CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Learning Abstractions via Reinforcement Learning

Master's thesis in Computer science and engineering

ERIK JERGÉUS & LEO KARLSSON OINONEN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Learning Abstractions via Reinforcement Learning

ERIK JERGÉUS & LEO KARLSSON OINONEN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Learning Abstractions via Reinforcement Learning
ERIK JERGÉUS & LEO KARLSSON OINONEN

Supervisor: Moa Johansson & Emil Carlsson, Computer Science - Algorithms, Languages and Logic
Examiner: Devdatt Dubhashi, Data science and AI

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Learning Abstractions via Reinforcement Learning
ERIK JERGÉUS & LEO KARLSSON OINONEN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

In this paper we take the first steps in studying a new approach to synthesis of efficient communication schemes in multi-agent systems, trained via reinforcement learning. We combine symbolic methods with machine learning, in what is referred to as a neuro-symbolic system. The agents are not restricted to only use initial primitives: reinforcement learning is interleaved with steps to extend the current language with novel higher-level concepts, allowing generalisation and more informative communication via shorter messages. We demonstrate that this approach allow agents to converge more quickly on a small collaborative construction task.

Keywords: RL, MARL, multi-agent, DreamCoder, neuro-symbolic, abstraction, communication, AI.

Acknowledgements

Our gratitude goes out to all friends that have listened to us talk about our project and provided us with moral support. Furthermore, we would like to give the biggest thanks to our supervisors Emil Karlsson and Moa Johansson that have provided invaluable feedback and many interesting discussion topics that has shaped our work. Finally, we must thank Chalmers Software Engineering Student Union Division for providing us with a great working environment.

Erik Jergéus, Gothenburg, 06-2022
Leo Karlsson Oinonen, Gothenburg, 06-2022

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

AI	Artificial Intelligence
DSL	Domain Specific Language
DQN	Deep Q-Network
FFNN	Feed-Forward Neural Network
MADDPG	Multi-Agent Deep Deterministic Policy Gradient
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
RMSProp	Root Mean Squared Propagation
TL	Transfer Learning

Contents

List of Figures

List of Tables

1
Introduction

Communication and language are arguably two defining features of human civilisation. By sharing a communication medium, humans are able to both cooperate and transfer information within and across generations, leading to a steady growth and refinement of information. Despite it being so vital to humans, it has proven difficult for artificial intelligence (AI) to communicate in similar fashions.

Learning to communicate and coordinate efficiently via interactions, rather than relying on solely supervised learning, is often viewed as a prerequisite for developing artificial agents able to do complex machine-to-machine and machine-to-human communication [2]. This approach to language learning and emergent communication is now a vibrant field of research in the deep learning community [3, 4, 5, 6]. Recent work has focused on developing agents with single message communication [7, 8, 9], variable length communication [10] and compositional language [11, 12], via interactions and reinforcement learning. However, a striking characteristic of human communication that has been overlooked in the literature is the ability to derive novel concepts and abstractions from primitives, via interaction.

In this thesis, we study machine-to-machine communication in the context of Multi-Agent Reinforcement Learning (MARL) [13], where agents cooperate towards solving some goal in a shared environment. Agent-to-agent communication is also used in competitive settings [14], or team-based environments [15], but our focus is in direct agent-to-agent communication. The issue is to develop communication protocols that fit the environment in such a way that they both convey enough information, and do not require sending too much information. This can be tackled in a few different ways:

- The designer creates a Domain Specific Language (DSL) for the AI. This is fixed and assumed to be sufficient for the task. However, this injects the designer's bias and also keeps the protocol static.
- The designer creates a minimal DSL. Since the AI learns from as basic origins as possible it avoids designer-bias, but it can result in inefficient learning and require frequent communication.
- The agents communicate all their observations. This removes the need for a pre-made DSL, and the issues which it presents. However, it doesn't scale well with more agents nor can it be said to be a good representation of multiple agents, as every agent knows everything.

In this paper, we investigate how artificial agents can develop linguistic abstractions via interaction and reinforcement learning, starting from a small set of primitive concepts and gradually increasing the size and efficiency of their language over time.

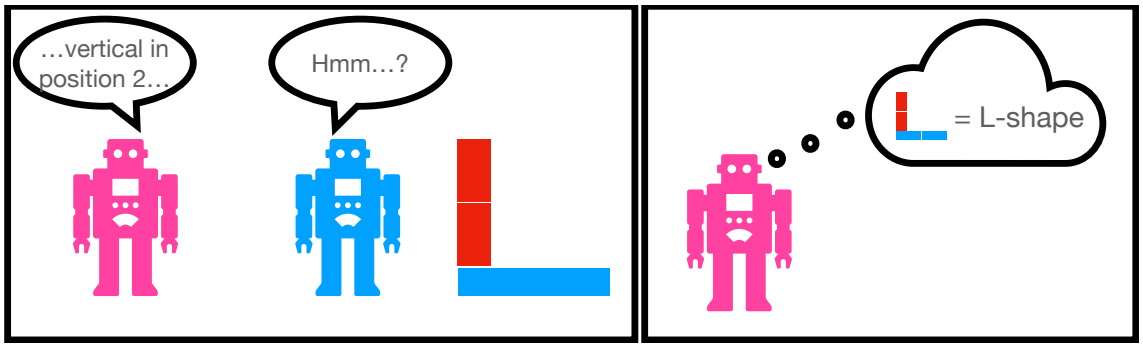

Figure 1.1: Agents should periodically reflect on their experience and consider introducing abstractions, allowing shorter utterances for constructing commonly occurring shapes.

We believe this has the benefit of being able to avoid supervising bias, while staying dynamic and allowing for an efficient language. Concretely, we investigate the impact of abstracting a series of primitive actions into higher-order actions in a reinforcement learning system. In this context, we pose our hypotheses as:

a) Having a language with messages also corresponding to common sequences of actions will facilitate the reinforcement learning construction task.

b) Our neuro-symbolic agent can discover and learn to use such concepts.

Our motivation is the architect-builder experiment by McCarthy et al. [16], investigating how humans develop communicative abstractions. Here, the architect is given a drawing of a shape, and has to instruct the builder how to construct it from small blocks. As the experiment progressed, participants developed more concise instructions after repeated attempts. Instead of talking about the positions of individual blocks, they started using abstractions describing commonly seen shapes, such as *L-shape* or *upside-down U*, see Figure 1.1.

Our contribution here is a study of a neuro-symbolic multi-agent reinforcement learning framework for this task. Inspired by neuro-symbolic program synthesis [17], the agent interleave reinforcement learning to train their neural network, with symbolic reflection to introduce new concepts for common action sequences. We show that agents learn to reconstruct the given shapes faster when allowed the capability to introduce abstractions.

1.1 Limitations

We limit the scope to only this specific cooperative environment. While testing the method on other environments would certainly be interesting, this is merely an initial study to test the viability of the system. Specifically, this environment was chosen due to the previous research in how humans acted in it.

Furthermore, we do not aim to find the perfect neural-representation of the agents, and instead we aim to see if a feasible representation can be improved. An imperfect neural-representation might even be beneficial to investigate the performance

difference of the agents', as opposed to agents that are close to perfect.
Lastly, our scope of research does not include agent-to-human communication.

2

Theory

The project touches upon multiple fields of research. Most prominently the reinforcement learning field, with a focus on multi-agent and neuro-symbolic programming. This chapter describes those fields and mention how they relate to the research we conduct.

2.1 Architect-builder Environment

The architect-builder environment was developed to investigate how humans develop cooperative communication strategies in collaborative construction tasks [1]. One subject is assigned the role of *architect* and gets a picture of a target scene to construct. The other subject, the *builder*, aims to reconstruct the shape, that is only shown to the architect, by placing vertical (2 x 1) and horizontal (1 x 2) blocks. In order to be successful, the architect needs to communicate information about the target shape to the builder.

Each scene was composed of two towers, consisting of 4 blocks each. As the experiment aimed to capture changes in behaviour, they repeated the towers multiple times. The three unique towers were paired together into a total of twelve trials.

The architect and builder were allowed any number of turns to reconstruct a scene, but they were limited by the architect not being allowed to send more than 100 characters before the builder placed one or more blocks. The builder could place blocks anywhere, if they had support from beneath. Blocks could not be moved once placed. After all eight blocks were placed, the participants got feedback related to the mismatch between the target scene.

