Improving Ad-Hoc Cooperation in Multiagent Reinforcement Learning via Skill Modeling

Ariel Jakub Kwiatkowski

School of Electrical Engineering

Thesis submitted for examination for the degree of
Master of Science in Technology.
Espoo 8.7.2020

Thesis supervisor:

Alexander Ilin

Thesis advisor:

Antti Keurulainen

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Ariel Jakub Kwiatkowski

Title: Improving Ad-Hoc Cooperation in Multiagent Reinforcement Learning via Skill Modeling

Date: 8.7.2020 Language: English Number of pages: 6+64

Supervisor: Alexander Ilin

Advisor: Antti Keurulainen

Machine learning is a versatile tool allowing for, among other things, training intelligent agents capable of autonomously acting in their environments. In particular, Multiagent Reinforcement Learning has made tremendous progress enabling such agents to interact with one another in an effective manner. One of the challenges that this field is still facing, however, is the problem of ad-hoc cooperation, or cooperation with agents that have not been previously encountered.

This thesis explores one possible approach to tackle this issue, using the psychology-inspired idea of Theory of Mind. Specifically, a component designed to explicitly model the skill level of the other agent is included, to allow the primary agent to better choose its actions.

The results show that this approach does in fact facilitate better coordination in an environment designed to test this skill, and is a promising method for more complicated scenarios.

The potential applications can be found in any situation that requires coordination between multiple intelligent agents (which may also include humans), such as traffic coordination between autonomous vehicles, or rescue operations where autonomous agents and humans have to work together to efficiently search an area.

Keywords: machine learning, multiagent systems, reinforcement learning, artificial intelligence, theory of mind

Preface

I'd like to thank Antti Keurulainen for proposing the topic and guiding the work leading up to this thesis. I'd also like to thank Alexander Ilin for his invaluable advice on the direction of the research and helping in the process of writing this thesis.

Contents

Symbols and abbreviations

Symbols

$\mathbb{R}$	the set of real numbers
$\mathbb{R}^{n\times m}$	the set of all $n \times m$ real matrices

Operators

$\bar{A}$	cardinality of a set A
ΔA	the set of all probability distributions over a set A
$A \to B$	the set of all functions from a set A to a set B
$x \oplus y$	the direct sum, or concatenation of vectors x and y
$x \odot y$	the Hadamard (component-wise) product

Abbreviations

AI	Artificial Intelligence
MLP	MultiLayer Perceptron
AV	Autonomous Vehicle
RN	Relation Network
RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
ToM	Theory of Mind
TSP	Traveling Salesman Problem
SM	Skill Modeling
GT	Ground Truth
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
MARL	MultiAgent Reinforcement Learning
MDP	Markov Decision Process
PG	Policy Gradient

1 Introduction

The problem of cooperation between intelligent agents (including humans) is one of the biggest challenges facing AI research in recent years [1]. In many fields it is desirable to have autonomous agents capable of cooperating with arbitrary other agents in an efficient and safe manner, but that skill is far from trivial and requires specially designed approaches that take into consideration the unpredictability of other agents.

As an example, consider self-driving cars. If, at some point, one AV (Autonomous Vehicle) manufacturer obtains a 100% market share and pushes any competition out of the picture, it can have complete control over coordination protocols, explicit or implicit – however this is unlikely to be the case. In the foreseeable future, AVs will need to share roads not only with other AVs, but also with humans, and humans are anything but predictable [2]. And yet, they mostly manage to intuitively understand other members of traffic, avoiding too many collisions.

One mechanism that seems to contribute to this phenomenon in humans is the so called **Theory of Mind** [3]. It describes the way that people internally impute mental states to others in a way that, while not directly observable, allows to make predictions on their behaviors. It has been extensively studied and observed to happen both in humans and other primates, and recently it was also shown that it is possible to create AI systems that can exhibit ToM-like properties when observing artificial agents [4].

The main approach to creating artificial autonomous agents is **Reinforcement Learning (RL)** [5], in which the key idea is trial and error. By taking (at first, randomly) actions in the environment and combining them with obtained rewards, it can be inferred which actions were beneficial, and which were harmful. After collecting a batch of experience, the agent is modified to take better actions, and the cycle continues.

In particular, **Deep Reinforcement Learning (DRL)** is a way of training RL agents by using deep neural networks. This approach has proven to be effective, for example in the games of Chess and Go [6], in Atari games [7] and in the multiplayer game DOTA 2 [8]. Some approaches to DRL involve learning the expected value of each state of the environment, which is then used to choose the next action. Others, which will be the focus here, directly optimize a policy function mapping states to actions so that it reaches a maximal reward on average.

When several agents are acting together in a single environment, **Multiagent Reinforcement Learning (MARL)** [9] becomes relevant. This carries many new challenges, such as non-stationarity [10] and the lazy agent problem [11] which makes it more difficult to accurately perform credit assignment.

Machine Learning [12] is a fundamental concept underlying Reinforcement Learning describing a mechanism for building programs (or mathematical models) which are not fully specified by the human designer. Instead, they only provide a general structure of the algorithm, and then selecting its parameter via an automatic process of training given some data.

Deep Learning [13] is a modern take on Machine Learning, where deep neural networks are utilised. They're extremely versatile algorithms that can be adapted for all sorts of applications like Computer Vision [14], Natural Language Processing [15], Visual Understanding [16], Reinforcement Learning [7] and more. They are also proven to be capable of approximating arbitrary functions with enough hidden units and given sufficient data [17].

The problem of ad-hoc multiagent cooperation can also be seen as a form of meta-learning, which is a general way of training agents on a variety of tasks, so that they can efficiently solve other, previously unseen tasks [18]. In this case, each potential partner agent is a different task (in the meta-learning sense), so by learning to cooperate with various partners, the agent can become better at cooperating with new ones.

An interesting property of multiagent learning systems trained via self-play is that they develop autocurricula [19] allowing for an emergent progression from simple to complex challenges. In competitive games, this can be understood as the fact that since the agent is its own opponent, it has to learn to counter any new strategy it creates. In a cooperative setting, it can take the form of social dilemmas, where agents need to make a choice between their self-interest and the collective goal.

This thesis focuses on the applications of ToM-based approaches to improve RL-produced agents' ability to cooperate with other agents that have not previously been encountered in the training process. In particular, the skill level of an agent is postulated to be a metric that's relevant for decision-making that involves cooperation.

To evaluate this hypothesis, I tested a neural network architecture designed to explicitly learn to estimate the other agent's skill level, and then use that information in deciding the policy. The agent is then evaluated with a range of partner agents on various skill levels.

All of that is performed in an gridworld-based environment, following common practices of AI Safety research [20] for testing novel ideas. The agents cooperate in a task of foraging their world for subgoals, and then reaching a final goal, while sharing the reward signal making it a fully cooperative setting.

1.1 Thesis goals and problem statement

The main goal of this thesis is exploring a new way of facilitating cooperation in multiagent reinforcement learning system by introducing a component dedicated to learning the skill level of its partner. This architecture is then evaluated in specially designed environments, and compared both with a baseline model, and one that receives the skill level as an input.

Leading up to this, I describe the theory of Reinforcement Learning and its multiagent variant, including the standard approaches and formalism, as well as less-known techniques that are relevant for the experiments, such as the Relation Network.

1.2 Structure

The theoretical background on Machine Learning and Neural Networks is introduced in Section 2, followed by a description of Reinforcement Learning, including the problem formalism and standard algorithms in Section 3.

Following that, Section 4 describes the experimental setup, consisting of the environment in which the RL agents act, the architecture of the agents themselves and the training procedures for policy optimization.

Section 5 describes the results of the aforementioned experiments, and Section 6 wraps everything up with a summary of the entire work.

2 Machine Learning

The goal of this and the following section is building up a coherent theory of all the key concepts used in this thesis, with as much mathematical rigor as reasonably possible. In the end, all the necessary notation should be introduced, justified and ready to be used in the description of the actual task and experimental design. To this end, I'll take a bottom-up approach, starting from elementary concepts, and ending with the specific niche of Multiagent Reinforcement Learning.

Machine Learning in its most general form can be described as a set of methods and algorithms that are only specified with a general architecture and a certain objective, and then use data to fully specify the algorithm in the process of training. It is generally divided into three parts: **supervised learning** where the data includes specific labels that are to be learned, **unsupervised learning** where the data doesn't contain a uniquely given objective, and **reinforcement learning**, the focus of this work, where the algorithm interacts with an environment, generating its own data, optimizing for a given reward function. Due to their respective relevance, I will provide a brief description of supervised learning now, skip unsupervised learning, and expand on reinforcement learning further on in this section.

2.1 Supervised Learning

In this setting, the task can be described function approximation. Given an input space $\mathcal{X}$, an output space $\hat{\mathcal{Y}}$, a label space $\mathcal{Y}$, a dataset $\mathcal{D} = \{(x_i, y_i)\}$ where $x_i \in \mathcal{X}$, $y_i \in \mathcal{Y}$, and a loss function $\mathcal{L}\colon \hat{\mathcal{Y}} \times \mathcal{Y} \to \mathbb{R}$, you want to find a function $f\colon \mathcal{X} \to \hat{\mathcal{Y}}$ called a **model** that minimizes the mean loss over the dataset:

$$\frac{1}{\bar{\mathcal{D}}} \sum_{i=1}^{\bar{\mathcal{D}}} \mathcal{L}(f(x_i), y_i)$$

Note that it is not true that $\mathcal{Y} = \hat{\mathcal{Y}}$ even though quite often it happens to be the case, e.g. in regression problems where $\mathcal{Y} = \hat{\mathcal{Y}} = \mathbb{R}$. In the other main type of supervised learning, which is classification, the label space is a discrete set of categories C such that $\bar{C} = M < \infty$. The output space can be the real vector space $\hat{\mathcal{Y}} = \mathbb{R}^M \cong (C \to \mathbb{R})$ interpreted as log-probabilities of a distribution from the set of all probability distributions on C, denoted ΔC. It's also isomorphic to the set of functions $C \to \mathbb{R}$ mapping each category to its log-probability.

However, in this case we can choose a selection function $\iota \colon \hat{\mathcal{Y}} \to \mathcal{Y}$ that maps the model's predictions to actual labels, e.g. $\iota(\hat{y}) = \arg\max \hat{y}$ choosing the highest-probability element in C.

An important concept to keep in mind when talking about Machine Learning is **generalization**, since in actual use cases, we want the algorithm to work well also on new, previously unseen inputs. That's why it is useful to view the dataset $\mathcal{D}$ as a finite random sample from a joint probability distribution $p(x, y)$. In this case, the task is to find a good approximation of the conditional probability $p(y|x)$, quantified as minimizing the expected loss

$$\mathop{\mathbb{E}}_{x,y \sim p(x,y)} \mathcal{L}(f(x), y)$$

With this formulation, it is no longer sufficient to just set $\forall_{(x_i,y_i)\in\mathcal{D}} f(x_i) \coloneqq y_i$, which is the trivial solution to the problem with a finite dataset. Instead, the model needs to, in a sense, understand the underlying structure of the dataset so that it performs well on the entire distribution.

2.2 Neural Networks

A neural network is a versatile machine learning algorithm that can be applied to supervised, unsupervised and reinforcement learning. It is a parametrized function composed of one or more layers, with each layer being a linear transformation followed by an activation function applied component-wise to its output.

If, as is usually the case, the input and output spaces are real vector spaces, we have $\mathcal{X} = \mathbb{R}^N$ and $\hat{\mathcal{Y}} = \mathbb{R}^M$. In the most common variant of a neural network called the multi-layer perceptron (MLP), the n-th layer can be then expressed as

$$x_n = g(W_n x_{n-1} + B_n) \tag{1}$$

where x_n is called the n-th hidden layer (with $x_0 = x$ being the input to the network), $g(\cdot) \colon \mathbb{R}^{l_n} \to \mathbb{R}^{l_n}$ is an activation function, l_n is the size of the n-th layer such that $x_n \in \mathbb{R}^{l_n}$. The parameters of the layer are the **weight** matrix $W_n \in \mathbb{R}^{l_n \times l_{n-1}}$ and the **bias** vector $B_n \in \mathbb{R}^{x_n}$. Often, the weight and the bias are collectively referred to as the weights of the layer (or of the network) and are denoted by θ.

Quite often, the activation function is in fact just a function $g \colon \mathbb{R} \to \mathbb{R}$, and is applied to the input vector elementwise, by considering each component separately. It also needs to be differentiable w.r.t. its input almost everywhere, which will be necessary for training the network. The most common choice is ReLU (Rectified Linear Unit) [13] defined as $\mathrm{ReLU}(x) = \max(0, x)$. It corresponds a very rough idea of a biological neuron, where the inputs are summed, and if the total exceeds some threshold, the neuron fires proportionally to the inputs.[1] Other common choices include the hyperbolic tangent $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ and the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. In Figure 1 you can see a visual comparison of each of these functions.

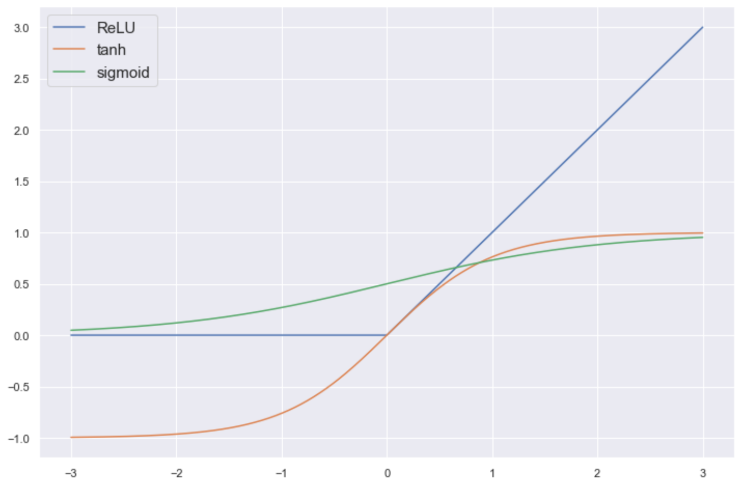


Figure 1: A comparison of the most commonly used activation functions: ReLU, tanh and sigmoid. Note that the sigmoid function is the only one where $f(0) \neq 0$ which can cause issues in deep networks, but it does fulfill the property $\forall_{x \in \mathbb{R}}\ 0 < f(x) < 1$ which makes it suitable as a final activation function in situations where we want the output to be interpretable as a probability.

