neural network represent the parameters of a policy and are updated directly using a policy gradient. Recently, Minh et al. proposed a framework that uses asynchronous gradient descent for optimizing deep neural networks [7]. This made it possible to train on-policy algorithms using neural network policies, which was previously thought to be fundamentally unstable. The problem is that succeeding experiences from one trajectory gathered in only one environment are highly correlated with one another which breaks the *i.i.d.* assumption and leads to instabilities. Executing multiple agents in parallel environments helps to decorrelate the agents' experiences since each environment is in a different state and different actions are executed. Doing so has a stabilizing effect on the training process.

Hierarchical reinforcement learning The general idea of hierarchical reinforcement learning had been around long before neural networks became popular. One of the very first approaches to this was the options framework by Sutton et al. [2]. Another hierarchical concept called feudal reinforcement learning was initially proposed in [1]. The authors tackled the problems of different temporal resolutions, finding smaller subtasks and improving exploration. With regard to these problems, Dayan and Hinton propose a Q-learning algorithm with a manager that identifies a sub-task which is then executed by a sub-manager.

With the recent successes of deep learning, more people have tried to find architectures which are able to learn hierarchical policies using neural networks. Consequently, people have tried successfully to embed the concept of options into neural networks [12, 14]. For example the option-critic architecture by Bacon et al. can learn a policy over a fixed number of intra-options each of which follows a deep neural network policy. Closely related to the idea of the option-critic are the Meta Learning Shared Hierarchies introduced in [13]. In this setting the neural network has a set of primitives which

*https://github.com/danielpalen/pysc2-rl-agents

are shared within a distribution over tasks. Hence, the authors learn a set of sub-policies which are task-related and a master policy whose action is defined as the selection of a sub-policy given a specific state. Other approaches have tried to combine hierarchical reinforcement learning with deep learning while using predefined sub-goals [6, 9]. One of their shortcomings, however, is that sub-goal discovery was not addressed. In our research, we take a closer look at FeUdal Networks [15] as they are an end-to-end approach which includes sub-goal discovery.

3 Background

3.1 Reinforcement Learning

For our work we used a standard model-free reinforcement learning setup where at each timestep t, we receive a state x_t. In this state we sample an action a_t according to the network policy π and execute that action. We then receive a corresponding reward r_t and a new state x_{t+1}. The total return $R_t = \sum_{k=t}^{\infty} \gamma^t r_k$ is the sum over discounted rewards received from time t onwards. Here, γ is a discount factor which regulates a trade-off between the algorithm's tendency to favor long term over short term reward. The overall goal is to maximize the expected return $\mathbb{E}[R_t]$. In the standard setup we define the value function of state x as the expected return $V^\pi(x) = \mathbb{E}[R_t|x_t = x]$ from this state under the current policy π. Additionally, the Q-function of an action a in state x is the expected return for executing action a in state x and after that following the policy π, $Q^\pi(x, a) = \mathbb{E}[R_t|x_t = x, a_t = a]$.

3.2 The Starcraft II Learning Environment

The SC2LE allows an agent to interact with the game of SC2 programmatically. It is similar to other learning environments that have previously been used for machine learning [3]. However, there are some unique differences compared to the Atari Learning Environment for example. The main difference lies in the complexity of the observation and action space.

Observations The environment does not provide classical RGB values of the game but spatial *screen* and *minimap* observations which are each structured into a set of discrete feature maps with fixed resolution. In this work we used a resolution of 32×32 pixels for all feature maps. A feature map displays some property of the current map state, for example, the type of a unit on the map or the height of the terrain. In addition, the environment provides non-spatial, scalar observation values which represent general game state information such as the total number of resources gathered. Scalar observations or individual layer values are either categorical or integer-valued. Categorical observations may have up to 1850 categories. Overall, we use all 17 screen feature maps, all 7 minimap feature maps, and the 11 non-spatial observations.

Actions The SC2 action space is tremendously large and sub-dividing it into distinct actions is not feasible. Therefore, the action space is structured into 524 distinct functions (i.e. move, select unit, etc.), each of which takes a fixed number of discrete arguments. These can be spatial or non-spatial information depending on the action selected. There are 13 distinct types of arguments and each function may take up to one argument of each type.

4 Hierarchical Deep Reinforcement Learning for StarCraft II

4.1 Network Architecture

The network structure is based on the implementation of FuN in [15] with modifications to account for the specific requirements of the SC2LE observation and action space. The most novel thing about FuN is that it incorporates a hierarchical algorithm within one fully differentiable neural network. To explain the entire architecture, we devote different paragraphs to different logical parts of the network. A visualization of the whole network architecture is shown in Figure 1.