While the humans were accurate on their initial tries, missing on average the location of one block, they still showed improvement across repetitions. McCarthy et al. hypothesized that the regularities in the shapes would result in more concise instructions over time. While they proved that the number of uttered words decreased, they also saw a specific shift in *referential words*, from words such as ”horizontal” and ”block” to ”shape” and ”C”.

2.2 DreamCoder

Program synthesis involves automatically generating programs. The field has always had a significant issue, that the valid programs are long, which results in a prohibitively large search space. Moreover, for each DSL, there is also a specific

hand-designed search algorithm required to efficiently utilize the given DSL, which leads to weaker generality.

A recent development in the field is to solve this by creating abstractions, which can drastically decrease the search space if suited to the problems at hand. DreamCoder[17] created a framework that utilizes the wake-sleep-dream algorithm, in order to solve problems by writing programs. The algorithm has three phases:

1. The **Wake** phase, where the model searches for solutions to the problems in the environment, using the current DSL.
2. The **Sleep** phase, where the model attempts to learn new abstraction primitives, by finding common fragments in programs gathered from the waking phase. If abstractions are found, they are added as an auxiliary function to the DSL dynamically.
3. The **Dream** phase, that improves the model by trying to apply the new library, learnt during the sleeping phase, on previous examples and self-generated "fantasies" gathered from a generative model.

Combining the generative model of the dream phase with the program synthesis of waking makes the entire architecture similar to a Helmholtz machine [18]. The Helmholtz machine is a type of artificial neural net that can account to the hidden data by being trained to create a generative model for the original set of data. This is done by the dual-network architecture [18]; where one of the networks takes the data as input and produces a distribution of the hidden variables, and the second, generative network, which generates values for the hidden variables.

2.3 Markov Decision process

A Markov Decision Process (MDP) is a commonly used framework for modelling decision-making, by a 4-tuple with the parameters (S, A, P_a, R_a) [19]. These parameters contain the following:

- S, the set of states that the model can reach, called state space.
- A, the set of actions that can be done, called action space.
- $P_a(s, s')$, the probability that an action a in the state s at the time t will lead to state s' in the time $t + 1$.
- $R_a(s, s')$, the expected immediate reward from action a by transitioning from state s to s'

The decisions agents make in the framework are described by a policy function π, that maps states to action (potentially probabilistically).

2.4 Reinforcement Learning (RL)

Reinforcement Learning (RL) is the process of, through trial and error, finding a policy for solving a problem. The policy is evaluated and subsequently updated through acting in the environment, either based on already gathered experiences, termed exploitation, or by new, randomized choices, termed exploration. Depending

on how good the environment deem action is, it supplies a corresponding reward, which is what the agent seeks to maximize over time [20].

All common methods, to maximize the reward, can be categorized into either off-policy or on-policy. The on-policy methods rely on the current policy to update it, as opposed to off-policy methods that only rely on the action. We have used a pure off-policy solution, but the understanding of on-policy is relevant for the field.

2.4.1 Policy Gradient (On-policy)

Policy gradient is an RL technique that optimizes parametrized policies with respect to the expected return, by using gradient descent. The advantages of policy gradients are numerous, but among the most important is the fact that policy representations can be chosen specifically for the task. However, they need to quickly forget data, as to not bias the gradient estimator. This also means that long term memory is impossible to retain, necessitating other solutions for concept retention.

$$\mathbb{J}(\theta) = E\{\sum_{k=0}^{H} a_i, \tau_k\} \tag{2.1}$$

The goal is to change the policy's parameters, $\theta \in \mathbb{R}^K$, such that the expected return of Equation 2.1 is maximized. τ denotes the trajectory formed by the sequence of states and actions $\tau = [x_{0:H}, u_{0:H}]$, where H is the horizon. a_i is the time-dependent weight factor [21].

In practice, there may exist many local maxima for any given task, which can result in the algorithm mistaking the local maxima as the global maximum. Therefore, policy gradient is suitable for improving an existing solution, but less proficient at discovering new ones.

2.4.2 Q-learning (Off-policy)

The Q-learning method is an off-policy method that defines its policy by mapping each state-action pair (s, a) to the expected reward from taking action a in state s, $Q(s_t, a_t)$. After taking an action in a state, the environment returns a reward r_t, which correspond to how good action a was in state s, which is then used to update the expected reward $Q(s, a)$.

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big(r_t + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\Big) \tag{2.2}$$

Equation 2.2 is used to update the expected value for an action a_t in state s_t. $Q(s_t, a_t)$ is the previously expected reward for action a_t in state s_t, which is increased by the *temporal difference*, multiplied by the learning rate α. The *temporal difference* is calculated by adding the received reward r_t with the expected future reward, $\max_a Q(s_{t+1}, a)$, discounted by a factor γ, corresponding to how focused the algorithm should be on receiving immediate reward. Finally, the temporal difference is subtracted with the potential reward of taking an action in the current state, $Q(s_t, a_t)$ [22].

Since some environments have too big of a state space, it is unfeasible to connect every possible state to an action (for example, the architect-builder environment would require a Q matrix of size $2^{width \cdot height \cdot |A|}$). In such a case, neural networks [23, 24] can be used to extract features from the state in order to reduce the effective space requirement for the Q-matrix, often referred to as deep reinforcement learning.

An influential work in the deep reinforcement learning field is DeepMind's paper [25], where they played Atari games using Deep Reinforcement Learning. They introduced the idea of training a variant of Q-learning exploiting experience-replay, called DQN. This approach achieved extremely impressive results and has paved the way for much of the deep reinforcement learning field. The performance was superhuman in multiple cases and fell short primarily on games that require long-term planning and/or had sparse rewards.

DQN's experience replay lets the agents store the outcome of everything the agent does, interleaving acting with actually updating the policy. A unique benefit of an off-policy method is that the policy does not need to be updated after each time step, as it does not depend on the policy used to take an action. Furthermore, the agent does not discard the data after using it to learn once, but instead keeps on learning on old data, which is useful since the Q-learning updates are incremental and do not converge quickly. This results in less volatile agents and a more efficient use of data. [25]

2.4.3 Neural Networks

A Feed-Forward Neural Network (FFNN) is a neural network without cyclical connections, as opposed to a recurrent neural network. In a FFNN, information only moves forward, from input, through hidden layers and into the output nodes[26]. For multi-layer networks, the most common learning technique is back-propagation, where the output is compared to the correct answer, adjusting the weights based on the error value. By repeating this process, the error is reduced, as the network converges to some state.

Between each hidden layer, an activation function tends to be applied which formats the output to the format desired. For example, a linear function can be used for linear regression, a sigmoidal function can be used to give a yes/no answer or a non-linear function, such as ReLU (Rectified Linear Unit), function can be used to find non-linear relations [27].

2.4.4 RMSProp (Root Mean Squared Propagation)

RMSprop is an optimization algorithm for neural networks, in the category of adaptive learning rate methods [28]. The main purpose is for mini-batch learning, while keeping the similarities of Adagrad [29]. However, it also deals with diminishing learning rates. Even with large initial gradients, RMSProp is highly efficient, while momentum-based methods overshoot towards the solution.

2.4.5 Catastrophic forgetting

Catastrophic forgetting is a common issue when training neural networks and pertains to when a neural network forgets old concepts when introduced to new once [30]. The neural networks mistakenly deem remembering old concepts as unimportant. This is a well known problem, often prevented by working with transfer learning or major changes to the way the neural network updates its weights [31].

A drastically different way of overcoming the issues is Parameter Generation and Model Adaptation, which splits the model into two neural networks [32]. The first network keeps the same parameters for all problems, while the second one's parameters are generated to adapt the solver to suit each test example. This method has resulted in high retention of task comprehension over several datasets.

2.4.6 Multi-Agent Reinforcement Learning (MARL)

In multiplayer games such as chess, it is common to view the opponent (the other agent) as part of the environment. That does not always work well and can lead to subpar performance. For example, in cooperative tasks the reward signals tend to correlate between the agents and thus require the agents to reach a consensus to get a good reward. Since the agents are incentivized to reach a consensus they will be less likely to explore new, potentially better, solutions once a consensus has been reached, leading to overfitting of policies [33]. However, by introducing a level of direct communication, such as message propagation between agents this can be mitigated, and possibly even overcome. This hints at an underlying benefit in communication in environments that are not purely competitive.