A Feedforward Neural Network, or Deep Neural Network, can then be constructed by stacking several linear transforms as defined in Equation 1 interspersed by activation functions. The word "deep" here refers to the number of layers – the **depth** of the network. The dimensionality of each hidden representation h_n is called the **width** of the layer, or the number of hidden units. This is the same as the size of the x_n activation vector. A simple schematic representation of a neural network can be seen in Figure 2.

In summary, a neural network can be viewed as a parametrized function $\mathrm{NN}_\theta(x)$ differentiable almost everywhere w.r.t. θ and x, with a specific layer-based internal structure that allows for flexibility and modularity (note: $\mathrm{NN}'_{\theta'} \circ$

[1]Keep in mind, this is **not** how biological neurons actually work – but it works well for artificial neural networks, and biological accuracy is not a goal in itself.

NN_θ is also a neural network as long as the dimension of the output of NN_θ is the same as the dimension of the input of $\text{NN}'_{\theta'}$).

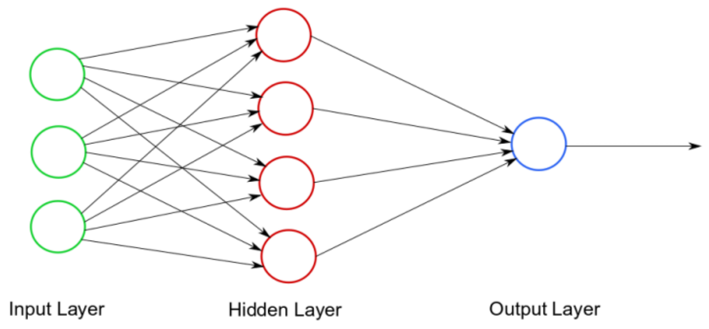


Figure 2: A graphical representation of a neural network. Each node represents one component of a vector (either input, hidden or output), and each edge represents a component of a weight matrix.

2.2.1 Gradient Descent

By far the most common way of training neural networks – by which I mean, finding a set of weights for which the network manages to approximate the target distribution $p(y|x)$ – is the **Gradient Descent** algorithm, or one of the many algorithms that are based on it. At its heart, it is a fairly simple idea dating back to Augustin-Louis Cauchy in 1847 [21] when he applied it to the problem of solving systems of simultaneous equations. To describe it informally: first, you choose an arbitrary argument (here: arbitrary weights) as a starting point. Then, evaluate the gradient of the loss function at that point, and update the weights in the direction of the negative gradient – and repeat until convergence. A more formal description can be found in Algorithm 1.

In practice, a commonly used approach is called Stochastic Gradient Descent (SGD), where instead of computing the gradient using the entire dataset at once, only a subset (or batch) of data is used at each iteration. This allows for more frequent updates, and counteracts overfitting due to extra stochasticity.

2.2.2 Adam optimizer

In many applications, including this work, the Gradient Descent algorithm only describes the main idea of the optimization method. The exact algorithm used to find the optimal weights is Adam [?], which is a modification of Gradient Descent that allows for adaptive learning rates for each weight individually,

Algorithm 1 Gradient descent

```
1: Require: $\alpha$ : learning rate, a real number
2: Require: $f(\theta)$ : the function to be optimized
3: Require: $\theta_0$ : Initial parameter vector
4: $t \leftarrow 0$
5: while $\theta_t$ not converged do
6:     $t \leftarrow t + 1$
7:     $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
8:     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot g_t$
9: return $\theta_t$
```

helping the network find better weights in a shorter time. The details of Adam can be found in Algorithm 2.

Algorithm 2 Adam optimization algorithm. Good default settings are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$. All operations on vectors are element-wise, and β_1^t, β_2^t denote β_1 and β_2 to the power t.

```
1: Require: $\alpha$ : stepsize
2: Require: $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates
3: Require: $f(\theta)$ : Function to be optimized
4: Require: $\theta_0$ : Initial parameter vector
5: $m_0 \leftarrow 0$
6: $v_0 \leftarrow 0$
7: $t \leftarrow 0$
8: while $\theta_t$ not converged do
9:     $t \leftarrow t + 1$
10:    $g_t \leftarrow \nabla_\theta f(\theta_{t-1})$
11:    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
12:    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
13:    $\hat{m}_t \leftarrow m_t \backslash (1 - \beta_1^t)$
14:    $\hat{v}_t \leftarrow v_t \backslash (1 - \beta_2^t)$
15:    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t \backslash (\sqrt{\hat{v}_t} + \epsilon)$
16: return $\theta_t$
```

2.2.3 Backpropagation

The remaining piece necessary to get a neural network-based machine learning algorithm is a way to compute the gradients $\nabla_\theta f(\theta)$, where

$$f(\theta) = \frac{1}{\bar{\mathcal{D}}} \sum_{i=1}^{\bar{\mathcal{D}}} \mathcal{L}(\mathrm{NN}_\theta(x_i), y_i)$$

In conjunction with the Gradient Descent algorithm or its modification like Adam, this would be enough to get an algorithm finding θ minimizing the loss as defined above.

The way the gradient can be obtained is by using the layered structure of neural networks and an algorithm called **backpropagation** [22]. The essence of this approach is computing the gradient w.r.t. the output layer's weights, and using the derivative chain rule to iteratively get the gradient w.r.t. each previous layer. The exact procedure is laid out in Appendix A.

2.3 Recurrent Neural Networks

In applications that possess a temporal structure, it can be beneficial to use models that have some sort of memory or internal state to allow for maintaining information between consecutive timesteps. For this purpose, Recurrent Neural Networks [13] have been introduced. The core idea here is that the network makes uses of a hidden state $h^t \in \mathbb{R}^{n_h}$ that's updated between each consecutive timestep in the following way:

$$h^t = g(W_{hh}h^{t-1} + W_{hx}x^t + B_h)$$

where h^t is the state and x^t is the input at the t-th timestep. To properly compute the gradients of the loss function, it is necessary to use a slightly modified version of the backpropagation algorithm, called Backpropagation Through Time [23], which takes into consideration the temporal structure and ensures proper accumulation of gradients.

2.3.1 Long Short-Term Memory Networks

In practice, RNNs turn out to have difficulties in retaining long-term relations across the timesteps, which prompted creating an enhanced version called the Long Short-Term Memory Network, or LSTM [15] which deals with this issues by introducing learnable gates in the state update rule. Specifically, an LSTM cell is updated as follows:

$$\begin{aligned}
f_t &= \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\
\tilde{C}_t &= \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\
h_t &= o_t \odot tanh(C_t)
\end{aligned}$$

where i_t is called the **input gate**, f_t – **forget gate**, c_t – **cell state**, o_t – **output gate**. A graphical representation of this computation can be seen in Figure 3.

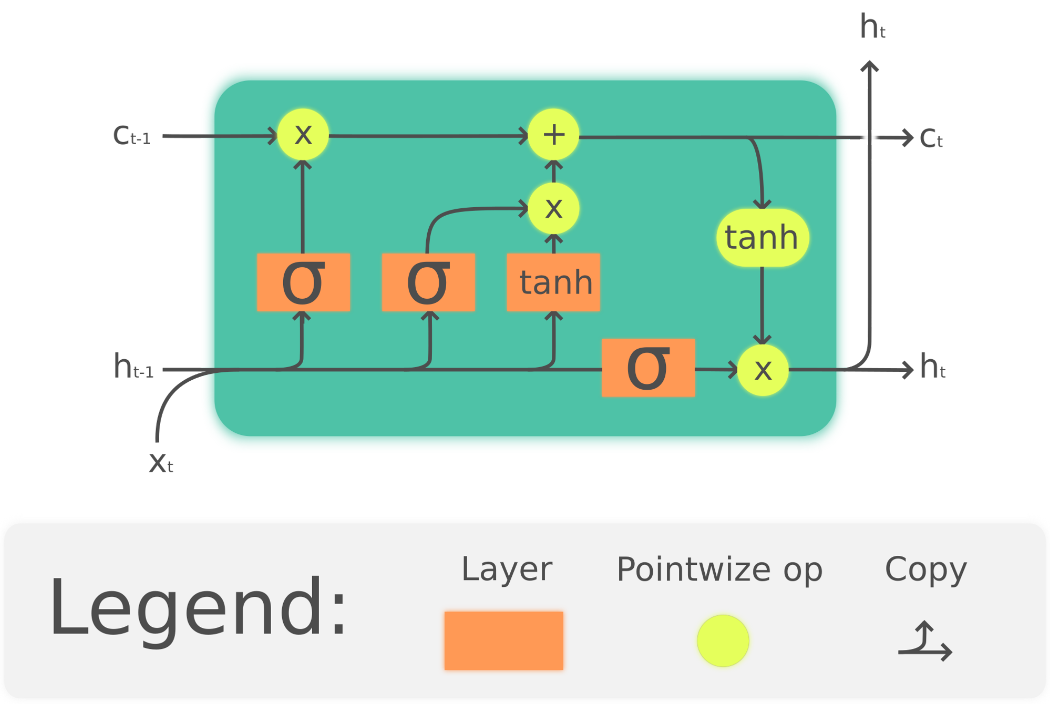


Figure 3: A visual depiction of the computation within an LSTM cell [24].

2.4 Relational Networks

In some applications, the network needs to process the positions of several identical objects. By default, a regular MLP network has to learn this property from experience, but with an appropriate architecture, this property can be enforced in the network, making the training easier. This is the purpose of the Relation Network (RN) [25], which in its original form was designed to enhance image processing applications. It works by processing the representations of each object pair independently, and then summing the outputs together – this way, due to the commutative property of addition, indistinguishable objects are processed in the same way and the output of the RN is invariant under the operation of swapping them.

The original formulation of RN is as follows:

$$\mathrm{RN}(O) = f_\phi\left(\sum_{i \leq j} g_\theta(o_i \oplus o_j)\right)$$

where f_ϕ and g_θ are neural networks, $O = \{o_i\}$ are objects present in the input, and the $\oplus$ operator is the direct sum, here equivalent to the concatenation of its operands.

In this work, the Relation Network has been adapted to the specific situation of processing entities in a gridworld-based reinforcement learning environment. Gridworlds are explained in more detail in Section 3.1.5. Firstly, the representation of each object is as follows:

$$o_i = x_i \oplus y_i \oplus f_i \oplus e_{t_i}$$

where (x_i, y_i) are the grid coordinates of the object o_i normalized to the range $[0, 1)$, $f_i \in \mathbb{B}$ is a flag whose meaning depends on the type of the object, for example indicating whether an object is active or inactive, t_i is the type of the object, and $e_t \in \mathbb{R}^{n_e}$ is the embedding of the object type t. The embedding is a trainable vector that allows the network to distinguish between objects of different types, while treating all objects of the same type the same way.

What's more, in RL applications, one object is the most important – the agent itself. The other objects are relevant mostly in relation to it, while their relations between each other are not as significant, which is why instead of summing over all pairs of objects (quadratic in the number of objects), the agent's representation is always used as the first object in the direct sum, so the final formulation of the relation network becomes

$$\mathrm{RN}(O) = f_\phi \left(\sum_i g_\theta(o_0 \oplus o_i) \right) \tag{2}$$

where o_0 is the agent's own representation.

This architecture can be used as an input layer for other models. For example, the output of the RN can be then used as the input to an LSTM cell, for what is called RNLSTM later in this work.

3 Reinforcement Learning

Reinforcement learning is the third of the aforementioned types of machine learning in which the task is to find a certain **policy** mapping states of the environment, to actions that can be taken by the agent. Unlike in supervised learning, there are not labels that can be used as a direct objective. In fact, there is not even a single specified dataset to learn from. Instead, the reinforcement learning problem is defined by an **environment** and a **reward** which defines the objective.

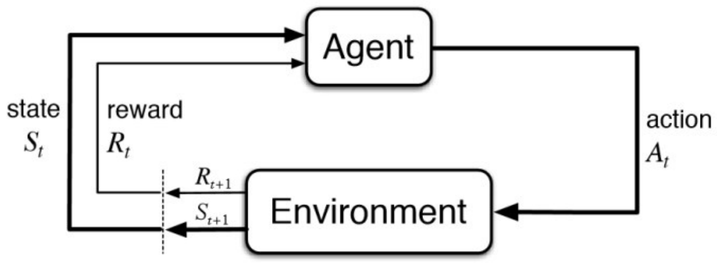


Figure 4: A schematic depiction of a reinforcement learning problem [26].

As depicted in Figure 4, the **agent** (which is the RL equivalent of a model) takes actions in the environment and changes it in some way. Then it receives the updated observation, as well as a reward signal.

3.1 Markov Decision Processes

A Markov Decision Process, or MDP, is a mathematical formalism common for specifying RL problems. To be precise, an MDP is defined as a tuple (S, A, T, R) where S is a set of states, A is a set of actions, $T\colon S \times A \to \Delta S$ is the transition function determining the probabilities of states based on the previous state and the action taken, and $R\colon S \to \mathbb{R}$ is the reward function[2] defining the objective to be optimized.