Observation pre-processing The first part of the network receives the raw observations from the environment and performs a preprocessing step. During this step, all of the raw observations are embedded using individual embedding layers. Note that spatial observations are embedded in a way which preserves their spatial information while for non-spatial observations this is not required.

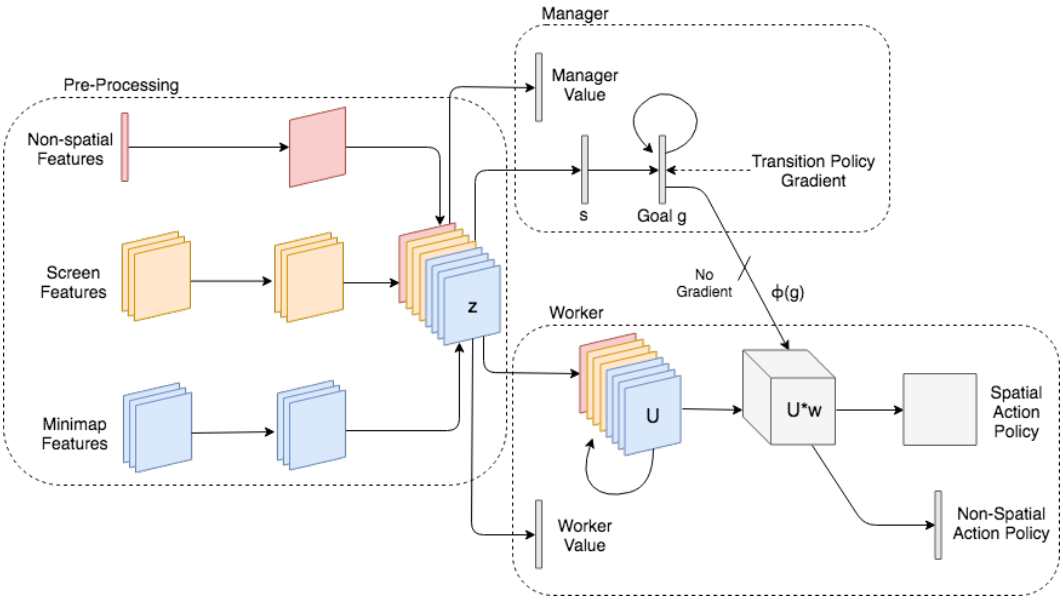


Figure 1: FuN architecture after adapting it for SC2. One can see how *screen, minimap* and *non-spatial* features are embedded and concatenated to form a single state representation. The manager predicts a goal which the worker combines with its internal state representation to predict spatial and non-spatial action policies.

Non-spatial observations are then broadcasted so their dimensions match the spatial ones. Then all observations are concatenated. The entire preprocessing part of the network is identical to what we used in our previous work [17]. The preprocessed observations are fed into both the manager and the worker part of the feudal architecture.

Manager In the manager part of the network, the preprocessed observations are flattened and transformed into a manager internal latent-state representation s_t using a fully connected layer. The output of the manager represents a goal g_t. The goal g_t is generated by summing over the last c normalized outputs of a dilated LSTM architecture on top of the internal state representation s_t. To encourage exploration, we emit a random goal with a probability of ϵ at each timestep. All of this is similar to the proposed architecture in [15]. Additionally, Vezhnevets et al. [15] derive a new update rule for the manager, which we also use in our implementation. This so-called *transition policy gradient* allows the manager to learn advantageous transitions in state space which can guide the worker part of the network in its behavior. The update-rule resulting from the transition policy gradient is given by

$$\nabla g_t = A_t^M \nabla_\theta d_{cos}(s_{t+c} - s_t, g_t(\theta)) \tag{1}$$

where d_{cos} is the cosine similarity, s is the internal state representation of the manager and c is the prediction horizon on which we will elaborate more in the dilated LSTM section. The manager advantage function is defined by $A_t^M = R_t - V_t^M(z_t, \theta)$, where $V_t^M(z_t, \theta)$ is the manager value function. Having the value function depend on the preprocessed observation input z_t instead of the raw observation input x_t as proposed in [15] is necessary due to the special structure of the observation space. Finally, a linear transformation $\phi(g)$ is used to create a vector w which is part of the input for the worker and represents its intrinsic motivation. It is important to note, that no gradients are propagated from the worker back into the manager. Hence, the manager goals g obtain a semantic meaning instead of just being latent variables in the network.