2.4.6.1 Practical Problems

The simplest approach to multi agent learning is using a standard reinforcement learning technique, such as Q-learning, with independent agents, but that does not work well in practice. This is caused by the agents' policies changing during training, coupled with them depending on each other. This makes the expected outcome of an action, in a state, dependent on non-stationary variables. Therefore, every time the policies of one agent updates, the experience of all other agents no longer reflects the new environment and thus regular experience replay does not suffice. This makes Q-learning no longer guaranteed to converge to an optimal policy [34]

The common on-policy method, policy gradient, is not viable either, as it demands observations and policies of all other agents to get useful results [35]. This is theorized to be caused by a lack of a consistent gradient signal over several time-steps [36]. If all other agents' are stationary or if the environment is fixed, it is possible to use policy gradient. However, if both of these aspects are dynamic, more advanced methods to analyse the environment are needed.

The *Multi-Agent Deep Deterministic Policy Gradient* (MADDPG) agent architecture attempt to alleviate these issues by using a centralized gradient for the agents [13]. This results in a cooperative environment where the recipient does not have a one to one match of the policy of the observer, and instead

an approximation that is sufficient for the same results as the true policy. This does not demand the same exhaustive search for policy convergence. However, this also assumes heterogeneity between the agents, and an applicable global gradient, which is impossible if the cooperative agents have different tasks and intermediate goals.

Experience Replay is a high-quality method in single-agent environments. Despite this, it has been shown that disabling experience replay can be beneficial in multi-agent environments. However, it is possible to use synchronized sampling between agents $(\psi_1, a_1, \psi_2, a_2)$, such that correlated actions corresponds in the buffer [37]. This means that a reward for the action of one agent is correlated to the action of the other agent, and not a completely independent reward function for each of the agents. This is relevant since the cooperative aspect is important for the learning process, and ensures the understanding that a good action for agent ψ_1 might be mediocre for the full environment, if the action of agent ψ_2 is incompatible.

2.4.7 Transfer Learning

Transfer Learning (TL) is a problem found in all branches of Machine Learning (ML), relating to reusing knowledge, gained from solving one problem, to different but related problems. Specifically within RL, it can significantly improve the sample efficiency [38]. TL therefore has the potential to remedy the relatively computationally expensive part of generating samples.

In the context of deep learning, both TL and cooperative MARL aim to propagate network alignments (knowledge) for different agents, but their methodologies are vastly different. TL assumes sequential transferral of knowledge. On the contrary, MARL relies upon continuous exchanges and non-stationary experiences, which makes them incompatible in practice [37].

3

Methods

We model the architect-builder environment in a multi-agent reinforcement learning setting, interleaved with symbolic reasoning to introduce new concepts. One agent takes the role of architect and the other is the builder. Both of these agents start with no pre-conceived notion about the environment. After some successful interactions, the architect will symbolically reflect on its interactions with the builder, and identify commonly occurring repeated sequences of instructions. It will then introduce a novel concept to be used in subsequent interactions with the builder, in the main reinforcement learning loop (see Figure 3.1). If the agents manage to construct a shape with fewer instructions, a higher reward is given.

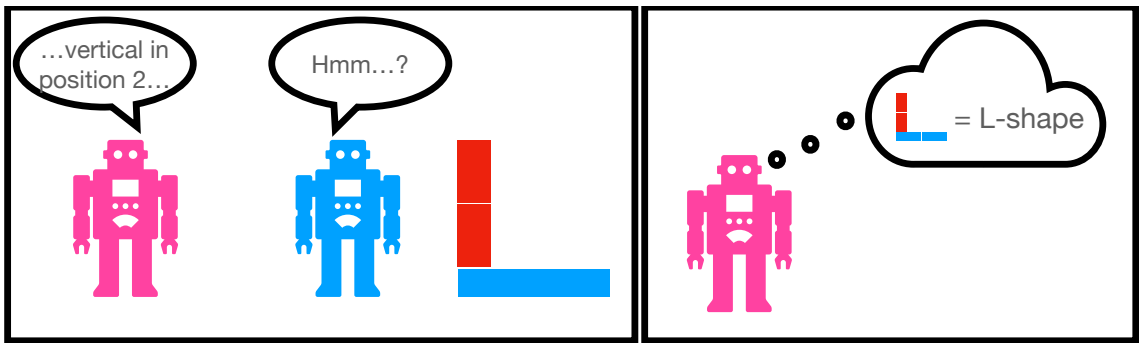

Figure 3.1: A collaborative assembly task. The *Architect* was shown a target scene and provided assembly instructions to the *Builder*, who aimed to reconstruct it without seeing the target. [1]

3.1 Environment

Our setup mimics the one from McCarty et al. [16] where two agents, the architect and the builder, communicate about a set of geometric shapes. The environment is modelled in the MDP framework (see Section 2.3). S is the target scene and current scene, A is the blocks that exist and where they can be placed, $P_a(s, s')$ is simply 1 when action a leads to s' in state s. Finally, the reward function is modelled to reflect a higher reward when the current scene becomes and/or approaches the target scene. The agents' objective is then to figure out an optimal policy π^* that results in a minimal set of actions to reach the goal state for any S.

The architect's input is a picture of the goal state alongside the current state, each of which is represented by binary W x H matrices ($W = 6$ and $H = 6$ for our

Figure 3.2: The architect sees both the goal and the current state and decides to instruct the builder to place a vertical block in position 4.

experiments), see Figure 3.2. Locations where there are blocks are represented as 1's and empty locations by 0. This implementation does not separate between horizontal and vertical blocks, in contrast to the experiments by McCarthy et al. This makes the goal harder to interpret for the architect, but it makes for a simpler model and results in a more ambiguous environment. Furthermore, in our model, there exists a primitive version of gravity, much like the gravity in the game of Tetris. This means that a sequence of actions are not reflexive.

This is passed through a FFNN, which outputs a message with instructions to the builder. The message is a one-hot array of size depending on the current amount of available actions, the size of the action space. That varies in size depending on the current amount of learned abstractions and which form of action space is used. The builder then interprets this through a FFNN, that takes the message and tries to output the corresponding action.

3.1.1 Action Space

As inspired by an RL approach to Tetris [39], the action space is implemented in one of two ways. Grouped actions used the form "place a block on a location", as described in Figure 3.2. With the other action space, non-grouped, the builder had a current location, which they could "place a block" on. The builder could alter its location with the actions "move one step left/right". Location need to be encoded in this case, which was done in one of two ways. Either by appending a row at the top of the state as a one-hot array encoding the location. Or by rotating the goal and current state in the opposite direction of where the builder moves, basically resulting in the agent always placing blocks on the initial location.

In both settings, blocks are always placed from the top of the current location, dropping until the ground or a block is met. If a block or movement goes past the width requirements, it is wrapped around to the other side. If the block is placed such that it is above the top location, the action will be skipped.

When abstractions were made for the non-grouped action space, the abstractions were simply a sequence of the primitive actions. The grouped action space's abstractions instead corresponded to adding another type of block into the available action space. Therefore, the action space's increase differ between the two. The non-grouped action space only adds 1 per abstraction, while the grouped adds W new actions per abstraction, which could affect their scaleability.

Table 3.1: The hyperparameters for both agents.

Parameter	Value
$\epsilon, \epsilon_{decay}, \epsilon_{min}$	0.99, 0.9995, 0.03
τ	200
γ	0.95
α	0.0001
Optimizer	RMSProp

3.1.2 Reward Function

The architect receives a reward R at each time step t when performing an action a. It either receives a large reward if the new state s matches the goal g exactly, or a smaller reward if the most recently placed block partially matches the goal. This reward function is given in Equation 3.1, where *partial_match* denote the number of new grid squares, covered by the most recently placed block, matching the goal. The reward function was written in such a way as to heavily encourage perfect completion, but not discourage incomplete solutions either. Without reward for partial construction, the architect had problems converging on difficult problems, where it is unlikely to randomly find the solution.

$$R_t(s, g, a) = (0.1 \cdot partial\_match + (s == g)) \cdot 0.9^t \tag{3.1}$$

The builder is rewarded simply depending on if the performed action is what the architect intended. The reward for a correct action was 1 and wrong actions resulted in -2. The penalty needs to be strictly more negative than the reward for being correct. If it was not, the builder did not get sufficiently penalised from intentionally stalling by moving back and forth (in the non-grouped case) or placing blocks above the grid.