On the agent side, we have a **policy** $\pi\colon S \to \Delta A$ whose purpose is choosing the next action, possibly in a nondeterministic way. Together, they produce **trajectories** obtained via the following process: start with an initial state s_0, choose an action a_0 based on $\pi(s_0)$ (e.g. via sampling from the distribution[3])

[2] Depending on the source, the reward function can be defined differently, e.g. as $R\colon S \times A \to \mathbb{R}$. Those definitions are equivalent and out of convenience, I'll use the one present in the main text

[3] In some cases, especially with deterministic policies, it might be beneficial to use a different method for the sake of exploration.

and observe the next state s_1 sampled from the distribution $T(s_0, a_0)$ and the reward $r_0 = R(s_1)$. The process is repeated until the end of the episode, yielding a trajectory $\tau = \langle s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_n, a_n, r_n \rangle \in \mathcal{T}$. Through slight abuse of notation, we can then define the total reward of a trajectory as $R(\tau) = \sum_{t=0}^{n} r_t$.

To solve an MDP is to find a policy π^* which maximizes the expected total reward when evaluated in the environment:

$$\pi^* = \arg\max_{\pi} \mathop{\mathbb{E}}_{\tau \sim \pi} R(\tau)$$

where $\tau \sim \pi$ denotes the process of sampling actions from the policy described above.

It is important to denote that a policy π can also be interpreted as a conditional probability function $\pi(a|s) \in \mathbb{R}$. Both of these formalisms are equivalent and are often used interchangeably in literature.

3.1.1 Partially Observable Markov Decision Processes

While POMDPs [27] are not used in this work, they are still worth mentioning due to their prevalence in Reinforcement Learning. They reflect the idea of imperfect information – instead of observing the real state s_t, the policy receives an observation generated by a function $O\colon S \times A \to \Delta\Omega$ where Ω is the set of possible observations. The observation in step t depends on the new state s_t and the previous action a_{t-1} which led to that state.

Formally, a POMDP is a tuple (S, A, T, R, Ω, O). It is a generalization of regular MDPs, since each MDP can be converted into one by setting $\Omega = S$ and $O(s, a) = \delta_s$. It also turns out that each POMDP can be represented as an MDP that combines the transition and observation mechanisms in its own transition function.

Trajectories and total rewards in POMDPs work the same way as in MDPs, with the only difference that the input to the policy is $o \sim O(s, a)$ rather than s directly.

3.1.2 Decentralized POMDPs

A generalization of POMDPs that allows for multiple agents is called Dec-POMDP [28]. It takes the approach of splitting the environment action into a joint action taken by multiple agents, each of which receives its own observation. Formally, a Dec-POMDP is a tuple $(S, \{A_i\}, T, \{R_i\}, \{\Omega_i\}, O)$ where $\{A_i\}$ is the set of actions that can be taken by the agent i, $\{\Omega_i\}$ are that agent's possible observations and R_i and each agent's reward functions. By considering the joint actions $A = \times_i A_i$ and joint observations $\Omega = \times_i \Omega_i$ this can be viewed as a POMDP, however it places a limitation in the information shared between agents.

3.1.3 Time horizon and discount factor

One component that I omitted from the definition of MDPs and their generalizations is the **discount factor** γ. Its purpose is ensuring that even for infinite horizon environments, the total reward of a trajectory remains finite as long as the rewards themselves are bounded.

$$R(\tau; \gamma) = \sum_{t=0}^{\infty} \gamma^t r_t < \infty \tag{3}$$

In this work, the considered environments maintain a finite horizon – so while its value would still have some impact on the learned policies, it loses its theoretical justification and is therefore set to $\gamma = 1$.

The intuition behind the discount factor is that it determines how much attention is paid to the distant future, as compared to the near future. In the case of $\gamma = 1$, sacrificing 0.1 reward in this step to receive 0.2 reward in the next step is well worth it. Alternatively, with $\gamma = 0.25$, the respective discounted rewards are 0.1 and 0.05, which makes the other decision to seem better.

For this reason, the discount factor can be seen in two ways – either as a property of the MDP itself, or as an implementation detail of the algorithm solving it. The result is largely the same, but the latter option gives more flexibility in the possibility of adjusting the discount factor to generate different behaviors. What's more, in many environments of interest, there is no obvious value for the discount factor, and it has to be chosen arbitrarily anyways, which is why I chose to not include it in the definition of MDPs.

In practice, the discount factors used are often very close to 1, like 0.9 or 0.99. This allows for infinite-horizon problems to get the finite total reward property, while not making the agent too myopic. In finite-horizon problems, the discount rate is often equal to 1.

3.1.4 Value function

An important concept defined for Markov Decision Processes is that of a **value function**. Intuitively, it is meant to represent the expected reward that will be collected starting from a certain state. However, there is not a unique way of choosing the actions in the process of unrolling an episode, hence different versions of a value function can be used depending on the context.

Given a certain policy π, we can take the approach of choosing the following action from that policy, giving the value of a state s under the policy π [26]

$$V^\pi(s) = \mathbb{E}_\pi\{R_t | s_t = s\} = \mathbb{E}_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s \right\} \tag{4}$$

If the policy π is the optimal policy π^*, we can also consider the optimal value function $V^*(s)$ which is the “true" value of a state in the environment.

Furthermore, it’s often useful to consider the state-action value function $Q^\pi(s, a)$, where the process is similar: in the state s you take the action a, and then proceed to act according to the policy π. These two views are equivalent as long as we have access to the MDP transition dynamics:

$$V(s) = \max_a Q(s, a) \tag{5}$$

$$Q(s, a) = \mathop{\mathbb{E}}_{s' \sim T(s,a)} R(s, a) + \gamma V(s') \tag{6}$$

Alternatively, the value function can be defined recursively using the Bellman equation [29]. There, the value of a state is expressed as:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \mathop{\mathbb{E}}_{s' \sim T(s,\pi(s))} V^\pi(s') \tag{7}$$

And the value under the optimal policy is given by the Bellman optimality equation:

$$V^*(s) = \max_a R(s, a) + \gamma \mathop{\mathbb{E}}_{s' \sim T(s,a)} V^*(s') \tag{8}$$

3.1.5 Gridworlds

A gridworld is a useful type of an MDP that allows to abstract away real-world mechanics to focus on a specific property of the environment being considered. They are especially suitable for basic research, where the subject of investigation is fundamental principles and mechanisms, rather than real-world efficiency, most notably in the area of AI Safety [30].

It has a certain width w and height h, and contains $w \times h$ grid cells aligned in a rectangle. Each of those grid cells contains 0 or more objects from a set of objects $\mathcal{O}$, some of which may be agents. Each object has a position in grid coordinates (x, y), as well as a type from a finite set of types T.

As an example, consider a labyrinth gridworld in which a single agent needs to reach a goal, navigating through a maze of walls which together occupy W grid cells. It would have the types $T = \{agent, goal, wall\}$, and the objects $\mathcal{O} = \{agent, goal, wall_1, \ldots, wall_W\}$[4]

There are various ways of representing a gridworld. For the purposes of RL agents, it is convenient for the representation to have the form of a real-valued tensor. The most straight-forward way is by using the grid topology directly, along with the binary encoding of each cell's contents. In other words, the state of a gridworld would be a tensor $s \in \mathbb{R}^{w \times h \times \bar{\mathcal{O}}}$. This however leads to very large and sparse state tensors and can be very inefficient. It also doesn't generalize to cases with a different number of objects.

A simple modification would involve using the objects' types instead, causing the state representation to stay in the constant space $s \in \mathbb{R}^{w \times h \times \bar{T}}$ as long as the number of types remains constant. This relies on the assumption that objects of the same type are indistinguishable, and that there can't be two objects of the same type in the same grid cell. In both of these cases, extra information can be conveyed by concatenating it to the tensor in a way dependent on the specific situation.

If the gridworld is relatively sparse, a dense state representation might be more suitable. The basis here is concatenating the coordinates of objects in a fixed order, with $s = [x_1,\ y_1,\ x_2,\ y_2,\ \ldots,\ x_{\bar{\mathcal{O}}},\ y_{\bar{\mathcal{O}}}]^\intercal \in \mathbb{R}^{2\bar{\mathcal{O}}}$. This representation can be supplemented with extra information by injecting it after each (x,y) coordinate pair, as long as the order remains consistent.

An alternative used in this work is using Relational Networks. Using the notation from Section 2.4, we can set $\mathcal{O} = O$ with the representation of each object as described there

$$o_i = x_i \oplus y_i \oplus f_i \oplus e_i$$

where the type embedding e_i corresponds to one of the types in T.

[4]Using *agent* and *goal* in both the object and type sets is a slight abuse of notation – in general the two sets can be completely different

This method has many properties that make it especially suitable for generalization in the reinforcement learning setting. First of all, it explicitly introduces a symmetry with respect to the operation of swapping objects of the same type, which can significantly simplify the task of the learning algorithm. What's more, it allows for seamless generalization to different gridworld sizes, different object counts, and even different type counts. A model trained with a certain set of objects can be easily used in a different environment with more (or fewer) objects, which wouldn't be possible with a dense representation since its input layer's size would have to be adjusted. Due to the coordinate normalization, it can also be used in a larger environment, which wouldn't be possible with the grid-based representation.

3.2 Policy Gradient

The main task of a Reinforcement Learning algorithm is to find a policy $\pi(s)$ which maximizes the expected return when evaluated in an environment. While there are various methods of approaching this, the one used in this work is based on the idea of **policy gradients**.

In a way, it is the most direct way of solving it in the spirit of machine learning. The reward function can be seen as a function mapping policies to real numbers (by evaluating them in the environment), and if that is then differentiable with respect to the policy's parameters, you can simply use gradient ascent to find the optimal parameter set.

In other words, we're treating RL as a standard optimization problem,[5] where the goal is to maximize the following reward function:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} R(\tau) \tag{9}$$

Assuming proper differentiability properties of the policy, this can be solved via gradient ascent, which is the same thing as gradient descent (see: Section 2.2.1) with the update multiplied by -1, and therefore can also use the modified approaches like the Adam algorithm.

Of course, this would require an expression for an estimate of the gradient $\hat{\nabla}_\theta J(\theta)$ in terms of the trajectory or its components (i.e. observations, actions

[5]Note that while we can to some extent treat it as such, it is a bit more complicated than that since the data itself changes depending on the current policy weights, adding an extra layer of complication over typical supervised learning.

and rewards). It turns out that this is, in fact, possible to obtain analytically, and the final **policy gradient** equation is as follows:

$$\nabla \mathbb{E}_\pi \left[R(\tau)\right] = \mathbb{E}_\pi \left[R(\tau)\nabla \log \pi(\tau)\right] = \mathbb{E}_\pi \left[\left(\sum_t R(s_t)\right) \nabla \log \left(\prod_t \pi(a_t|s_t)\right)\right] \tag{10}$$

The full derivation can be found in Appendix A. The main value of this equation is in the fact that the gradient can be estimated from empirical data using only the rewards and the probabilities of actions taken in the trajectory, both of which can be sampled by simply rolling out the environment with a differentiable policy.

An important observation is that due to the Markovian assumption, actions taken at step t' cannot affect rewards obtained in earlier steps $t < t'$. Since adding a baseline value to the rewards does not change the expected value of the policy gradient, this means that the total reward $R(\tau) = \sum_t R(s_t)$ can be replaced with the reward-to-go $\hat{R}_t = \sum\limits_{t'>t} R(s'_t)$ without increasing the bias of the estimation, but reducing its variance.

This is the basis of the REINFORCE algorithm [31] described in Algorithm 3.

Algorithm 3 REINFORCE, a basic Policy Gradient algorithm.

```
1: Require: $\theta_0$: initial policy parameters
2: Require: $\alpha$: learning rate
3: $k \leftarrow 0$
4: while $\theta_k$ not converged do
5:     Collect a batch of trajectories $\{\mathcal{D}_k\}$ with the policy $\pi_{\theta_k}$
6:     Compute rewards-to-go $\hat{R}_t$
7:     Compute the policy gradient estimate as:
           $$\hat{g}_k = \frac{1}{\bar{\mathcal{D}}_k} \sum_{\tau\in\mathcal{D}_k} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)|_{\theta_k} \hat{R}_t$$
8:     Update the policy weights using gradient ascent or a related algorithm:
           $$\theta_{k+1} = \theta_k + \alpha \hat{g}_k$$
9:     $k \leftarrow k + 1$
10: return $\theta_k$
```

An important thing to keep in mind is that due to the fact that the sampling is performed according to the current policy, a single batch of data can only be used for a single gradient update, making it an on-policy algorithm. What's more, this estimate tends to have a very high variance. Finally, since the reward is summed across the entire episode, credit assignment becomes an issue and the agents might have difficulties recognizing which actions led to good outcomes, and which were associated with high returns by coincidence.

Due to these problems (sample inefficiency, high variance, credit assignment), REINFORCE is rarely used in practice and is infeasible for most RL problems. However, similarly to gradient descent, modified approaches have emerged, designed to counteract those issues, one of which will be described now.

3.2.1 Proximal Policy Optimization

The PPO algorithm [32] attempts to fix the aforementioned issues by including some changes to the algorithm that allow it to work more efficiently, even if it is not quite as elegant theoretically.

First of all, to improve the credit assignment issue, it utilizes an actor-critic approach, with a value prediction network used to estimate the expected return of a certain state. This is formalized as the **advantage** estimated as $A_t = \hat{R}_t - \hat{V}(s_t)$ With this, it is possible to determine whether a certain action was better than expected ($A_t > 0$) or worse ($A_t < 0$). The value network can either be separate from the policy, or can share some of the weights to take advantage of the multitask learning phenomenon.

Using a learned baseline (i.e. subtracting a constant value from the reward function) also helps reduce the variance without introducing any bias – although in the early stages of the training, the value estimation is completely random, once it gets better, it can give a good estimation of the state's value, making the advantage more estimations reliable.

The largest difference, and the thing that makes PPO unique, lies in the loss function it uses, the so called **surrogate loss**. To recap, a standard PG loss that uses the advantage values can be expressed as the following:

$$L(s, a, \theta) = A^{\pi_\theta}(s, a) \log \pi_\theta(a|s) \tag{11}$$

This, after differentiation, gives the same expression as the equation for the policy gradient, so it can be used as a loss function for automatic differentiation software.