Worker The second part of the network is the worker. By taking the output of the preprocessing and putting it through a convolutional LSTM [5], we are able to maintain spatial latent feature maps U. This is essential to maintain spatial structure in order to predict spatial action arguments as described earlier. We actively encourage the worker to be guided by the manager by multiplying this three-dimensional output with the manager's linearly transformed goal w. Note that in the original paper U is a matrix transformed directly into a vector incorporating a single probability over actions.

As this is not possible in the SC2 scenario, we add an additional fully-connected layer to predict the non-spatial policy. To consider the spatial outputs we add separate 1×1 convolutional layers with 1 output channel each, which are subsequently flattened to obtain distributions over all pixel positions. All predictions are made using a softmax activation function on the last layers. To update the worker we use a regular policy gradient:

$$\nabla \pi_t = A_t^W \nabla_\theta \log(\pi(a_t|x;\theta)) \tag{2}$$

To take into account the fact that the worker receives a goal from the manager, the worker's advantage function is being slightly modified in [15], $A_t^W = (R_t + \alpha R_t^I - V_t^W(z_t, \theta))$. R_t^I is the intrinsic return the worker is trying to collect by satisfying the manager's goal. The corresponding intrinsic reward is defined by $r_t^I = \frac{1}{c}\sum_{i=1}^{c} d_{cos}(s_t - s_{t-i}, g_{t-i})$. Hence, α is a weight to regulate the influence that the manager has on the worker. Again, the worker value function $V_t^W(z_t, \theta))$ depends on z_t instead of x_t. Using the intrinsic reward, the worker can cause directional shifts in the manager's latent state and can try to achieve the manager's current goal in the future. Also, it is worth noting that in [15] the authors mention that the worker and manager can potentially have different discount factors γ. Later on, we elaborate why this is essential to making FeUdal Networks work.

Dilated LSTM One of the novelties proposed in [15] is the dilated LSTM. This architectural element is inspired by the dilated CNN proposed in [10]. A dilated LSTM is a classical LSTM with multiple cell states. The number of cell states is defined by the dilation radius r, which in our implementation is set equal to the manager horizon c. Therefore, the full state of the network is given by $h = \{\hat{h}^i\}_{i=1}^{c}$. All cell states share the same weights, and the output of the dilated LSTM is pooled over the last c states. At each timestep t only the cell state $\hat{h}^{t\%c}$ is updated. Thus, it takes the manager c timesteps in total to update the whole cell. This enables the manager to retain its memory state for much more extended periods of time and hence, operate on a larger timescale than the worker who uses a single-step convolutional LSTM [5]. For more information, a more general presentation of the dilated LSTM can be found in [11].

Transition Policy Gradient The transition policy gradient is a new form of policy gradient that was proposed by Vezhnevets et. al [15] specifically for optimizing the manager part of FuN. It is motivated by the idea of having a higher-level policy $o_t = \mu(s_t, \theta)$ which selects a lower level sub-policy based on the internal state representation. As each sub-policy comes with a transition distribution $p(s_{t+c}|s_t, o_t)$ to execute actions, the higher level level policy can be described as a transition policy $\pi^{TP}(s_{t+c}|s_t) = p(s_{t+c}|s_t, \mu(s_t, \theta))$. Optimizing this with regard to θ yields the following gradient equation $\nabla_\theta \pi_t^{TP} = \mathbb{E}\left[(R_t - V(s_t))\nabla_\theta \log p(s_{t+c}|s_t, \mu(s_t, \theta))\right]$ which gives rise to the manager update rule in Equation (1). This implies that indeed the manager is learning a policy over transition distributions which it can choose from to guide the worker's behavior.