3.2 Deep Reinforcement Learning

The agents are constructed as DQN agents [25] with experience replay, using RMSProp for optimisation. The replay buffer is simply a queue of size $1e + 6$. The network architecture varies between the architect and the builder. For more hyperparameters, see Table 3.1.

The builder's network, see Figure 3.3, is a rather small FFNN of 3 linear layers interspersed with ReLU activation functions. Larger networks resulted in slower learning, and the builder's optimal policy is so simple that there is no need to use a complex network to describe it.

The architect's network, see Figure 3.4, went through a lot of iterations. The initial thought was to use convolutional neural networks, as there is a spatial relation in the states. When using those, the learning speed (and thus cumulative reward) was significantly better than simple FFNN for the non-grouped action space, but significantly worse for the grouped action space. As the grouped action space still performed better in all our final tests, the final architecture settled upon was a

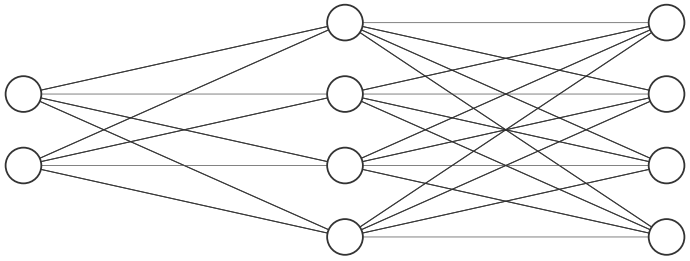

Figure 3.3: Each of the neurons in the picture correspond to $W \cdot H$ actual neurons. Furthermore, there is a final layer of size $|A|$, which is not shown as it varies.

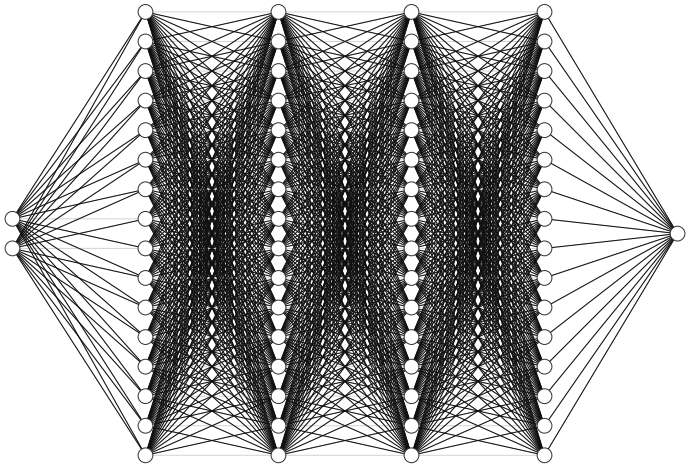

Figure 3.4: Each of the neurons in the picture correspond to $W \cdot H$ actual neurons. Furthermore, there is a final layer of size $|A|$, which is not shown, as it varies.

FFNN network of 6 layers, similarly interspersed with ReLU activation functions. Additionally, the network has a dynamically variable size dependent on the number of possible abstractions the network is initialized with. However, as the input values for the basic actions exist from the beginning, the input values will have low meaning until abstractions have been created.

3.3 Wake-Sleep-Dream cycle

In order to vary abstraction and acting, a version of the cycle described by DreamCoder [17] interleaves creating abstractions with solving the problem and reflecting upon the abstractions.

This section describes the flow of the program in detail, with some high level reasoning behind important decisions. The pseudocode in this section clarifies the details of the wake-sleep-dream loop described above in broader strokes. For the exact hyperparameters and implementation, see GitHub [1].

The main loop alternates between the three phases. Furthermore, it checks if the architect and builder has constructed a viable, combined policy. That combined policy is then evaluated, without exploration rate, by attempting to solve all the

[1] `https://github.com/jerge/MARL/tree/Communicative-Abstractions`

currently allowed goals of the environment. When the policies correspond to a decision-making that solves the current possible goals, the environment adds a new goal to the pool of target goals it chooses from when resetting. The environment is initiated with 1 allowed goal and is always skewed towards selecting the latest added goal.

```
Repeat:
  sleep_history <- Wake(Architect, Builder, Environment)
  abstraction   <- Sleep(sleep_history)
  Architect.AddAbstraction(abstraction)
  Builder.AddAbstraction(abstraction)
  Dream(Architect, Builder, sleep_history, abstraction)
  if Architect.policy and Builder.policy reaches the goal
      for all states in Environment:
    Environment.AddNewState
  endif
```

3.3.1 Wake

The wake phase aims to train the agents by acting in the environment. It starts by resetting the environment in order to generate a state and a random goal. This state and goal is then sent to the architect in order to generate a message for the builder, which in turn generates an action to be taken in the environment. This repeats until the environment deems the agents' done. They are considered done when the current state contains the same number of blocks (or more) as the goal.

While the builder and architect act upon the environment, they interleave who is exploring and who is staying stationary. This is necessary as regular experience replay does not account for non-stationary environments. For example, if the architect tells the builder a message m_1, which correspond to an action a_1, the builder might take a random action, a_2, to explore what the message means. If this interaction is trained upon by the architect, it will find that there is a chance that m_1 results in the state and reward that a_2 does. The algorithm will still converge towards finding the optimal policies as the exploration rate decreases, but it takes a lot more time than the downside of interleaving the trainee.

The history of the latest 500 successful epochs is saved for use in the other phases. Finally, the wake phase repeats until there has been a total of at least 1000 steps in the wake phase.

```
Wake(Architect, Builder, Environment) -> sleep_history:
  state, goal <- Environment.Reset
  while not done:
    message <- Architect(state, goal)
    action <- Builder(mesage)
    new_state, reward, done <- Environment(action)
    history += (state, message, action, reward, done, new_state)
    state = new_state
  Architect.AppendReplayBuffer(history)
  Builder.AppendReplayBuffer(history)
```

```
  Architect.Train
  Builder.Train

  success <- Environment.goal_is_state()
  if success:
    sleep_history += history
  endif
  repeat until 1000 steps has been taken
  Architect.SwitchTrainingMode()
  Builder.SwitchTrainingMode()
  return sleep_history
```

3.3.2 Sleep

The sleep phase aims to find the best abstraction available in the current situation, through analysing the recent history.

The sleep phase starts by using the Cartesian Product (`CP`) to get all combinations of successful sequences. Between each pair, the Longest Common Subsequence (`LCS`) is computed to get the most useful abstraction for that pair. Alternatively, all subsequences of a sequence could be deemed a candidate, instead of the longest common sequences in a pair. It would be more computationally expensive, but would not skew the program towards as large sequences. By picking the longest common subsequence, the candidates are instead more likely to encompass structures corresponding to entire towers, as McCarthy et al. showed that humans did. However, by giving further weight to occurrences as compared to length, one can adjust the propensity of generation of the length of abstractions.

Each of those candidate abstractions' utility are then rated to find the best one. We value the potential improvement as the number of actions that can be avoided by picking the abstraction. Furthermore, it is valued based on how common the sequence occurs.

This is the best utility function that was found, but multiple other factors could play a part. For example, it would be possible to account for how good the agents are at picking the sequence currently, in order to help the agents with sequences that are hard for it. While that approach might help the agents at this point, it does not account for the agents' policies being dynamic.

The phase finishes with a check to see if the top-rated abstraction contains at least 2 blocks and does not correspond to a previously made abstraction. That clause was added to avoid the addition of abstractions that have no practical meaning.

```
Sleep(sleep_history) -> abstraction:
  candidates = empty_map
  for i,j in CP(sleep_history, sleep_history)
    candidates[LCS(i,j)] += 1
  evaluation = [Length(sequence) * value for sequence, value in candidates]
  return candidates[Argmax(evaluation)] if Length > 1
```

3.3.3 Dream

The dream phase aims to evaluate the agents' abstractions. It chooses to either evaluate a newly acquired abstraction to train the networks to use it immediately, or it evaluates if any of the previous abstractions are not being used.

New samples containing the new abstraction are made by searching through the history used for the sleep phase, replacing every instance of the abstraction's sequence with the abstraction. Then, these edited epochs are inserted into the agent's replay buffer and included in a training iteration. By adding them to the replay buffer, the agents are able to continue training on them during the wake phase. Otherwise, there is a high chance that the agents would immediately forget the implications of the new abstraction, as newly acquired samples would substantiate a small part of the replay buffer.