In PPO, however, the corresponding equation is more complicated:

$$L(s,a,\theta,\theta_k) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\pi_{\theta_k}}(s,a)\right) \tag{12}$$

Starting with the difference in the signature of this function – the reason for this is that in the PPO algorithm, each batch of data can be used for several gradient updates. To counteract that the data becomes off-distribution after the first update, an importance sampling term $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ has to be included, where θ_k is the weights before performing any updates with the current batch.

The clip function discourages the policy from going too far from its original values. A small modification to the weights can have a significant impact on the policy, and combined with the fact that the loss function relies on data generated by the “old" policy, this means the policy could easily stray too far and experience a significant performance decrease, which might turn out not to be recoverable.

Finally, the min operator between the clipped and unclipped expressions (which are otherwise identical) means that we only really want to clip really high values of the loss. If it is low, the gradient ascent algorithm shouldn't be moving in that direction anyways, so there's no harm in having that part unclipped.

3.2.2 Advantage estimation

An important improvement to the policy gradient algorithm is using the advantage function as opposed to the raw rewards to optimize the policy. This idea, stemming from actor-critic algorithms [33] gives the agent a better mechanism for performing credit assignment.

The core idea is that if the agent is given an estimate of the current state's overall value, then it can compare the actual reward obtained through taking a certain action to know if that action was better or worse than expected.

To this end, the network needs some way of estimating the state value, which can be learned with another neural network, as described in the PPO algorithm. With this, a common way of estimating the advantage values is by using the Generalized Advantage Estimation (GAE) algorithm [34].

When thinking about advantage estimation, the question of how to actually estimate the value of the state given a trajectory. In episodic tasks, it can be viable to just rollout the entire episode and sum up the discounted rewards to go through Monte Carlo estimation:

$$\hat{V}(s_t) = \sum_{t' \geq t} \gamma^{t'-t} R_{t'} \tag{13}$$

On the other hand, you can also use the existing value estimate of the next state with a Temporal Difference-based approach, which doesn't require observing the entire episode, and is better suited for continuing tasks:

$$\hat{V}(s_t) = R_t + \gamma \hat{V}(s_{t+1}) \tag{14}$$

Both of these approaches come with pros and cons. What's more, various intermediate methods are easy to define, by considering the next n rewards and the $(n+1)$th value.

GAE introduces a way of interpolating between all of those by introducing a special parameter $\lambda \in [0, 1]$. The GAE advantage estimate can then be computed as

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V \tag{15}$$

where $\delta_t^V = R(t) + \gamma V(s_{t+1}) - V(s_t)$ is the TD error at timestep t.

By adjusting the λ parameter, the algorithm can put more or less emphasis on the value estimation. In the extreme cases, $\lambda = 1$ reduces to the Monte Carlo estimate from Equation 13, and $\lambda = 0$ produces the TD estimate from Equation 14.

3.2.3 Exploration - Exploitation trade-off

While training a Reinforcement Learning agent, there appears a dilemma regarding how it should choose its actions. On one hand, its objective is to optimize the total reward by **exploiting** its current knowledge. On the other hand, it has an instrumental goal of obtaining more knowledge to be able to fulfill the aforementioned task more effectively, which it can do by the way of **exploration**, or taking actions that do not seem optimal at the moment.

The general rule is that in early stages of the training, agents should focus on exploration to build up their knowledge, and then transition to exploitation by taking the best action as the training goes on. In some cases, the RL agent is **deterministic**, which means it doesn't have a natural way of sampling suboptimal actions. Then, a common strategy is called ϵ-greedy in which a value $\epsilon \in [0, 1]$ is set (and possibly varied throughout the training), and the agent takes a random action with probability ϵ, and the optimal action with probability $1 - \epsilon$.

In case of **stochastic** algorithms, which include policy gradient-based ones, the policy itself is a probability distribution over all possible actions, so the actions can be directly sampled from it, and over the course of the training, the randomness should decrease naturally, if it is beneficial in that particular environment (there are, after all, examples in game theory where optimal strategies are stochastic).

However, even for stochastic policy gradient algorithms, exploration can be enhanced by adding an **entropy bonus**. Recall the formulation of the PG training objective in Equation 9, extended by an additional term proportional to the entropy of the policy $S(\theta)$:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} R(\tau) + \lambda S(\theta) \tag{16}$$

where $\lambda \in \mathbb{R}$ is the entropy bonus coefficient, and $S(\theta)$ is the expected entropy of the policy π_θ. In practice, this can be computed from collected experience by computing the average entropy of distributions $\pi_\theta(s) \in \Delta A$ encountered in the collected trajectories.

3.3 Multiagent Reinforcement Learning

A generalization of Reinforcement Learning is Multiagent Reinforcement Learning (MARL) where it is possible for several agents to interact with the environment and, by extension, one another [35]. While it has clear applications in real-life scenarios where agents have to interact with other agents, including humans, it also carries many challenges.

The various MARL environments can be divided into three main categories, for simplicity described on the example of two agents A_1, A_2:

- Cooperative – the reward of a state is equal for all agents $R_i(s)$
- Competitive – one agent's gain is another's loss, with $R_1(s) + R_2(s) = const$, the equivalent of a constant-sum game in Game Theory
- Mixed – anything that doesn't fit in the other two categories

3.3.1 Non-stationarity and agent modeling

A standard assumption in RL algorithms stemming directly from the Markovian assumption is that the environment is stationary, meaning that the transition $T(s, a)$ always yields the same distribution, or in other words, is an actual function that remains the same throughout the training [10].

In a multiagent setting (Dec-POMDP) observed from the perspective of a single agent, that is not quite the case. During the course of training, the second agent's policy changes, and because of that even if the action taken is the same, the joint action will be different, yielding a different successor state distribution.

One approach to dealing with this issue is the MultiAgent Deep Deterministic Policy Gradient (MADDPG) algorithm [36] based on Deep Deterministic Policy Gradient [37], an algorithm combining policy gradient and value-based methods. Specifically, MADDPG uses a centralized critic shared between the agents to facilitate implicit information exchange and coordination between the agents.

This work explores an alternative approach based on agent modeling. In principle, given full information about the partner agent and including it in the state representation, the problem becomes stationary again. However, since a neural network-based agent can be arbitrarily large, I'm using an intermediate step where the policy is conditioned on the information about the other agent's skill level – this will be explained in further detail in Section 4.

3.3.2 Self-play

An approach commonly used especially in competitive environments is self-play, where the agents are copies of each other. It has been shown to achieve great success for example in the game of Go with AlphaZero [6]. With self-play, the agents can develop a natural training curriculum making it possible to learn advanced and versatile strategies [38]. The core mechanism here is that if the agent learns a new effective tactic, it can immediately start learning a way to counter it.

This phenomenon, however, does not occur in cooperative environments – there it is quite possible for the agent to overfit to only cooperating with itself, making it incapable of cooperating with other, arbitrary agents. To some extent, this can be mitigated by including old versions of the agent in the training, which is the approach taken in this work, explained in more detail in

Section 4.3.

3.3.3 Ad-Hoc Cooperation

In a standard ML context, it is quite simple to define generalization – the model should learn to perform well on data it hasn't seen before, but that comes from the same distribution as the training data. It is not as simple in the case of MARL, where there are different types of generalization to be considered. Grover et al. [39] introduce a valuable classification in the context of policy representation learning which has an "external" observer, but similar types can be applied to agents observing themselves along with their partners or opponents.

One axis to consider is generalization across tasks, which is relevant also in single-agent RL. Given a family of MDPs, we want an agent trained on its subset to perform well on all of them, including ones it hasn't seen yet. In many cases, this is included in the environment design by randomizing the starting positions of various objects, like the agent or a goal it has to reach.

Then, specifically for MARL, there is the issue of generalization across agents. Often it is desirable to have an agent capable of performing well with a wide range of partner agents. So on one hand there is **weak generalization** which refers to cooperation between agents which appeared throughout the training but not with each other. **Strong generalization**, in turn, corresponds to acting with a partner that has never been encountered in the training by the agent.

Finally, this work focuses on a slightly different direction of **skill generalization** – a robust agent should be able to adapt to partners performing either well, or poorly, coordinating with them or assisting in doing the tasks they are not doing due to the lower skill level. Standard iterative RL approaches can easily provide an excellent testbed for this, since early versions of the agents are usually randomly initialized and can be seen as unskilled, later versions are skilled, and there's a wide range of intermediate states.

3.4 Meta-learning and mesa-optimization

Meta-learning is a concept tightly connected to generalization, most commonly used in the context of the aforementioned task generalization. While standard learning algorithms (whether Machine Learning or specifically Reinforcement

Learning) learn to effectively perform a certain task, possibly generalizing across some parameters of the task, the goal of meta-learning is learning to learn to solve different tasks, using different, albeit in some way related, environments.

Meta-learning was introduced by Jurgen Schmidhuber in his Master's thesis [40], and the work that was particularly inspiring here was the Model-Agnostic Meta-Learning framework [18]. It uses a general algorithm involving a double training loop – one across different tasks, and another one across the timesteps of the task. It requires computing second order derivatives (Hessian matrices) of the loss function, although that can become computationally intractable with large networks, so a first-order approximation was also shown to perform well.

A related idea is that of **mesa-optimization** [41] which is considered primarily in the context of AI safety and value alignment. It comes into play whenever an outer optimizer (meta-optimizer) is used to create an agent acting as the inner optimizer (mesa-optimizer). Using meta-learning algorithms carries the risk of value misalignment, where the two optimizers' rewards differ from one another.

In the context of multiagent RL, a natural form of meta-learning is generalizing across various partner agent – by training with some of the possible partners, it is desirable for the agent to learn to adapt to a new one efficiently. Whenever it uses a recurrent policy (e.g. an LSTM network), the agent's inner state updates to reflect its knowledge about the partner agent can also play the role of a mesa-optimizer, making this issue relevant even with an unchanging environment.

3.5 Theory of Mind

Theory of Mind [3] is a concept stemming from Cognitive Sciences, describing how humans (and other animals) build an inner belief state regarding others' belief states. In human children, for example, theory of mind capabilities have been shown to develop between the ages of 3 and 5 [42] which constitutes an important step in their ability to function in a society alongside other people.

A common way of recognizing whether a system (e.g. a child, an animal, or a computer program) is capable of modeling others is the so called Sally-Anne Experiment. [43] It involves two children (or, in general, agents): Sally and Anne, as well as an external observer whose capabilities we're evaluating. The experiment proceeds as follows:

1. Sally and Anne are in a room with a basket and a box, Sally has a marble
2. Sally puts the marble in the basket
3. Sally leaves the room
4. Anne removes the marble from the basket and places it in a box next to the basket
5. Sally returns to the room

A cartoon used to depict the experiment in the original paper can be seen in Figure 5.

Now, the question to the observer is: where will Sally look for the marble? The correct answer, obviously, is the basket – this is the last place where she saw it, and there's no way for her to know that it is been moved. However, a naive system incapable of reasoning about Sally's beliefs would point to the box instead, as that's where the marble is actually located, it overwrites Sally's belief with its own beliefs.

Since it is important for humans, it can also be expected to be a useful ability for artificial agents to possess in order to cooperate with other heterogeneous agents in a shared environment, which is bound to be necessary in case of wide-spread RL agents in public life.

Rabinowitz et al. [4] have shown that this is, in fact, possible for data-driven systems in their Machine Theory of Mind algorithm which observes agents acting in an environments and predicts their beliefs and intents.

A more applied approach has been taken by Foerster et al. [44] in the Bayesian Action Decoder, followed up by the Simplified Action Decoder [45]. There, using explicitly updated predictions of the other agents' belief states has shown a significant improvement in their abilities to cooperate in the game of Hanabi [1].

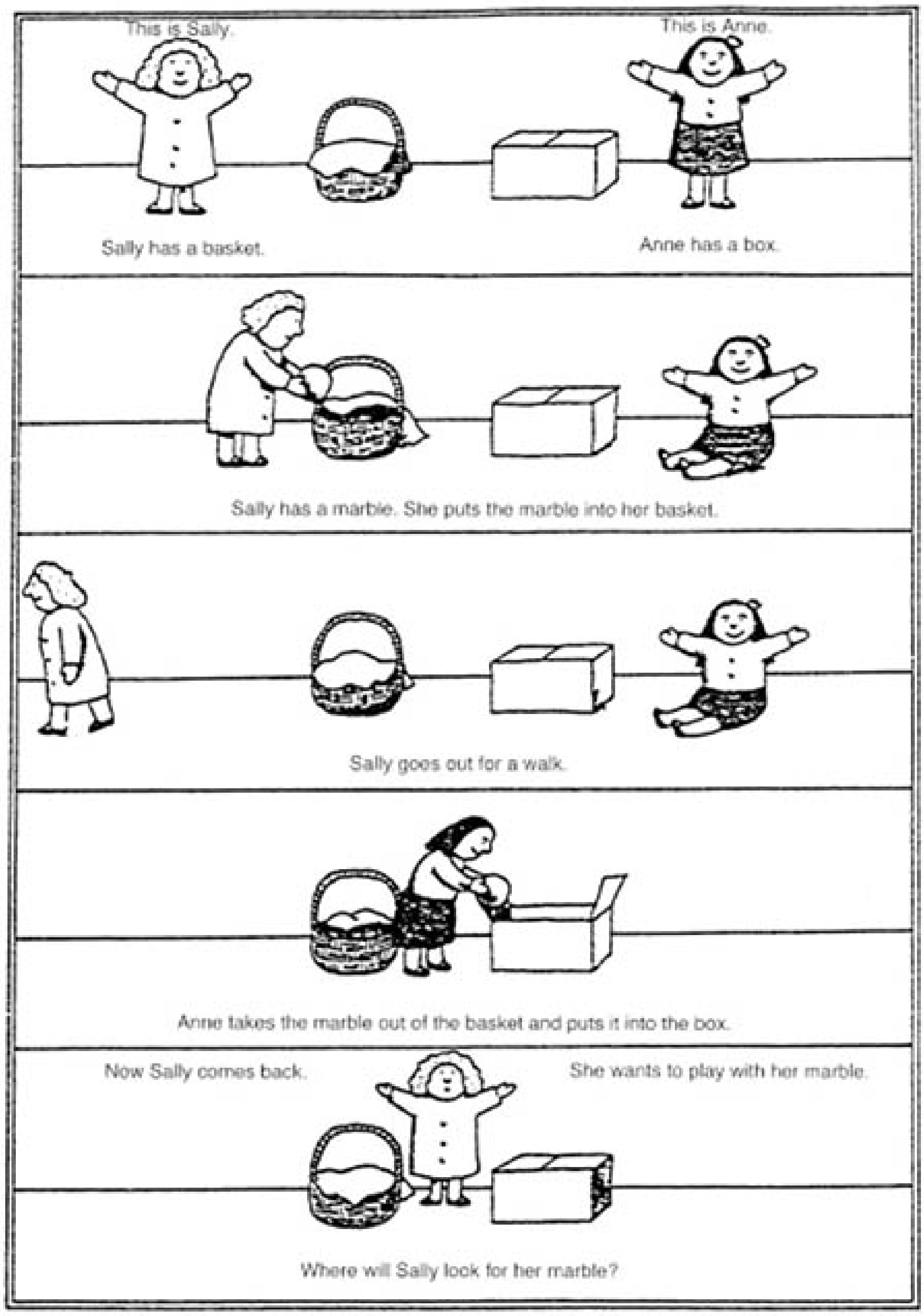

This is Sally.
This is Anne.
Sally has a basket.
Anne has a box.
Sally has a marble. She puts the marble into her basket.
Sally goes out for a walk.
Anne takes the marble out of the basket and puts it into the box.
Now Sally comes back.
She wants to play with her marble.
Where will Sally look for her marble?