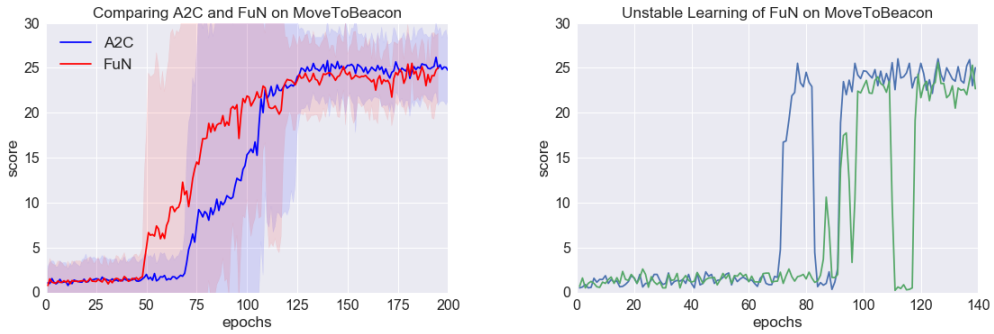
5 Experiments

5.1 Mini-Games

All mini-games from the SC2LE are played in episodes, after which the game is terminated and re-started. Mini-games terminate after a fixed time limit, and after mini-game specific ending conditions. After each episode, the cumulative score from the episode is recorded which functions as the reward.

MoveToBeacon The agent has a single unit which is pre-selected and has to move this unit (by placing clicks at the desired target location) towards a beacon that appears at a random position on the map. Upon reaching the beacon, the agent's score increases by 1 and the beacon re-appears at a different, random location. The map fits onto the screen, and thus the beacon is always visible to the agent. One episodes takes 120 game seconds.

CollectMineralShards The agent starts with two units, which have to be selected and moved in order to pick up mineral shards which are randomly distributed on the map. Upon reaching a shard, the agent's score increases by 1 and the shard disappears. Once all shards have been collected, the map is re-filled with mineral shards until 120 game seconds have passed. Since the shards are spread over the map, the agent needs to collect them in an efficient order to receive maximum reward.



(a) (b)

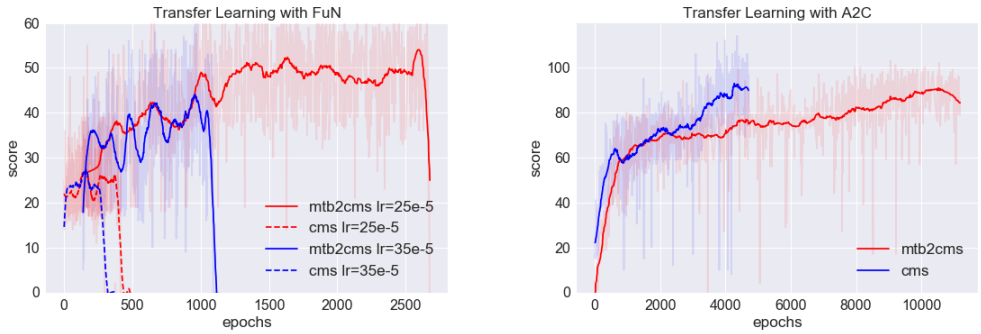
Figure 2: (a) Comparing the learning performance on the MoveToBeacon mini-game of FuN to A2C. The results are averaged over 10 runs each and show a 2σ interval. Both algorithms end up at around the maximum mean score of 26, while FuN starts learning significantly faster. (b) 2 examples of the unstable learning behavior that FuN produces at times.

5.2 Results on the Mini-Games

Results on MoveToBeacon Currently, the algorithm only manages to learn MoveToBeacon which is the simplest of the mini-games. Figure 2a shows a comparison between FuN and the previously reported A2C baseline. The results indicate that FuN is capable of learning the MoveToBeacon mini-game up to the same score as the A2C baseline. Both agents reach an average score of 26 collected beacons per game, which is comparable to human performance [16]. However, the results of FuN show a speed-up in learning. The reason for this finding could be the intrinsic motivation of the algorithm that additionally guides the learning process. Although FuN takes fewer training epochs, the overall training runtime on MoveToBeacon is approximately 5 times longer for A2C. One reason for this are $2 \times c$ additional environment steps that FuN has to take before and after each training batch for calculating s_{t-c} and g_{t-c} from the intrinsic reward r_t^I and s_{t+c} from ∇g_t in Equation (1). All of the information from these extra steps is discarded except for s_t and g_t. The rest of the information cannot be reused in future batches either, because at this point it was sampled from an older policy. Another reason is that the FuN network architecture is considerably larger than the A2C baseline network. Therefore, backpropagating through FuN takes longer.

Instability of Feudal Networks While we showed that FuN can learn MoveToBeacon, the algorithm shows a non-consistent learning behavior. In our experiments, $2/10$ runs did not learn at all, and $3/10$ crashed in performance during the learning process before then learning the game. Examples of these unstable runs are shown in Figure 2b. On complex mini-games, these instabilities get even worse. FuN is not able to learn any of the other mini-games so far. On CollectMineralShards it does not start learning, and performance crashes due to numerical instabilities, as shown in Figure 3a. The reason for this is not apparent, but might be the configuration of hyperparameters or the need to change the network architecture.