The old abstractions are only evaluated based on their recent frequency in the successful epochs. If they are used as seldom as if it were picked solely based on the current exploration rate, they are deemed to be useless enough to be removed from the action space. While the parameters are not explicitly changed afterwards, the network quickly forgets them, as they were hardly used in the first place. In practice, it is uncommon to remove abstractions. The abstractions picked in the sleep phase are generally useful enough to be used and recognized. Otherwise, the sleep phase would not have rated them highly enough to create the abstraction in the first place.

```
Dream(Architect, Builder, sleep_history, abstraction) -> {}:
  if abstraction is not none:
    for sequence in sleep_history:
      if sequence contains abstraction:
        new_sequence = InsertAbstraction(sequence, abstraction)
        Architect.AppendReplayBuffer(new_sequence)
        Builder.AppendReplayBuffer(new_sequence)
      endif
    Architect.TrainOnRecent
    Builder.TrainOnRecent
  else:
    # Note that iterations != epochs
    Iterations <- sum([|epoch| for epoch in sleep_history])
    MinAllowedUses = Iterations * Epsilon * (1 / |Architect.ActionSpace|)
    for abstraction in Architect.ActionSpace:
      if Iterations.count(abstration) <= MinAllowedUses:
        Architect.RemoveAbstraction(abstraction)
        Builder.RemoveAbstraction(abstraction)
      endif
  endif
```

3.4 Set generation

To evaluate our system, three separate sets of data were constructed, $S_{random}, S_{structured}$ and S_{mixed}. S_{random} is an unbiased set which is used as a baseline. The other sets have inherent bias and are used to test hypotheses. All of these sets contain a number of constructs. These constructs have a difficulty value, D which is the number of primitive blocks contained within the construct. Furthermore, the environment's width W and height H are both 6 for all sets.

3.4.1 Random set

The random set of data, S_{random} is important to measure when abstractions are meaningful. We do not expect abstractions to be useful, or even generated on random data, as they are based on recurrent observation of regular structures. The expectation here is that a neural network would not attempt to fill the data with abstractions where none can be found - unlike a human being that often create abstractions out of noise.

S_{random} was created by randomly sampling actions from the non-grouped action space on a blank environment. As half of the actions were block placements, it was common for all blocks to be densely packed at the initial location. Therefore, the actions that move were twice as likely as the ones that place blocks. It does this until D blocks has been placed and then saves the current state as a potential goal for S_{random}. This set was generated by doing that 20 times for $D \in [1, 2, 3]$ and then removing duplicates.

3.4.2 Structured set

The set $S_{structured}$ is a hand-crafted set of data, where each possible goal consists of either the shapes "C", "L" or "upside-down U" ($\cap$), see Figure 3.5. These shapes were chosen based on the work in McCarty et al. [16], as simple, human-understandable abstractions. Due to basing the shapes on their experiment, it is easier to make a more direct comparison between how our system, as opposed to humans, utilizes abstractions. The difficulty, for all of these sets is $D = 4$, and is the shapes and their placements.

McCarthy et al. used two shapes for each example, but we opted to only do one shape per example. There were three issues with having multiple shapes per example. Foremost, it would have been difficulty for the agents to reliably be able to solve a $D = 8$ problem as it has not been a focus to create an optimal neural structure for them. Furthermore, the rewards would become more sparse, which makes the agents have an even harder time to converge. Lastly, the size of the environment would have to increase, which both make the neural architecture different from the other sets and more importantly increases the neurons, which could have adverse effects on training time.

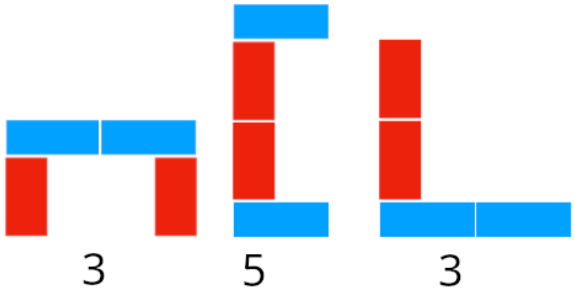

Figure 3.5: $S_{structured}$ contains 3 upside-down U, 5 C and 3 L constructs in a 6 by 6 grid. This is all possible locations that do not result in a wrap around the sides.

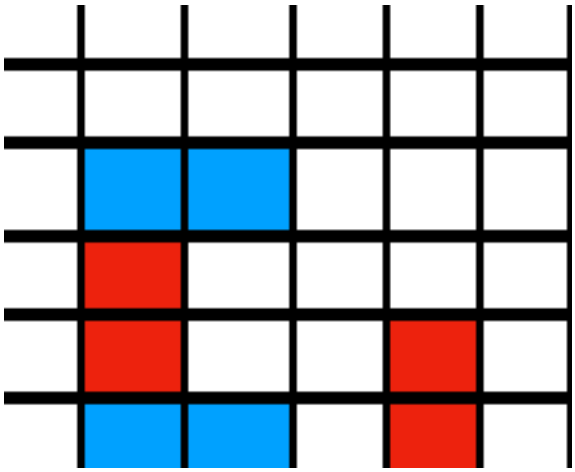

Figure 3.6: S_{mixed} contains multiple small ”c” shapes, both combined and on their own. This is an example of a small ”c” placed on location 2 and a vertical block on position 4.

3.4.3 Mixed sets

The mixed set, S_{mixed} is a partially randomized set, but with a handcrafted ”c” shape, see Figure 3.6, inserted into the samples. This was done upon a baseline S_{random}, set, and then changed. The set was initialized in the same manner as the purely randomized test set (except that we created 10 instead of 20 examples per value of D), but then altered by randomly replacing one of the blocks, for some randomly determined cases, with the small ”c”. This set has a difficulty equal to $D_{S_{random}} + 2$.
The purpose of this set is to see if the agents are able to pick out and use the abstraction ”c”, despite there being multiple random shapes obfuscating it.

4
Results

The paper's goal has been stated to be:
Investigate how artificial agents can develop linguistic abstractions via interaction and reinforcement learning, starting from a small set of primitive concepts and gradually increasing the size and efficiency of their language over time
The most important part to evaluate in the goal is if the artificial agents develop linguistic abstractions, which leads to a more efficient language, from primitive concepts. This could be measured by comparing the number of interactions needed for agents that are allowed to make abstractions with the agents which are not. That metric is a self-fulfilling prophecy, as it is impossible for agents without abstractions to perform better than those with it. The agents will always make D actions for an example with difficulty D if they are optimal. Except the agents that utilize abstractions, that has the possibility to use fewer abstractions. Therefore, we find it more fair to compare the number of epochs required to find a policy which solves each example in a set.
With this reasoning, our hypotheses are:

a) Having a language with messages also corresponding to common sequences of actions will facilitate the reinforcement learning construction task.
b) Our neuro-symbolic agent can discover and learn to use such concepts.

We have conducted 3 experiments to test these hypotheses. They all used the grouped action space, as it performed better for all scenarios. The results were gathered by running all types of systems in the experiments thrice (independently) and taking the median results. The order in which the sets were sorted was random, except that they were in an ascending difficulty level and kept stationary within each experiment. While some detail changes between the iterations, it is uncommon and mentioned when applicable.

4.1 Experiment 1 (Random)

The initial experiment tests our hypotheses against a set which is randomly generated, as described in Section 3.4.1. As seen in Figure 4.1, without any abstractions, it takes 140 000 epochs to solve all 49 shapes. Conversely, agents that can make abstraction require 175 000 epochs, as seen in 4.2. Naturally, it was not possible to include a trial with pre-given abstractions, as we as humans do not deem any abstractions useful for this task.

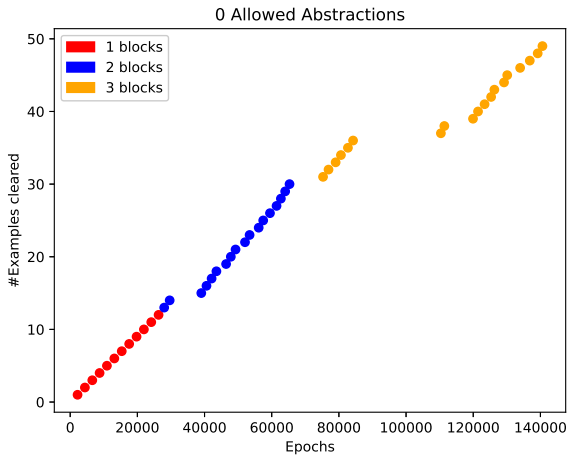

Figure 4.1: Without the capability to create abstractions, using only initial primitives, learning to build all 49 shapes in S_{random} requires over 140 000 epochs.