Figure 5: The original cartoon describing the Sally-Anne test [43].

4 Experiments

This section will detail the experimental procedure used to test the skill modeling approach, including the specific models used, as well as the environments in which they were trained.

4.1 Environment

The environments used to evaluate this approach are variations of the base foraging environment, inspired by the Traveling Salesman Problem. They were designed as to accentuate the properties of cooperation between agents, and in some cases, making them explicit to enable directly measuring the cooperation skill.

The environment is a fully observable, fully cooperative (meaning that the reward is shared by both agents), deterministic, decentralized MDP on a 7×7 gridworld with two agents, N_S subgoals and one final goal. Using the notation from Section 3.1.5, that means we have the object and type sets as follows:

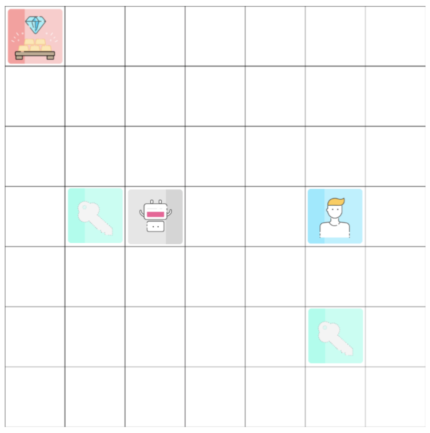
$$\mathcal{O} = \{Agent_1,\ Agent_2,\ Subgoal_1,\ \ldots,\ Subgoal_{N_S},\ Goal\}$$
$$T = \{Agent_1,\ Agent_2,\ Subgoal,\ Goal\}$$

In each timestep, both agents can take one of five actions:

$$A_i = \{north,\ south,\ west,\ east,\ stay\}$$

Each of them moves the agent in the respective direction by one grid cell (or makes it stay in place, for the *stay* action).

Informally speaking, the agents' goal is to collect each of the subgoals by stepping on them, and then reaching the final goal. More precisely, the first time one of the agents steps on a subgoal, they both receive a fixed reward and the subgoal is marked as collected. After all subgoals have been collected, stepping on the final goal (by at least one of the agents) also gives a positive reward and ends the episode. To encourage finishing the episodes quickly, the environment also gives a small negative reward at each timestep. A visual representation of the environment can be seen in Figure 6a.


(a) Base environment

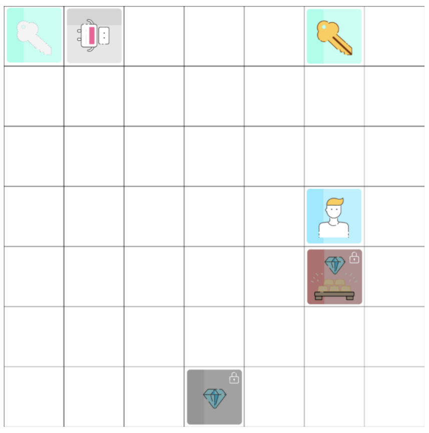

(b) Two goals environment

Figure 6: A visualization of the environment in the a) base and b) two goal variants. The meaning of icons is as follows: robot – first agent, human – second agent (whose skill level determines the reward), gold key – uncollected subgoal, empty key – collected subgoal, diamond with gold – the sole final goal, or the goal giving a larger reward with a skilled partner goal, diamond – goal giving a larger reward with a novice partner. The shading of the goals indicates whether they are active, i.e. whether all subgoals have been collected.

4.1.1 Two goals variant

This version was designed as a simplified case in which the task of estimating the other agent's identity is made more explicit. Agents are trained and evaluated in two configurations only: either a trained agent with a copy of itself, called the expert or skilled partner, or with a randomly initialized novice agent. There are two distinguishable final goals G_1, G_2 that give different rewards R_1, R_2 upon collection, but collecting either of them results in terminating the episode. This means that the sets $\mathcal{O}$ and T now each contain an additional element corresponding to the second final goal and its type.

If the second agent is a trained agent, we have $R_1 > R_2$, and conversely, if the second agent is a novice, $R_2 > R_1$. This way, the first agent is encouraged to build up an internal model if its partner's skill level, to choose the final goal appropriately. A depiction of this environment can be seen in Figure 6b.

Although this environment loses some of the nice theoretical properties by entangling the reward function with the identity of one of the agents (therefore de facto ceasing to use the Dec-MDP formalism), it serves well as a stepping stone towards the full system. So while the multiagent cooperation aspect is not as emphasised, there is a clear focus on agent modelling, allowing to see whether the approach used in this work can fulfill that goal.

4.1.2 Final action variant

Here, the guiding principle is putting an explicit emphasis on the cooperation mechanism between the agents, by incorporating a second phase inspired by the game-theoretic concept of coordination using a fully cooperative Assurance Game [46].

In this game, whose payoff matrix can be seen in Table 1, each player A_0 and A_1 has two available actions: cooperation C and non-cooperation N. Both (C, C) and (N, N) are Nash Equilibria, however only one of them is Pareto-optimal. What's more, to realistically model the scenario of cooperating with partners of various skill levels, unskilled agents are more likely to not cooperate (N) rather than cooperate (C).

		A_0	
		C	N
A_1	C	$1, 1$	$0, 0$
	N	$0, 0$	$0.3, 0.3$

Table 1: Payoff matrix of the Assurance Game

To make it consistent with the rest of the environment, the action C is represented by the action corresponding to movement north in the gridworld, and the action N is represented by any other action. This way, a randomly initialized agent has a 20% chance of cooperating.

Staying in the simplified realm of only two types of agents (expert and novice) and assuming perfect information about the other agent's identity, the optimal behavior in the Assurance Game when acting with a skilled partner is, naturally, the Pareto-optimal joint action (C, C) with the expected reward of 1. With a novice partner, the optimal action can be found by finding the expected utility as a function of the probability of cooperating (to consider both pure and mixed strategies):

$$\begin{gathered} U(p) = 0.2\,p + 0.8 \cdot 0.3\,(1-p) = -0.04\,p + 0.24 \\ p^* := \arg\max_p U(p) = 0 \\ U(p^*) = 0.24 \end{gathered} \tag{17}$$

In other words, the optimal action with a partner whose probability of cooperating is 20% is deterministically not cooperating. This way, the joint action will be (N, N) 80% of the time, and (N, C) 20% of the time.

The way it is implemented in the environment is that now the game has two phases. In the first phase, the agents have to collect subgoals and head to the final goal as usual. After that, however, instead of ending the episode, the agents have to choose a single action in the Assurance Game. This way, they can use the first phase to make a judgement about the other's skill level, and then use that information to choose the correct strategy, extending the two goal variant with an actual coordination mechanism.

4.2 Models

In order to solve those environments, I tested three types of models, each with a different way of utilising Theory of Mind parameters (here: the skill level) visualized in Figure 7.

The first model is the baseline that doesn't involve any explicit skill modelling components. It consists of a relation layer (see Section 2.4) to serve as the input state embedding, followed by a recurrent LSTM layer, and the heads for policy and value predictions.

The second model is the main variant investigated in this work, the Skill Modelling (SM) network. While the model itself doesn't receive any extra inputs when choosing actions, it receives an extra reward signal, making it perfect for the centralized training - decentralized execution (CTDE) approach. An intermediate layer is trained to output its partner's skill level, which is also fed through a bottleneck layer and concatenated with the relation layer's output, going into the main policy LSTM.

Finally, the last model is the Ground Truth (GT) model which serves as an estimate of the best performance attainable by any skill modeling agent. Since the skill modeling capabilities of the SM model are isolated from the rest of the network via the bottleneck, the best it can do is output the labels that it is trained to predict, so feeding that value directly into the policy network provides an upper bound of its performance.

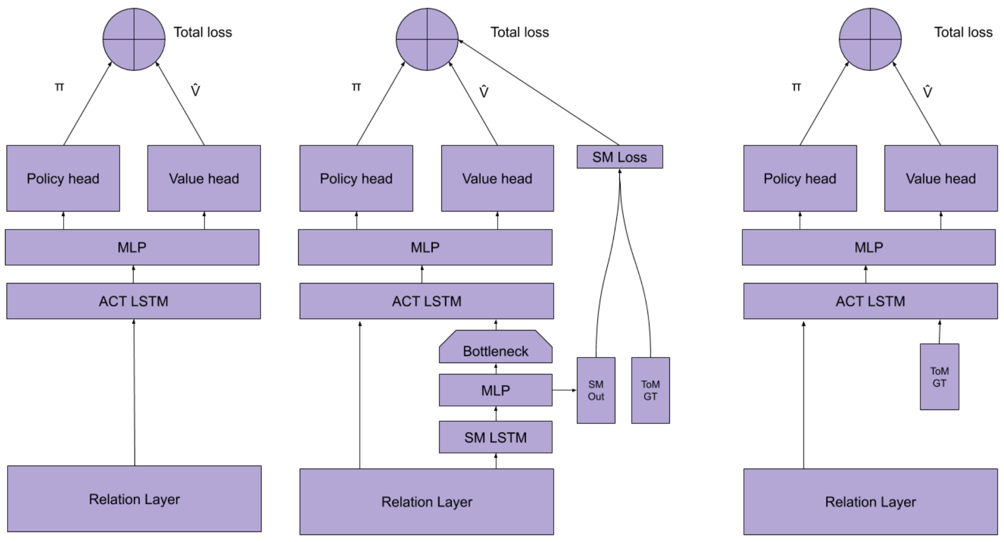


Figure 7: A schematic of the models used for the policy and value functions.
(left) Baseline model where no explicit skill level information is used
(middle) Skill Modelling (SM) model where there's a component trained to predict the skill level
(right) Ground Truth (GT) model where the skill level is directly passed as an input to the network

4.2.1 Stateless models

Some experiments were also performed on a less powerful, stateless model architecture, with analogous variants as described above (i.e. baseline, SM and GT; note that the SM model is then no longer stateless, but its memory is restricted to modeling the other agent's skill level).

The capabilities of such a model are significantly limited, given that the only information it receives is the snapshot of the current state, so any inference of the partner's skill level cannot take into consideration the episode's history.

4.2.2 Separate skill prediction

Another briefly explored concept is that of a separate skill predictor. This method approaches the problem from the perspective of a purely supervised task – given a population of agents, train a model to predict their skill levels given a sequence of observations gathered throughout the episode. If sufficient performance is reached, the model's output can then in principle serve as an input into a GT agent, ridding it of its dependency on external knowledge for the policy function.

4.3 Training methods

The policy optimization process is based on the idea of self-play, where the agent is partnered either with a copy or an old version of itself. The general training procedure is described in Algorithm 4:

Algorithm 4 The general training procedure

```
1: **Require:** $T$: number of iterations
2: **Require:** $S_s$: number of steps to collect with a copy
3: **Require:** $N_o$: number of partner agents to sample
4: **Require:** $S_o$: number of steps to collect with each sampled partner
5: for $t \in [0, T]$ do
6:     Collect a batch $B_s$ containing at least $N_s$ steps with agents $(A_t, A_t)$
7:     for $i \in [0, N_o]$ do
8:         Sample a past agent $A_n \in \{A_0, \ldots, A_t\}$
9:         Collect a batch $B_o^i$ with at least $N_o$ steps with agents $(A_t, A_n)$
10:    Concatenate batches $\{B_o^i\}_i$ into $B_o$
11:    Concatenate batches $B_s$ and $B_o$ into $B_t$
12:    Update the agent $A_t$ using the data batch $B_t$
13: return $A_t$
```

Collecting a batch of at least N steps works in the way that episodes are rolled out until N steps are collected, and then the last episode is allowed to finish, usually resulting in slightly more than N steps. This is not a problem as the batch size doesn't need to be constant, but allows for computing all advantages and rewards-to-go using Monte Carlo estimation.

Specifically, three methods of partner sampling were considered:

- Binary
- Iteration-based
- Skill-based

In the binary method, it is not so much sampling as it is deterministic partner selection. Each sampled agent is the first, randomly initialized agent A_0. This is a relatively simple setting where the skill level can be represented as a binary value 0 or 1. It is also the only possible method for the Two Goals environment, since its reward structure depends on the partner agent's identity and is inherently binary.