Transfer learning on CollectMineralShards Hierarchical approaches generally support the concept of transfer learning since lower-level abstractions can potentially be reused and transferred to different tasks. We trained agents on MoveToBeacon and transferred the learned network parameters as prior knowledge into the learning process of CollectMineralShards. Figure 3a shows the results in comparison to regular CollectMineralShards training runs. The plot suggests that the injected prior knowledge supports the training process as these runs increase in score. These are the only experiments where FuN has improved to better than random behavior. They also train for many more episodes before crashing from numerical instabilities. The best run was able to reach an average score of around 55 which is still not comparable to the baseline results of 97 [17]. Training a MoveToBeacon pre-trained A2C agent on CollectMineralShards does not seem to have much of an influence on the performance (Figure 3b). The training time, however, increases significantly but since we are only showing single runs, this could be a coincidence.



(a) (b)

Figure 3: (a) Transfering pre-trained FuN MoveToBeacon (MTB) runs to CollectMineralShards (CMS) in comparison to regular FuN CMS runs at different learning rates. Pre-trained runs run significantly longer and learn much more. (b) Transfering a pre-trained A2C MTB run to CMS in comparison to a regularly trained A2C CMS agent.

5.3 Optimization and Hyperparameters

Our implementation of FuN is based on A2C, a synchronous version of the Asynchronous Actor-Critic (A3C) [7]. Instead of updating a global policy asynchronously, all the environments are executed synchronously, the observations are collected, and the policy update is run in batch form on a GPU. All experiments were run using 24 parallel game environments. This is the maximum amount we could fit onto the GPU for FuN, since increasing the number of environments increases the batch size which has to be uploaded onto the GPU. Usually, A3C on SC2 is trained using 32 game environments. The lower number of environments and the resulting smaller batch size could potentially be a root of instability issues. We trained FuN with a trajectory length of 16 time steps which is comparable to the length we used in our previous report for the A2C baseline [17]. Common values for A3C on SC2 and Atari that are reported include a trajectory length of 40 time steps. Having a smaller trajectory length decreases the batch size similarly to decreasing the number of environments. Also, the memory efficiency of the dilated LSTM cannot be leveraged as nicely. For FuN, a trajectory length of 400 time steps was reported in order to best leverage the long-term dependency memory capability of the dilated LSTM on Atari. Here, we are limited by hardware constraints, but we think that even with a shorter trajectory length, a hierarchical approach should be able to speed-up learning. Thus, we use the same trajectory length for the FuN implementation as we did for the A2C baseline so we can better evaluate the impact of the hierarchical structure of the algorithm.

We use a manager horizon c of length 10, which is comparable to the published value in the original publication. However, it is notable that in our implementation the ratio between c and the trajectory length is much larger. This has a high impact on the runtime as mentioned earlier, since the $2 \times c$ extra steps per update have to be executed more often. For α, which regulates the manager's impact on the worker's learning behavior, we used a value of 0.5. Finally, we chose a discount value γ of 0.99 for the worker and 0.999 for the manager. It is necessary for the manager to discount its rewards more slowly than the worker does considering that the manager is supposed to operate on a larger timescale. When choosing a γ of 0.99 for the manager which would be equal to the worker's discount value, the network was not able to learn the MoveToBeacon mini-game.

A more thorough analysis of the hyperparameters is necessary in order to understand their impact on the network. However, since training is very expensive, big hyperparameter sweeps are very costly.

5.4 Issues with FeUdal Networks in SC2

We identified several main issues with the current approach of FuN. The first one being its sample inefficiency. Compared to other approaches FuN has a systematic sample inefficiency built in, due to the $2 \times c$ extra environment steps it has to do before and after it samples the steps for the trajectory

itself. These steps are necessary in order for the algorithm to evaluate whether the future goal the manager set was reached. Everything else from these extra environment steps is discarded. Of course, this has a much more significant impact when training with small trajectories of 16 steps compared to the A3C implementation on Atari, where DeepMind used trajectories of 400 steps. The second issue is the algorithm's instability and sensitivity to hyperparameters. One reason for this might be, that the gradient is stopped at the goal and the manager part is updated with the transition policy gradient, while the worker part is updated using a regular policy gradient. This mix of fundamentally different gradients could potentially lead to the observed instabilities. Another conjecture could be that the policy space of the higher level policy over sub-policies is too large in SC2 and hence, a small change in parameter space leads to a big change in the manager's policy space. As a result, the manager will choose sub-policies that are not related to each other in subsequent time steps, leading to un-natural behavior. A solution for this would be to limit the updates on the manager policy using a KL-divergence term. This could prevent sudden entropy collapses of the policy. Finally, we did not have the possibility to perform a big hyperparameter sweep. It might also be that the algorithm would actually be capable of learning more of the mini-games with a different configurations. This has to be tested more extensively in the future.