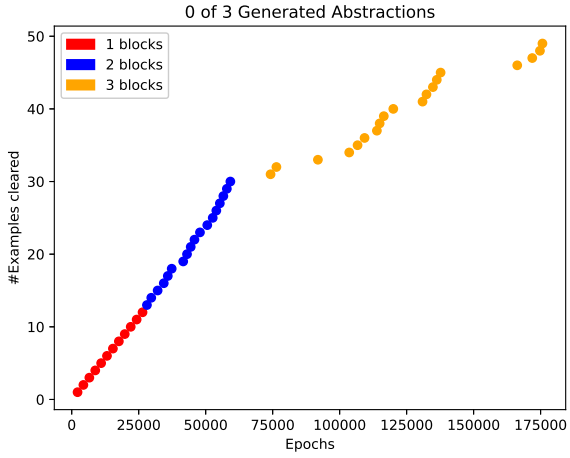

Figure 4.2: With the capability to create abstractions, but choosing to never create any, learning to build all 49 shapes in S_{random} requires over 175 000 epochs.

Most notably however is the fact that at no point in these 175 000 did the agents create an abstraction. After every sleep phase, the agents did not find any abstraction that were useful. This is partly due to the agents often being quick enough at finding a good policy that the system did not enter the sleep phase (all examples with 1 or 2 blocks, except #19). The major reason as to why it did not happen however is that in 2 out of the three runs the sleep phases determined that the initial primitives were more useful than any candidate abstraction. In the one case that an abstraction was found, it was deemed useless 3 epochs later by the dream phase and therefore did not have any noticeable impact on the results.

We consider that these are reasonable results, since the data in our samples have no consistency or recurrent shapes, and thus should not be helped from our abstractions, which is also consistent with the expectation. The added time for the agents that can create abstractions is also reasonable, as the complexity of the network increases, which necessitates higher training times. The created ”useless” abstractions that have little long-term value can be inserted into the model at worst, but if they have no actual value, the pruning mechanism will remove them.

4.2 Experiment 2 (Structured)

The second experiment is most notable as it provides results supporting our hypotheses and is tightly connected to the human experiment done by McCarthy et al. [16]. The structured set, described in Section 3.4.2, was ordered such that all versions of the same shape were next to each other, but otherwise shuffled. The set is more complex than the other sets and is therefore almost impossible for the agents to solve from scratch. Therefore, all agents in this experiment were pre-trained on the 49 examples in Experiment 1 (4.1), in order to have preconceptions of the primitive action mappings. Moreover, the agents without abstractions were not able to complete all eleven examples of the set in a reasonable time. Therefore, the builder was assumed to understand the architect perfectly for this experiment, as that does not necessarily impact the value of abstractions.

The experiment uses three different versions of the system. The first version was not capable of creating abstractions and required 350 000 epochs to complete all examples 4.3. Due to the high complexity of the sets, this version had serious issues converging to a sufficient policy. The specific problems varied across iterations, but one commonality is that compared to when abstractions were allowed, it was uncommon to be fast in completing the second construct of a shape fast. We believe that is due to the networks being relatively bad at generalising.

The next version was allowed to generate 3 abstractions and learnt the entire set in 160 000 epochs, see Figure 4.4. In the graph, it is marked when the agents developed abstractions, but notably only 2 abstractions were generated in this iteration. The first abstraction, the ”upside-down U” ($\cap$) arose from exploring the correct solution once and then immediately deciding that it was a good abstraction. The generation of the second abstraction generally required more epochs, often due to the sleep phase re-generating the first abstraction, but ordering the actions in a different sequence. The agents did not decide on an abstraction that corresponded to the last

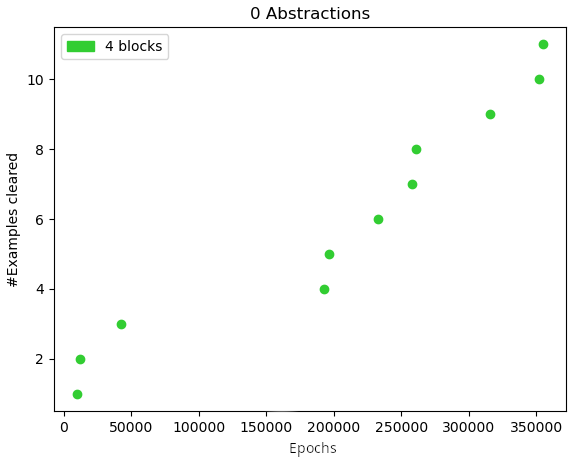

Figure 4.3: Without the capability to create abstractions, learning to build all shapes require 350 000 epochs.

shape in two out of three iterations. In those cases, it was due to the agent solving the problems efficiently, as no abstraction step was taken between the addition of new examples into from the training data.

A sidenote is that in no case did the algorithm decide that an abstraction that helped for multiple types of shapes (or any that was not a complete construct) was better than either of the more obvious abstractions. Due to the specific shapes in this test case and how we generate abstractions, this is expected. No abstraction has more than 1 primitive in common across all three examples. Between the ”C” and ”L” a 3 block abstraction could help, but as the algorithm already did not create an abstraction for the ”L” and the ”C” already had an abstraction, it was unlikely to be found. However, larger environments would give precedence to the possibility of such an abstraction (discussed in Section 5.1).

Finally the best-case was tested, when the abstractions corresponding to each construct were given in advance. For this case, the agents were able to complete the examples in almost the fastest possible time with only 17 500 epochs required.

4.3 Experiment 3 (Mixed)

For completeness, the impact of the abstractions is also evaluated upon a set that mixes random, unbiased, constructs with once that have reoccurring shapes, S_{mixed}.

The agents completed the 16 examples in 80 000 epochs without creating abstractions, see Figure 4.6. In Figure 4.7 it is shown that the agents that were allowed to create one abstraction were almost twice as fast at solving the examples, with less than 50 000 epochs. This clearly indicates that the abstractions have a strong positive impact on the proficiency of the agents in test samples with a mix of structured and random structures. This indicates towards the capability of efficient utilization of abstraction, even in irregular environments.

Notably, the main difference in time is the 8th example (see Figure 3.6), which is

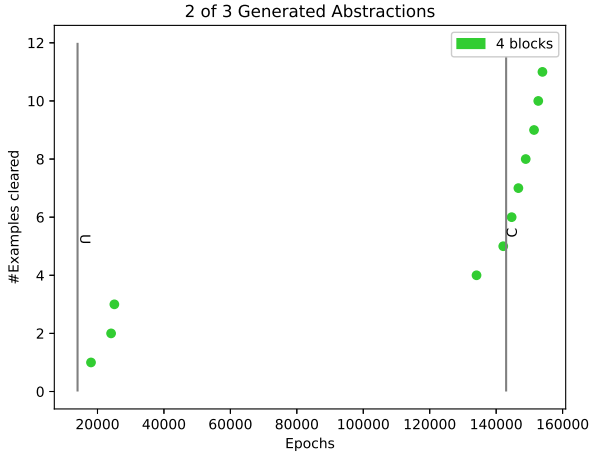

Figure 4.4: With abstraction-discovery, the agents need 160 000 epochs to learn to build all shapes. The horizontal lines mark when the abstraction *upside-down U* and *C*-shape were introduced.

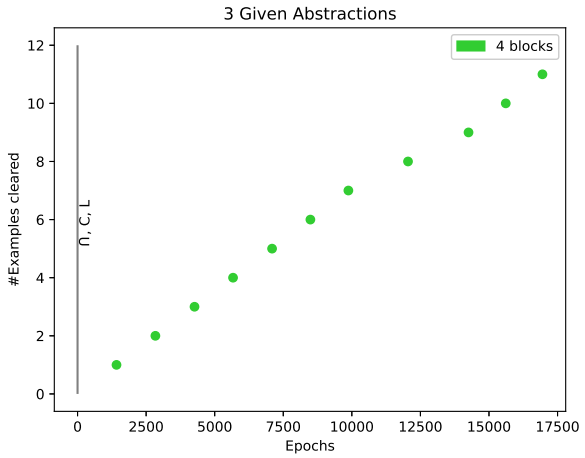

Figure 4.5: If agents are given the relevant abstractions upfront (”C”, ”L” and ”upside-down U”), the task can be learned in 17 500 epochs, which is a magnitude faster than if no abstractions are allowed.