Iteration-based sampling is the naive way of choosing from the entire range of agents (as opposed to just the first one, as in the binary version). In each iteration t, you uniformly sample an iteration $n \in [0, t]$ and choose its associated agent A_n. While reasonable at first glance, this approach has a serious flaw when we consider the way that skill level usually progresses during the training - importantly, that at some point the training curve flattens out as the agents converge. This means that the skill level of sampled agents will over time lean more and more heavily towards higher values, defeating the original purpose of sampling, which is having the agent experience a wide range of partner agents.

Skill-based sampling is a way of alleviating that issue by drawing inspiration from the inverse transform sampling. Instead of sampling an iteration directly, a skill level is sampled. The details of skill representation will be laid out in the next section, but what's important is that the skill level is represented as a number $R \in \mathbb{R}$, and that we have access to the skill levels of all past agents $\{R_t\}_t$. Therefore, a skill value $\tilde{R}$ is sampled from the range $[\min\{R_t\}, \max\{R_t\}]$, and then an agent A_n is chosen in a way that minimizes $|R_n - \tilde{R}|$. This way, a consistent representation of various skill levels is ensured, in return causing newer near-converged agents to be sampled very rarely.

4.4 Skill representation

An important thing to consider is what is actually meant by "skill level" of an agent. Intuitively, it should correspond to the expected episode return obtained by the agent, but the complication is that since the environment is inherently multiagent, you need two agents to actually roll out an episode.

In case of the binary sampling method, this issue is solved easily by assigning the skill level value 0 to the randomly initialized agent and 1 to its newest version. The interpretation of those values differs at each iteration, as 1 corresponds to iteration 10 at $t = 10$, and 100 at $t = 100$, but in the later stages of the training, when the policy starts to converge and stabilize, this should not be a significant problem.

When sampling from the entire range of past agents, it becomes necessary to represent the skill level with better granularity. There are two issues to consider here. First of all, sometimes the agents need to cooperate with an agent of a skill level higher than their own. This means that the skill level input would be larger than anything they've experienced throughout the training, which could have an unpredictable impact on their policy.

The second issue is a bit more subtle - when collecting a batch of data, two copies of the most recent agent are put in the environment together. For a

GT model to work in such a setting, it needs to receive a measure of the skill level when choosing the actions, but it cannot know the skill level value before rolling out some episodes and measuring their returns. This leads to a "chicken and egg" situation where it is impossible for an agent to know its own skill level.

To mitigate this issue, a normalization procedure is used, derived as an extension of the binary skill level labels. Given a history of skill levels $\{R_t\}_t$, at iteration t_0, the agent's own skill level is set to be equal to 1, and the skill level of a sampled agent is normalized linearly so that $\min_t\{R_t\}$ maps to 0 and $\max_t\{R_t\}$ maps to 1. This way, the skill level value observed by any agent is bounded in $[0, 1]$. Those same values can also be used for SM agents as the value their skill modeling components are trying to predict.

4.5 Joint-optimal planning

In order to estimate the duration (and, by extension, the reward) of the optimal strategy in an environment, I used a planner to search for the best joint strategy assuming perfect cooperation and coordination between the agents. Since the core of the task is similar to the multiagent Traveling Salesman Problem which is known to be NP-complete [47], the possibilities of developing an efficient algorithm are limited. However, the number of subgoals considered in the experiments is generally quite low, up to 4, which makes it viable to be solved using a brute-force approach.

To make the problem tractable, I simplified the state space so that each environment is represented by a weighted complete graph in which the nodes are positions of subgoals, goals and the starting positions of agents, and edges represent the distances in the gridworld. Agent actions are then interpreted as going from one position to another, e.g. from a subgoal to a final goal.

In the base environment case, to finish an episode, each subgoal has to be visited, and afterwards the goal. From this a few observations can be made:

- There's no need to visit a subgoal twice
- There's no need to do anything after visiting the final goal
- Each goal has to be visited by at least one agent, and there's no need for another agent to visit it as well
- The last action for one of the agent needs to be going to the final goal

Starting with a set of strategies encompassing all possible permutations of visiting the subgoals for both agents (for a total of $(n!)^2$ joint strategies where n is the number of subgoals), I applied the aforementioned constraints removing all redundant strategies, reducing the size of the search space to $(n+1)!$. Since $(4+1)! = 120$, it is sufficiently fast to just evaluate each of those joint strategies and choose the one which results in the episode finishing the soonest.

As previously mentioned, this assumes perfect coordination, which might not be possible in reality, however it provides an upper bound on agents' performance in an environment with specific object locations. Through a Monte Carlo simulation it is also possible to estimate the optimal mean reward across all possible environments, which yields an approximate upper bound on the performance when averaged across many sampled instances.

4.6 Implementation

In order to ensure proper flexibility and reusability, the code was structured in a way that separated out the different parts of the RL logic, in order to obtain a set of abstractions which can be combined together in any variations. This section will provide an overview of the implementation of the program, as well as the abstractions and how they fit together to create the full training and evaluation pipeline.

4.6.1 Tools

The primary tool used for this project was the Python [48] programming language, specifically Python 3.7. Python is a general purpose language, supporting both object-oriented and functional programming paradigms. Even though it is relatively slow, its simplicity and the fact that it can use libraries written in other, faster languages to handle heavier computations, caused it to become the de facto standard in machine learning applications.

Within Python, the two main frameworks I used for efficient computation were SciPy (with a particular focus on NumPy) [49] and PyTorch [50]. The first was used for general purpose data manipulation using its ndarray data structure, while the latter was used specifically for the deep learning parts: building the neural networks, computing the gradients and updating the weights.

For visualization purposes, I used TensorBoard [51] for tracking the training progress in real time, and Matplotlib [52] supported by Seaborn for all other plots and evaluations after the training.

In order to build the experimental environments, I used the Pycolab library [53]. It provides a set of components useful in building gridworld-based MDPs that can be used for RL research. The environments were built adhering to the interface introduced by OpenAI Gym [54] standardizing the methods an environment should implement for easy use with RL algorithm.

4.6.2 Abstractions

Since the code was structured in a mostly object-oriented way, each of the following abstractions was implemented as a class, possibly holding some of the other abstractions as fields or inheriting from them.

Firstly, there's the BaseModel abstract class inheriting from the PyTorch Module class, which is later used to create specific models (e.g. MLPModel or

RelationModel). Its main feature is the model.forward (...) method which takes the following arguments: the network input, optionally the previous hidden state (for recurrent models), and optionally skill-related information. It has three outputs: an action distribution, the next hidden state, and a dictionary with extra outputs like the value or skill estimation. This interface is common for all models, even though some, e.g. a simple MLP model, ignore the optional inputs and return trivial skill estimates.

The next important abstraction is an Agent which is responsible for connecting a BaseModel object with a Gym-like environment by providing methods like agent.compute_single_action(...) for choosing an action to take, and agent.evaluate_actions (...) to compute quantities necessary for gradient-based optimization.

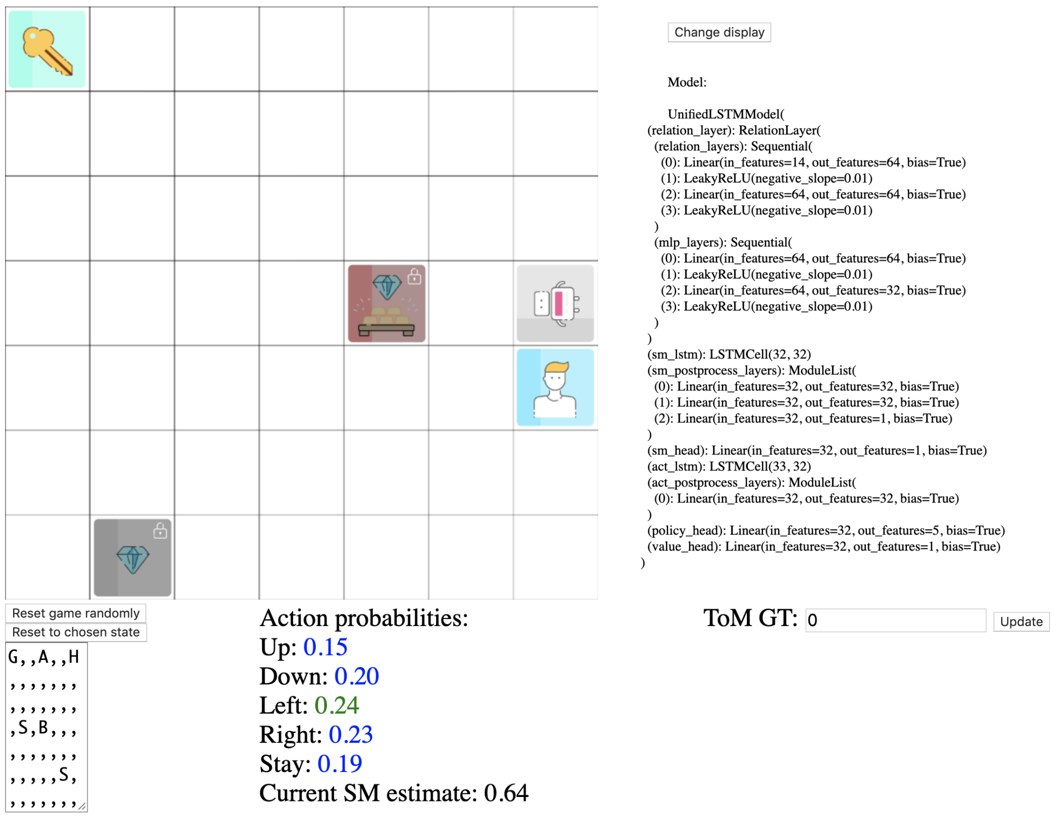
To actually use these agents, I created a Collector class holding two Agent instances and the environment in question. Its main role is providing the collector .collect_data (...) method rolling out a certain number of steps or episodes in the environment, with the chosen agents.

To actually update an agent, data collected in a Collector is used in a PPOptimizer object which holds an Adam optimizer and provides the ppo.train_on_data(...) method, using said optimizer to modify the agents' weights.

Finally, all of this is combined in a Trainer object that takes care of the general training logic: it holds the agents, the environment, a collector and an optimizer, and provides the method trainer . train (...) which iterates a specified number of time, collecting a batch of data and using it to update the agents.

4.6.3 Interactive mode

To simplify experimenting and evaluating various agents, I developed a web app using Flask and TypeScript as a front-end for the environment – the user can run a local web server with a specified environment and model to be used as one of the agents, and then control the other agent with their keyboard. A picture of the interface can be seen in Figure 8.



```
Model:

UnifiedLSTMModel(
(relation_layer): RelationLayer(
  (relation_layers): Sequential(
    (0): Linear(in_features=14, out_features=64, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=64, out_features=64, bias=True)
    (3): LeakyReLU(negative_slope=0.01)
  )
  (mlp_layers): Sequential(
    (0): Linear(in_features=64, out_features=64, bias=True)
    (1): LeakyReLU(negative_slope=0.01)
    (2): Linear(in_features=64, out_features=32, bias=True)
    (3): LeakyReLU(negative_slope=0.01)
  )
)
(sm_lstm): LSTMCell(32, 32)
(sm_postprocess_layers): ModuleList(
  (0): Linear(in_features=32, out_features=32, bias=True)
  (1): Linear(in_features=32, out_features=32, bias=True)
  (2): Linear(in_features=32, out_features=1, bias=True)
)
(sm_head): Linear(in_features=32, out_features=1, bias=True)
(act_lstm): LSTMCell(33, 32)
(act_postprocess_layers): ModuleList(
  (0): Linear(in_features=32, out_features=32, bias=True)
)
(policy_head): Linear(in_features=32, out_features=5, bias=True)
(value_head): Linear(in_features=32, out_features=1, bias=True)
)
```

```
G,,A,,H
,,,,,,,
,,,,,,,
,S,B,,,
,,,,,,,
,,,,,S,
,,,,,,,
```

```
Action probabilities:
Up: 0.15
Down: 0.20
Left: 0.24
Right: 0.23
Stay: 0.19
Current SM estimate: 0.64
```

Figure 8: The interactive mode interface. Its main features include: the current environment state visualization (top left), a printout of the currently loaded model (top right), a possibility to reset the environment to a specific state (bottom left), the current action probabilities of the RL agent along with its skill level estimate (bottom) and the option to give the agent a specific skill level estimate value (bottom right). The human agent is controlled with direction keys by the user, while the robot agent is controlled by the trained RL model.

5 Results

This section will describe the experimental results obtained during the course of the project. The core metrics for each experiment are the mean episode length and mean total episode reward. While correlated, the episode length provides a measure independent of the reward structure, allowing for reward shaping to improve the training performance.

It will begin with the two goal experiment, as it was designed to be the simplest one serving as a test of the basic principles. Then, progressively more difficult, the final action experiment and base TSP environment results are described.

5.1 Two goals environment

In this setting, I have trained recurrent models in standard, GT and SM modes with binary partner choice and skill representation. As can be seen in Figure 9, all models manage to achieve a decent performance, with the GT model achieving the highest final mean reward, followed by SM and then the baseline model.

A similar picture can be seen in the graph of correct goal collection ratio, defined as the number of episodes with a correctly chosen final goal (according to the partner agent) divided by the total number of episodes taking place in that iteration.

An important thing to keep in mind is that the way a single training batch is collected, every batch will contain an approximately constant number of environment steps. Since in later stages of the training, a single episode with an expert partner is likely to be shorter than an episode with a novice partners, this skews the per-episode metrics in the direction of expert values.

To counteract this effect, Table 2 shows the final results separated by the partner agent type, which show the advantage of GT and SM models in both pairings. The effect is the most visible in the correct goal ratio, which was designed explicitly to require modeling of the partner agent. In the GT variant, the ratio is noticeably lower with a novice partner than with an expert partner, which happens due to the partner agent also sometimes stepping on a final goal while taking random actions, which is impossible for the main agent to prevent.