6 Conclusion

6.1 Summary

We show that it is possible to train FuN on much more complex games with larger state-action spaces and more complex action space definitions than Atari, which the algorithm was initially published for. Although the algorithm did only train on one mini-game successfully, we show promising results, learning significantly faster up to the same performance with a hierarchical end-to-end approach. Room for improving the algorithm's stability, runtime and sample efficiency remains.

6.2 Future work

The next step for our project is to perform a more extensive hyperparameter search on the more complex mini-games in order to be able to learn them. As the GPU's memory size limits the trajectory length and number of environments, it might make sense to switch to an A3C implementation instead of A2C in the future. This would disallow us to do efficient GPU training but enable us to better leverage the dilated LSTM architecture. Since the dilated LSTM is supposed to increase the managers time scale and thereby decrease the sparsity in rewards, this change could potentially help to stabilize the algorithm. In case that hyperparameter optimization does not enable the learning process, we have to think about changes to the current architecture. For example, one could imagine making the value estimation in the network dependent on the manager's or the worker's internal state representation respectively. Additionally, we could try to change the way in which the manager's goal is currently embedded into the worker architecture. Finally, it might make sense to use a different distance measure for our goals because we have a spatial representation to account for.

Additionally, we would like to elaborate on the possibility of transfer learning by using knowledge from one mini-game and integrating it as a prior into the learning process of another mini-game. Here, we show preliminary results on transfer learning with CollectMineralShards. These results have to be improved first, by addressing the issues mentioned earlier in 5.4. In the future, one could imagine retraining the manager part of a pre-trained network on a different mini-game. This would intuitively make sense because the worker should already have learned how to interpret the state input and how to follow the manager's goal. By merely reteaching the manager what goals are important in the new environment, the agent might already be able to learn.

Acknowledgements

We thank professor Kristian Kersting and his institute for providing us with additional computational resources to run our experiments.

References

[1] P. Dayan and G. E. Hinton. "Feudal reinforcement learning". In: *Neural Information Processing Systems*. 1993.

[2] R. S. Sutton et al. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence*. 1999.

[3] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[4] V. Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013).

[5] X. SHI et al. "Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting". In: *Advances in Neural Information Processing Systems 28*. 2015, pp. 802–810.

[6] T. D. Kulkarni et al. "Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation". In: *CoRR* abs/1604.06057 (2016).

[7] V. Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: *CoRR* abs/1602.01783 (2016).

[8] D. Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), 484 EP.

[9] C. Tessler et al. "A Deep Hierarchical Approach to Lifelong Learning in Minecraft". In: *CoRR* abs/1604.07255 (2016).

[10] F. Yu and V. Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions". In: *International Conference on Learning Representations*. 2016.

[11] S. Chang et al. "Dilated Recurrent Neural Networks". In: *Thirty-first Conference on Neural Information Processing Systems*. 2017.

[12] R. Fox et al. "Multi-Level Discovery of Deep Options". In: *arXiv preprint arXiv:1703.08294* (2017).

[13] K. Frans et al. "Meta Learning Shared Hierarchies". In: *arXiv preprint arXiv:1710.09767* (2017).

[14] D. P. Pierre-Luc Bacon Jean Harb. "The Option-Critic Architecture". In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. 2017.

[15] A. S. Vezhnevets et al. "FeUdal Networks for Hierarchical Reinforcement Learning". In: *International Conference on Machine Learning*. 2017.

[16] O. Vinyals et al. "StarCraft II: a new challenge for reinforcement learning". In: *arXiv preprint arXiv:1708.04782* (2017).

[17] D. Palenicek, M. Hussing, and S. Meister. "Deep Reinforcement Learning for StarCraft II". In: (2018).

[18] O.-C. G. Per-Arne Andersen Morten Goodwin. "Deep RTS: A Game Environment for Deep Reinforcement Learning in Real-Time Strategy Games". In: *arXiv preprint arXiv:1808.05032* (2018).