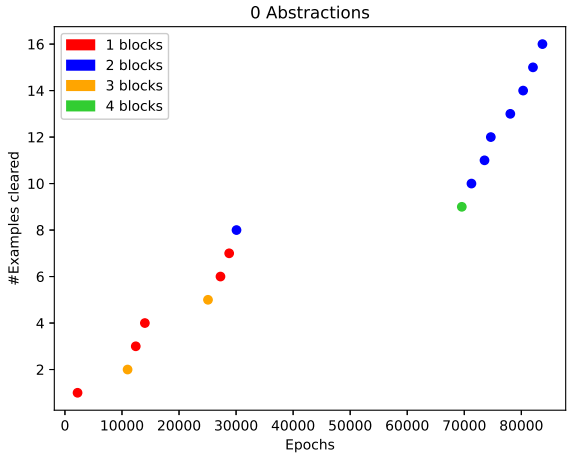

Figure 4.6: If agents are not able to create abstractions, they learn to build all shapes in S_{mixed} in 80 000 epochs.

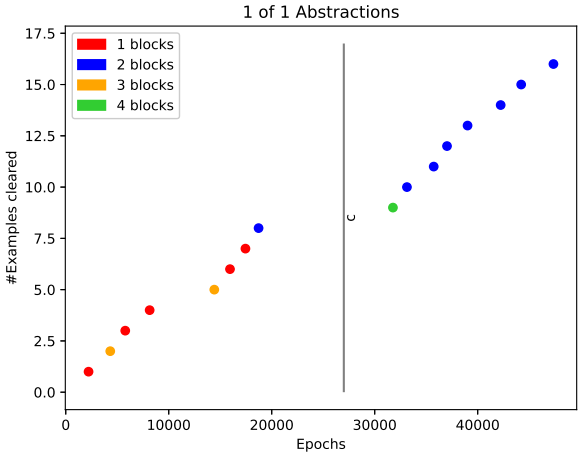

Figure 4.7: If agents are able to discover abstractions, they require significantly fewer than 50 000 epochs to learn to build the shapes.

also the last example with a c. The agents that discovered an abstraction were able to solve that example a lot faster, which was the main contribution towards being faster. Furthermore, if there were c's later on in the process, the differences would presumably have been even larger, in favour of the agents with an abstraction.

These experiments all point towards abstractions being useful to facilitate learning for scenarios where the shapes contain some inherent structure. Regarding how much information needs to be sent in a previously encountered problem, abstractions (with our size limitations) can only be positive. Therefore, we only need to show that they are used. As is apparent from experiment 2 and 3 being positively impacted from the presence of abstractions, the agents choose to use abstractions.

5
Conclusion

The results clearly show that abstractions are useful for the architect-builder environment in the context of improving the efficiency of a reinforcement learning system. It minimizes how often the agents need to communicate and increases the speed at which the agents find a solution. The method we proposed is sufficient to generate abstractions and the agents utilize such abstractions.

Moreover, the way we introduce abstractions dynamically allows us to inject new concepts without excessive re-training. This is a necessity as the abstractions are dependent on the current solution, and complete re-training may therefore result in different abstractions. Notably, we also show that the abstractions can be manually supervised and the re-training should therefore also work for such abstractions, even if they are made ad hoc.

In conclusion: *Having a language with messages also corresponding to common sequences of actions* ***facilitates*** *the reinforcement learning construction task,* ***in biased environments****. Our neuro-symbolic agent* ***discovers and learns*** *to use such concepts.*

5.1 Future Work

We have only scratched the surface in generating higher order actions in a reinforcement learning setting, and there are many avenues left to explore. In this section, we will mention how we think it could be possible to build on our work.

5.1.1 Larger Scopes

One worry we have is that in environments where the agents are more proficient in completing the task, the abstractions might not be useful for making the learning process faster. Instead, they might only be useful for lowering the number of interactions needed for agents that are already trained. This is based on the conception that neural-networks are generally proficient at generalization and thus might not require the increased complexity of the action space. Thus, allowing higher-order actions may inadvertently increase the total learning times.

5.1.1.1 Environment scopes

To further solidify our results, one improvement is to increase the size of the environment to allow a more complex interaction between possible abstractions. We

briefly tested if the agents that had made the ”C” abstraction could place it twice, to possible results, but we did not have time to investigate this in detail. This is partly due to the limitations of our environment, as we only trained and generated sets for six by six grids, in our experiments. Most other combinations of abstractions would thus be larger than the grid itself. A larger environment would allow the interaction of abstractions as proper ”blocks” for construction, such as placing an ”L” upon an ”upside-down U”. It would also let us more directly compare and investigate the results in McCarthy et al.'s research [16], where they always used pairs of the three shapes in $S_{structured}$. That could lead to potentially interesting behaviour, such as abstractions created by the combination of other abstractions.

5.1.2 Builder Improvement

A core initial thought in the project was to investigate this in a multi-agent setting, but we have not managed to find any interesting representation of the builder that is not a simple classification network. On that note, it would be interesting to try introducing higher-order communication in the more traditional multi-agent RL setting, where every agent has the same goal.

5.1.3 Improvements to the Wake-Sleep-Dream cycle

Furthermore, an effective way to prune or replace abstractions as they are deemed unnecessary could help improve the network, but this poses several issues in the general learning process of a neural network in itself.

To improve the pre-training of new abstractions introduced in the dreaming phase, it would be reasonable to also re-train the neural connections on the cases where the abstractions are not used. As no negative reinforcement is used in the current solution, a newly introduced abstraction is proportionally overrepresented as to their use cases. However, as a neural system is self-regulating, it will result in the over-representation being proportionally adjusted towards a realistic level.

A deeper or more complex model for evaluating abstractions could benefit the agents, as the generative function and its evaluation are simple and based on naïve assumptions. This could lead to more valuable abstractions, or the capability to keep intermediate, less specialised, abstractions.

In our current version, we are unsure if we have a proper evaluation of the actual utility of abstractions. It might even be the case that it varies wildly based on the problem. Therefore, some more dynamic version of evaluation such as a neural network may yield better results.

5.1.3.1 Re-construction of Existing Abstractions

One issue with the current version of the sleeping method is the construction of abstractions that already exists. As the model does not benefit from more than one representation of an abstraction, we would like to avoid duplicating them. One example of this, which we encountered, is the $\cap$ shape. It can be constructed by first placing a vertical block, with a horizontal on top, and then doing it again to the side.

Alternatively, one can build both vertical blocks first, then both horizontal blocks. Therefore, the order can be irrelevant. However, as we know that our problem is not symmetrical, as $(A_{h1}, A_{v1}) \neq (A_{v1}, A_{h1})$. The combination of order sometimes being irrelevant with it sometimes being relevant creates issues. This problem is not easy to solve, as we do not know if two abstractions are equal before testing [40].

For a small environment as ours, this is not a big issue, as it is simple to test candidate abstractions to see if they are the same as an already existing abstraction. However, as the environment scales up, this problem becomes more pronounced. We have incidentally remedied one of the problems, which is that the same abstraction would get suggested as a candidate over and over. As one abstraction is created, it will reduce the number of actions used to solve the cases when that abstraction is relevant. Therefore, the estimated utility of such abstractions decreases, since the abstraction counts as one action.

However, when the environment scales up, more of the abstractions should have several ways of construction. Since we cannot determine if two abstractions are equal without testing, we do not know in the sleep phase if two or more suggestions for an abstraction are equal. Thus, the number of times such abstractions occurs can get split between multiple options and thus be disproportionally valued.

5.1.4 Other Possible Environments

In general, we believe that many cooperative tasks could benefit from the concepts laid out in this paper. This is due to communication being particularly useful if the environment is, at least partly, cooperative. In particular, communication is to great benefits if the agents observe different parts of the environment. Where the environment is not a zero-sum game, communication can often be used to improve cooperation between agents.

Another possible environment to research this type of question would be a cooperative invisible grid world [41]. Two agents, A and B, would exist in two separate grid worlds, W_a and W_b, but A can only "see" W_b and B can only see W_a. In such a case, both of the agents would need to communicate aspects of the other's environment to be able to complete their task. Furthermore, the amount of data allowed to be communicated could be limited, by a cost or hard cap, in order to deter the agents from sending all their observations. Then would have an incentive to send as short messages as possible, while still being descriptive enough for the other agent to complete their task. This environment has the benefit that both agents are of the same type, which some architectures prefer.