The skill modeling approach shows moderate improvements in all metrics, particularly the correct goal ratio, which shows that this method can indeed be used to encourage explicit modeling of other agents.

The approximate optimal values in this table are computed using the planning algorithm described in Section 4.5. With an expert partner, the value is directly taken from the results of a simulation with two agents. However, with a novice partner, it's not obvious how exactly to include its randomness, so the choice taken here was to model it as an immobile agent, and the optimal values in the table are the results of a simulation with only a single agent in the environment. This means the values are biased, but the effect is small enough that the planner still provides valuable insights regarding the best possible performance.

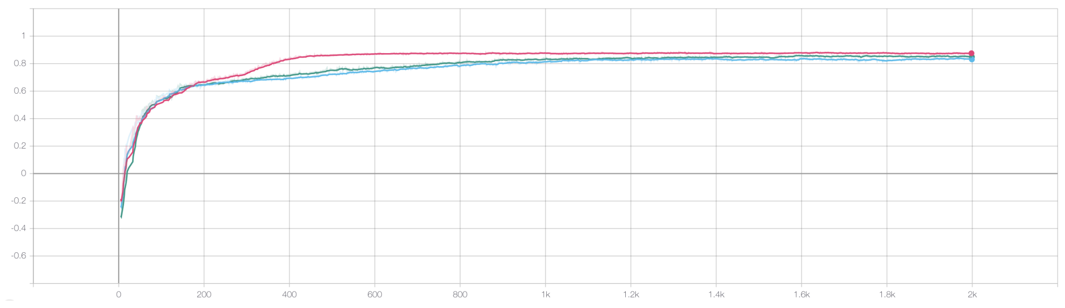
	Partner type	Mean episode reward	Mean episode length	Correct goal ratio
Baseline	Novice	0.80 ± 0.004	15.25 ± 0.18	0.88 ± 0.010
	Expert	0.86 ± 0.004	9.92 ± 0.13	0.89 ± 0.009
Skill Modeling	Novice	0.82 ± 0.004	13.37 ± 0.15	0.88 ± 0.010
	Expert	0.88 ± 0.003	9.82 ± 0.13	0.94 ± 0.007
Ground Truth	Novice	0.84 ± 0.003	13.04 ± 0.16	0.93 ± 0.008
	Expert	0.90 ± 0.002	9.38 ± 0.12	0.99 ± 0.004
Optimal	Novice	0.88	12.52	1
	Expert	0.94	6.14	1

Table 2: Evaluation of fully trained agents in the two goal setting with two subgoals computed on a sample of 1000 episodes. Confidence intervals for reward and length values correspond to the standard error of the mean, and for the correct goal ratio, it is the 95% Wald confidence interval of the binomial distribution.

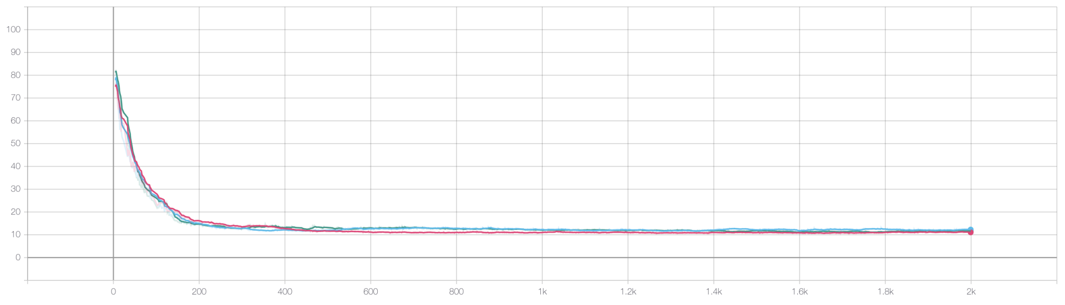
5.1.1 Emergent cooperative behavior

An interesting phenomenon could be observed when a memoryless (i.e. the LSTM was replaced with an identity layer) agent was used instead of the typical stateful one in the two goals experiment, with binary sampling method. As can be seen in Figure 10, the correct goal selection ratio approaches relatively high values, even without GT information, reaching even 80% accuracy. At first this seems like it shouldn't be possible – after all, the agents do not have any memory, so they shouldn't be able to create any internal skill representation of their partners.

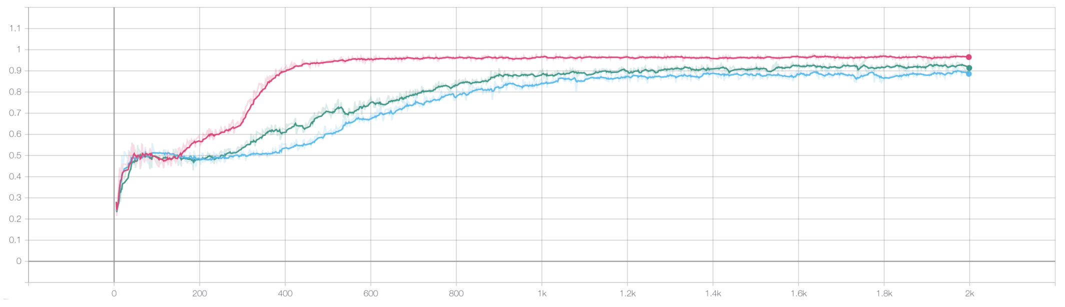
A hint as to what's happening can be found in Figure 10b which shows how the mean episode length changes throughout the training. While at the



(a) Mean episode reward



(b) Mean episode length



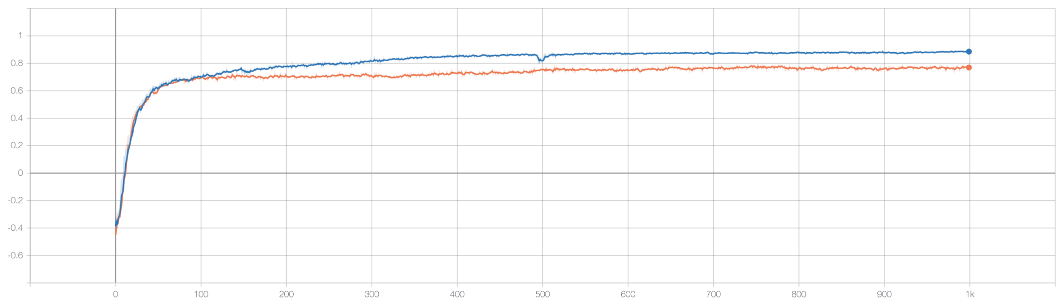
(c) Correct goal ratio

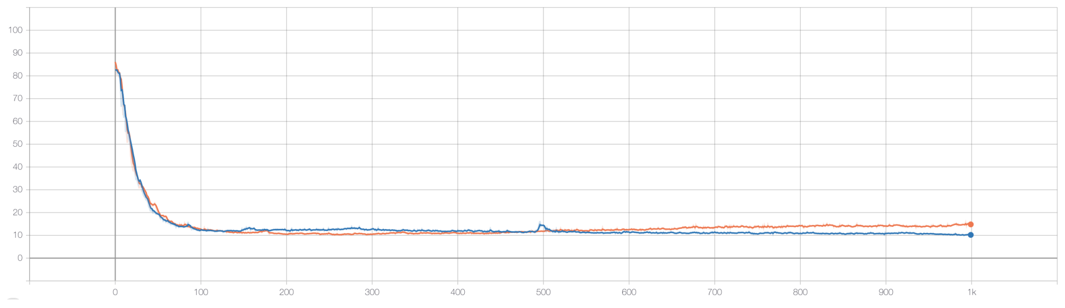
Figure 9: Training curves of the baseline (blue), SM (green) and GT (pink) models throughout the training in the two goal environment.

beginning it is steadily decreasing, there's a clear increase towards the end when the correct goal ratio in Figure 10c also increases. This suggests that through taking some extra steps, the agents manage to develop a way to coordinate so that the obtained reward becomes higher (Figure 10a) through the higher probability of choosing the correct final goal.

While interpreting the decisions of a neural network-based agents is not easy, this hypothesis is readily confirmed by investigating visualizations of episode rollouts, and interacting with a trained agent in the interactive mode. It turns out that if two fully trained versions of the model are interacting with each other, they will move in a very close distance to each other, often occupying the same grid cell, and will walk towards the correct goal together. However, if the agents are far away from one another, they will instead proceed to the goal corresponding to the novice partner.

In the interactive mode, it is possible to investigate this phenomenon and make the other agent switch between heading towards either goal by repeatedly increasing and decreasing the distance between agents.


(a) Mean episode reward


(b) Mean episode length

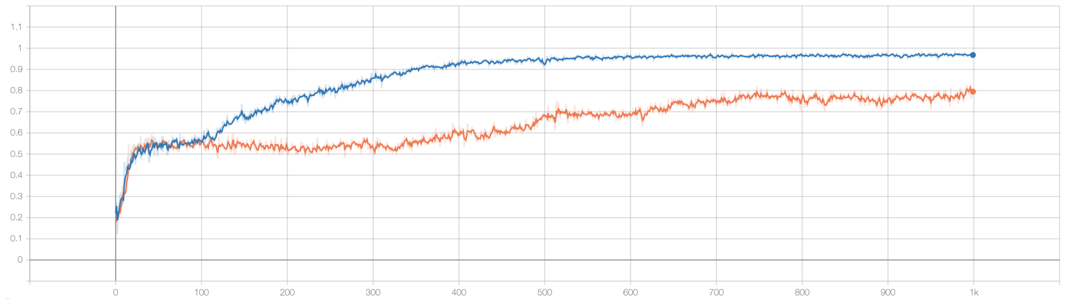

(c) Correct goal ratio

Figure 10: Training curves of the stateless baseline (orange) and GT (blue) models throughout the training in the two goal environment, showing the emergent memoryless cooperation capabilities of the agents.

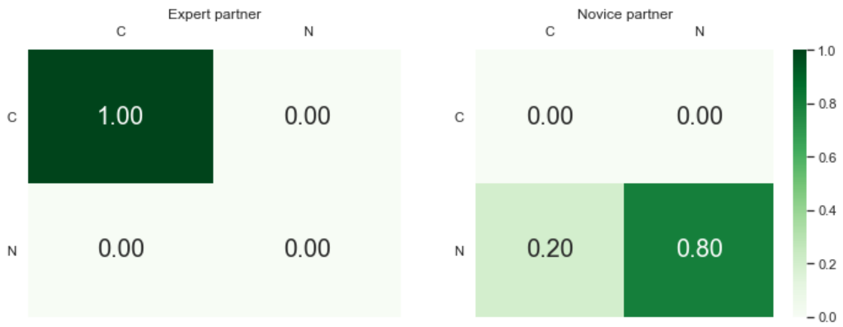
5.2 Final Action Environment

In this environment, there is a natural measure of cooperation in the Assurance Game mechanism which is the ratio of episodes ended with each joint action. When evaluated separately with an expert and novice partner, the optimal expected behavior is cooperating with an expert producing only (C, C) joint actions, and not cooperating with a novice producing a mixture of 80% (N, N) and 20% (N, C). The uncertainty here is unavoidable since the novice agent behaves randomly and sometimes will cooperate as if by accident.

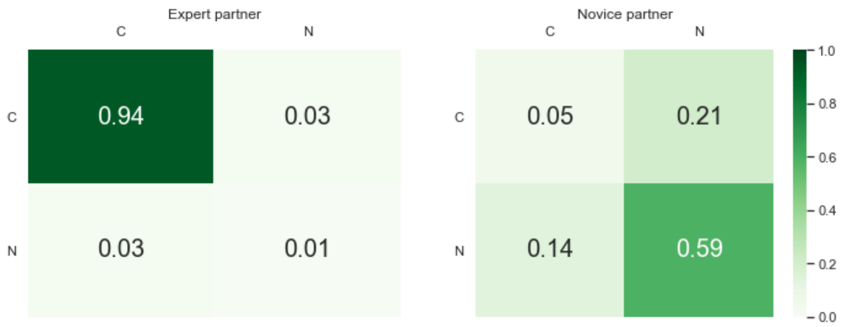
A good way of visually depicting this measure is by using a frequency matrix of joint actions. This can be seen in Figure 11, including the theoretical optimum and real experimental results of different models in a binary setting, where only the fully trained agent and the randomly initialized one are considered.

The most important values for comparing different models are the diagonal values: (C, C) for expert partners, and (N, N) for novice partners, as those are the respective optimal choices for expert agents. While all models perform fairly well with expert partners, the baseline agent is noticeably underperforming when partnered with a novice partner. Skill Modeling agents attain a performance close to the Ground Truth agents despite receiving no extra information in the evaluation phase, which shows the strength of this approach.

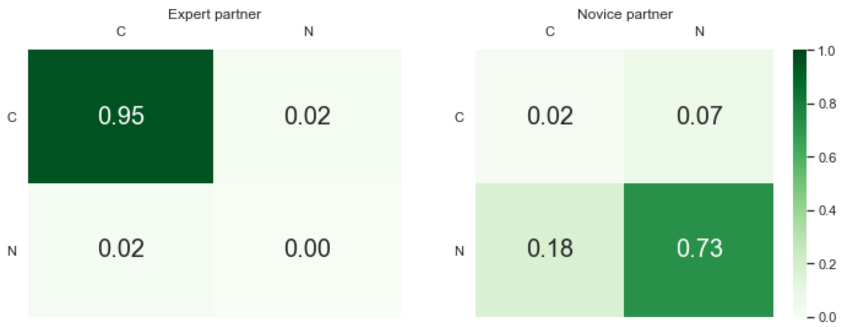
Due to the inherent stochasticity of the models, even expert agents with expert partners and perfect information about their identity, sometimes make mistakes. The largest difference between the models is in the (C, N) frequency of the baseline model. It means that it relatively often tries to cooperate with a novice partner, as if due to incorrectly predicting its skill level. The corresponding values for GT and SM models are significantly lower.