5.1.5 Different network architecture

The neural networks we used were simple and perhaps not the best suited to the issue. We tried briefly to incorporate convolutional layers, which had minor improvements in results for the non-grouped action space. It also created many issues in network structure and flexibility. However, the prospect of including recurrent neural networks or transformers seems to have potential. They have both seen to be powerful in general settings such as in GPT-3 and particularly useful for envi-

ronments where order matters, much like the environment which we have used in our work.

5.1.5.1 Multiagent bidirectionally coordinated nets

By using a bi-directional recurrent neural network, it is possible to model the dependency between agents over hidden layers, instead of a Q-learning model. The benefit over a simple greedy solution is that the communication can happen in latent space, allowing higher order information to be transmitted between the agents, and propagate gradient updates for all agents [42].

The model was shown to work by a simple metric, where multiple sequential agents should guess the cumulative sum of a randomly assigned value for each agent. The agents knew both their own value and were allowed to send a message to the next agent. In their experiments, the message converged to the cumulative sum so far. The answer got more accurate for each agent in the queue. This method has been proved to work for both hetero and homogeneous agents, and different, distinct strategies depending on the environment. Implementing such a network structure could result in another methodology and area to work with in the context of generating linguistic abstractions, while possibly reducing the epochs required compared to the requirements in a Deep Q-network

Bibliography

[1] W. McCarthy, R. Hawkins, C. Holdaway, H. Wang, and J. Fan, "Learning to communicate about shared procedural abstractions," in *Proceedings of the 43rd Annual Conference of the Cognitive Science Society*, 2021.

[2] T. Mikolov, A. Joulin, and M. Baroni, "A roadmap towards machine intelligence," in *Computational Linguistics and Intelligent Text Processing*, pp. 29–61, 2018.

[3] J. Foerster, I. A. Assael, N. de Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[4] A. Lazaridou and M. Baroni, "Emergent multi-agent communication in the deep learning era," 2020.

[5] F. Hill, A. K. Lampinen, R. Schneider, S. Clark, M. Botvinick, J. L. McClelland, and A. Santoro, "Environmental drivers of systematicity and generalization in a situated agent," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*, 2020.

[6] F. Hill, O. Tieleman, T. von Glehn, N. Wong, H. Merzic, and S. Clark, "Grounded language learning fast and slow," 2020.

[7] E. Jorge, M. Kågebäck, F. D. Johansson, and E. Gustavsson, "Learning to Play Guess Who? and Inventing a Grounded Language as a Consequence," 2016.

[8] A. Lazaridou, A. Peysakhovich, and M. Baroni, "Multi-agent cooperation and the emergence of (natural) language," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, pp. 1–11, 2017.

[9] M. Kågebäck, E. Carlsson, D. Dubhashi, and A. Sayeed, "A reinforcement-learning approach to efficient communication," *PLoS ONE*, vol. 15, no. 7, pp. 1–26, 2020.

[10] S. Havrylov and I. Titov, "Emergence of language with multi-agent games: Learning to communicate with sequences of symbols," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[11] I. Mordatch and P. Abbeel, "Emergence of grounded compositional language in multi-agent populations," 2018.

[12] J. Mu and N. Goodman, "Emergent communication of generalizations," in *Advances in Neural Information Processing Systems*, 2021.

[13] M. Lewis, D. Yarats, Y. N. Dauphin, D. Parikh, and D. Batra, "Deal or No Deal? End-to-End Learning for Negotiation Dialogues," *arXiv:1706.05125 [cs]*, June 2017. arXiv: 1706.05125.

[14] M. Noukhovitch, T. LaCroix, A. Lazaridou, and A. Courville, "Emergent Communication under Competition," *arXiv:2101.10276 [cs]*, Jan. 2021. arXiv: 2101.10276.

[15] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," *arXiv:1912.06680 [cs, stat]*, Dec. 2019. arXiv: 1912.06680.

[16] W. McCarthy, R. Hawkins, C. Holdaway, H. Wang, and J. Fan, "Learning to communicate about shared procedural abstractions," in *Proceedings of the 43rd Annual Conference of the Cognitive Science Society*, 2021.

[17] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum, "DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, p. 835–850, 2021.

[18] P. Dayan, G. E. Hinton, R. M. Neal, and R. S. Zemel, "The Helmholtz Machine," *Neural Computation*, vol. 7, pp. 889–904, 09 1995.

[19] C. C. White III and D. J. White, "Markov decision processes," *European Journal of Operational Research*, vol. 39, no. 1, pp. 1–16, 1989.

[20] K. Gurney, T. J. Prescott, J. R. Wickens, and P. Redgrave, "Computational models of the basal ganglia: from robots to membranes," *Trends in Neurosciences*, vol. 27, no. 8, pp. 453–459, 2004.

[21] S.-i. Amari, "Natural Gradient Works Efficiently in Learning," *Neural Computation*, vol. 10, pp. 251–276, 02 1998.

[22] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[23] M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. Wald, and E. Muharemagic, "Deep learning applications and challenges in big data analytics," *Journal of Big Data*, vol. 2, p. 1, Dec. 2015.

[24] D. J. Foster, A. Krishnamurthy, D. Simchi-Levi, and Y. Xu, "Offline reinforcement learning: Fundamental barriers for value function approximation," 2021.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[26] T. Gupta, "Deep Learning: Feedforward Neural Network," Dec. 2018.

[27] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *arXiv preprint arXiv:1811.03378*, 2018.

[28] G. Hinton, N. Srivastava, and K. Swersky, "Neural networks for machine learning lecture 6a overview of mini-batch gradient descent," *Cited on*, vol. 14, no. 8, p. 2, 2012.

[29] R. Ward, X. Wu, and L. Bottou, "Adagrad stepsizes: Sharp convergence over nonconvex landscapes," 2018.

[30] R. Kemker, M. McClure, A. Abitino, T. Hayes, and C. Kanan, "Measuring Catastrophic Forgetting in Neural Networks," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[31] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming catastrophic forgetting in neural networks," 2016.

[32] B. L. C. T. Z. T. J. M. D. Z. R. Y. Wenpeng Hu, Zhou Lin, "Overcoming Catastrophic Forgetting via Model Adaptation."

[33] T.-Y. Tung, S. Kobus, J. R. Pujol, and D. Gunduz, "Effective Communications: A Joint Learning and Communication Framework for Multi-Agent Reinforcement Learning over Noisy Channels," *arXiv:2101.10369 [cs, eess, math, stat]*, Apr. 2021. arXiv: 2101.10369.

[34] L. Matignon, G. J. Laurent, and N. Le Fort-Piat, "Independent reinforcement learners in cooperative Markov games: a survey regarding coordination problems.," *Knowledge Engineering Review*, vol. 27, pp. 1–31, Mar. 2012.

[35] X. Zhao, J. Lei, and L. Li, "Distributed policy gradient with variance reduction in multi-agent reinforcement learning," *CoRR*, vol. abs/2111.12961, 2021.

[36] S. Lu, K. Zhang, T. Chen, T. Başar, and L. Horesh, "Decentralized policy gradient descent ascent for safe multi-agent reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, pp. 8767–8775, May 2021.

[37] S. Omidshafiei, J. Pazis, C. Amato, J. P. How, and J. Vian, "Deep decentralized multi-task multi-agent reinforcement learning under partial observability," in *International Conference on Machine Learning*, pp. 2681–2690, PMLR, 2017.

[38] T. George Karimpanal and R. Bouffanais, "Self-organizing maps for storage and transfer of knowledge in reinforcement learning," *Adaptive Behavior*, vol. 27, no. 2, pp. 111–126, 2019.

[39] M. Stevens, "Playing tetris with deep reinforcement learning," 2016.

[40] R. M. Kline, "Grammar undecidable problems." June 2022.

[41] S. Thrun and M. L. Littman, "Reinforcement learning: an introduction," *AI Magazine*, vol. 21, no. 1, pp. 103–103, 2000.

[42] P. Peng, Y. Wen, Y. Yang, Q. Yuan, Z. Tang, H. Long, and J. Wang, "Multiagent Bidirectionally-Coordinated Nets: Emergence of Human-level Coordination in Learning to Play StarCraft Combat Games," *arXiv:1703.10069 [cs]*, Sept. 2017. arXiv: 1703.10069.