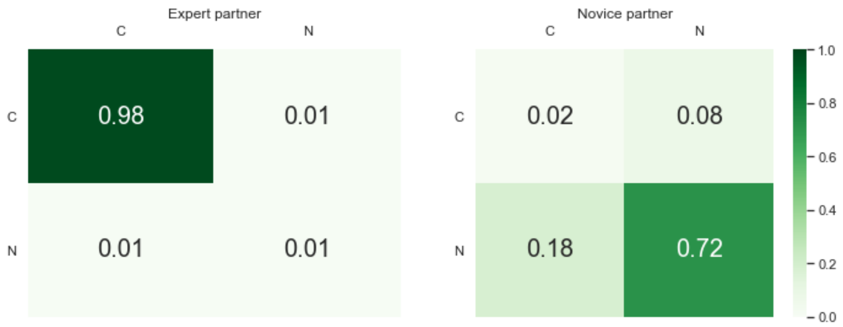
(a) Optimal agent with ideal information



(b) Baseline agent



(c) Skill Modeling agent



(d) Ground Truth agent

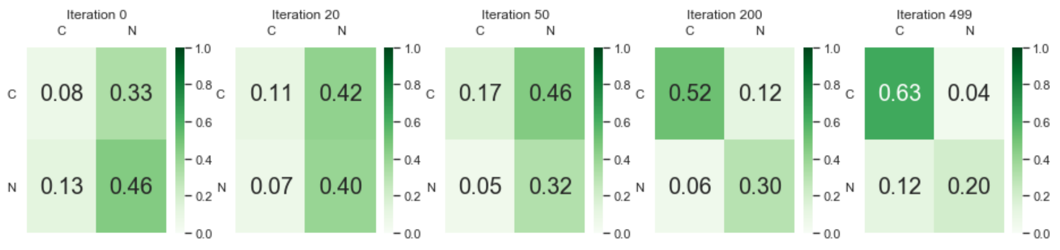
Figure 11: Frequency matrices of each joint action with different model architectures in the final action experiment and for the optimal agent. The partner agent corresponds to the column player of the matrix.

5.2.1 Continuous skill level

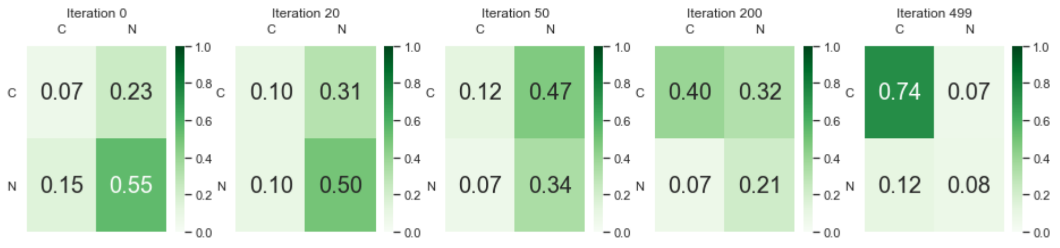
For more robust results, another evaluation was performed – instead of only using the binary skill levels, the continuous values normalized to the range $[0, 1]$ as described in Section 4.4. For easier interpretability of the results, the final agent was evaluated with a set of agents at chosen iteration numbers (0, 20, 50, 200, 499) from an independently trained population, with the values set to represent the entire range of possible skill levels. A visualization of those results in the form of final action frequency matrices can be seen in Figure 12.

The improvement in this case is not quite as apparent as in the binary skill representation. The Ground Truth model still achieves the best performance, especially with very unskilled and very skilled partners. The performance of the Skill Modeling agent is in some cases higher than the baseline, and sometimes lower, making it impossible to draw clear conclusions.

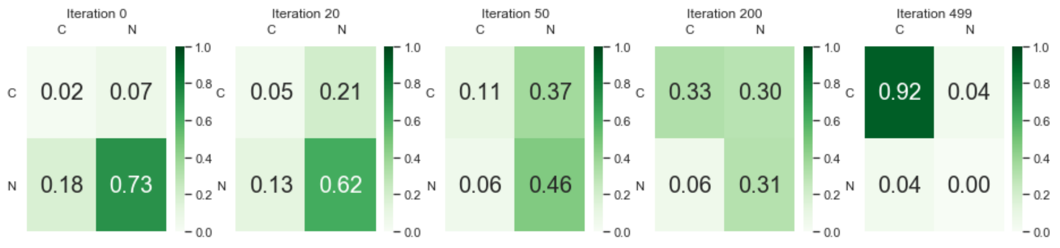
It is also apparent that the most difficult part of this task is, in fact, cooperating with intermediate agents since for each model, those have the highest off-diagonal values in the frequency matrices. In a way, those are the least predictable – early agents can be reliably expected to uniformly sample their actions, and advanced agents can generally be trusted to behave optimally, but there are not any such guarantees for intermediate ones.



(a) Baseline agent



(b) Skill Modeling agent

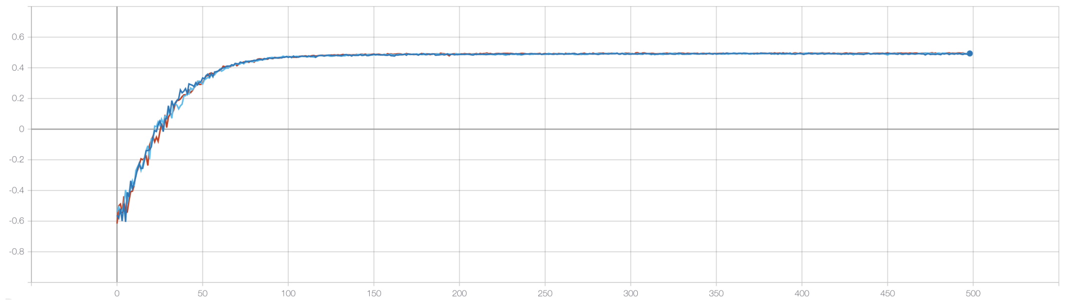


(c) Ground Truth agent

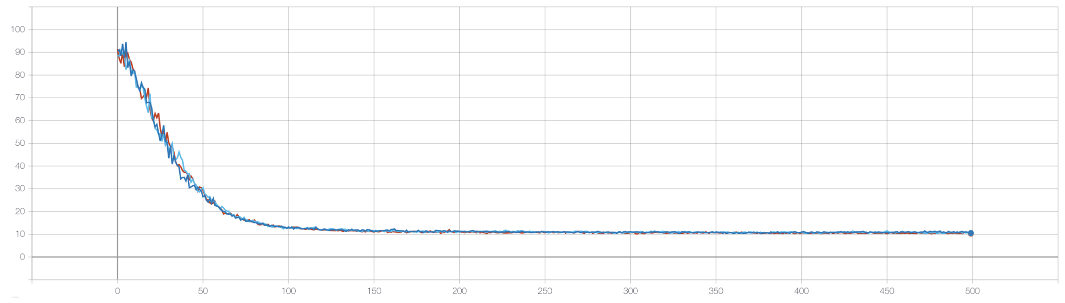
Figure 12: Frequency matrices of each joint action with different model architectures in the final action experiment in the continuous skill representation case. The main agent is fully trained and corresponds to the row player of the matrix, while the partner agent is taken from the denoted iteration and corresponds to the column player.

5.3 Base TSP environment

In this setting, there is no explicit metric that focuses specifically on the cooperation. However, since the ability to effectively cooperate should improve the agents' overall performance, the standard metrics of mean episode length and total reward can be used for the comparison. Those quantities, plotted against training iterations, can be seen in Figure 13.



(a) Mean episode reward



(b) Mean episode length

Figure 13: Training curves of the baseline (light blue), SM (red) and GT (dark blue) models throughout the training in the base environment with two subgoals.

The three lines overlap almost entirely, showing that neither Ground Truth nor Skill Modeling approaches bring any major improvements. This is somewhat surprising, as the underlying assumption is that the ground truth information should be useful for the agent's decision-making, suggesting that this environment might not be the best choice for this specific phenomenon. It is also possible that a more extensive hyperparameter search, together with larger neural networks, would yield better results, but this is left for future work.

More detailed insights can be obtained by evaluating the agent with a specific type of partner (i.e. expert or novice), and investigating the episode lengths and total returns in both cases. This can be seen in Table 3. Even then, the results are quite close for all the models, which indicates that the extra

	Partner type	Mean episode return	Mean episode length
Simple	Novice	0.470 ± 0.001	12.96 ± 0.139
	Expert	0.506 ± 0.001	9.37 ± 0.109
Skill Modeling	Novice	0.469 ± 0.001	13.09 ± 0.143
	Expert	0.511 ± 0.001	8.90 ± 0.098
Ground Truth	Novice	0.471 ± 0.002	12.90 ± 0.155
	Expert	0.508 ± 0.001	9.20 ± 0.108
Optimal	Novice	0.475	12.45
	Expert	0.538	6.13

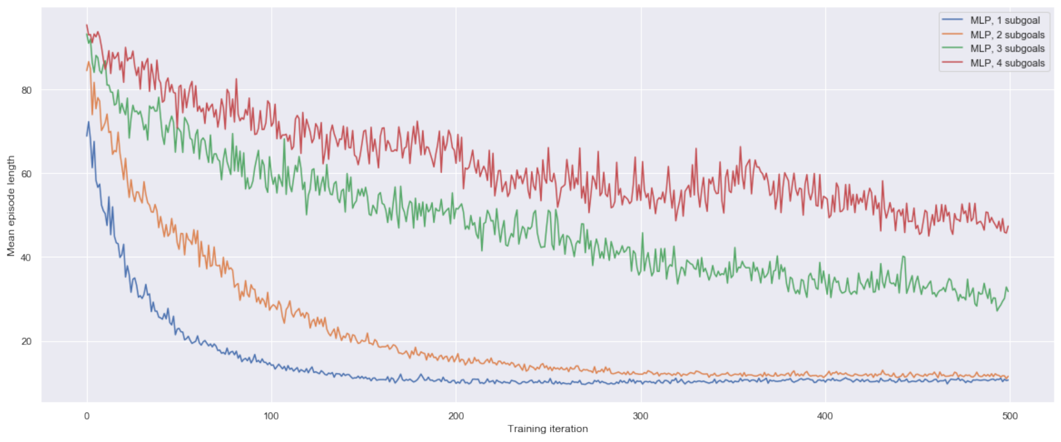
Table 3: Evaluation of fully trained agents in the base setting. Confidence intervals for reward and length values correspond to the standard error of the mean. The optimal values were obtained using the planning algorithm, with Novice values computed under the assumption that the partner doesn't collect any subgoals.

information is not really being used by the agents, or that it doesn't actually bring any significant benefit.

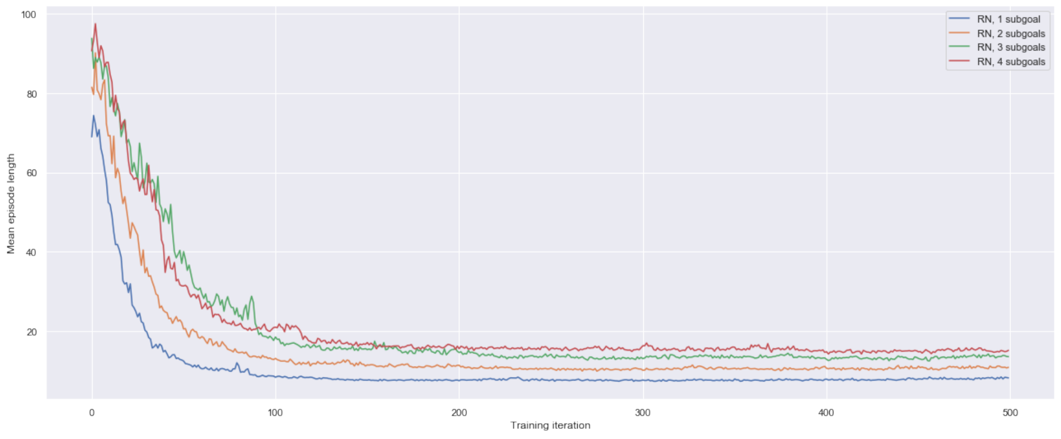
In analyzing this behavior, it can be observed that the agent's performances with a novice partner are generally fairly close to the optimum, but are not that great when matched up with an expert partner. To some extent, this can be explained by the fact that it can be extremely risky to assume a partner is skilled. In an extreme case, it could lead the main agent to wait near the final goal, hoping their partner would gather the remaining subgoals, to no avail. On the other hand, the penalty for not cooperating with a skilled partner is not that large, causing this to be a “safe" option that hinders the overall performance.

5.4 Efficiency impact of the Relational Network

An interesting thing to explore is the effect of the Relational Network-based embedding of the input observation, as opposed to a regular dense representation which is then processed by an MLP, in the standard TSP environment. Since the advantage of RNs comes into play when there's multiple objects of the same type, the basis of the comparison is changing the number of subgoals in the environment and analysing the dynamics of the training process.



(a) Mean episode length with an MLP model



(b) Mean episode length with an RN model

Figure 14: Training curves of MLP and RN models, with a varying number of subgoals.

In Figure 14 you can see the results of training a model with and without Relational layers, with the number of subgoals ranging from 1 to 4. As expected, RN models scale pretty well, experiencing only small performance decrease due to the more difficult task, whereas MLP models with 3 and 4 subgoals require significantly more data.

The reason for this is that RN models inherently encode the fact that all subgoals are identical except for their positions, whereas MLP models have to learn that through trial and error. For example, once the model learns that collecting one subgoal gives it a reward, an RN model automatically knows this for all subgoals, but the MLP has to learn it again for each additional goal, which requires additional training data.

6 Summary

The core question of this work is whether including an explicit skill modeling component in policy-based Reinforcement Learning can improve their capability to cooperate with various agents across different skill levels. In order to evaluate this hypothesis, I built three multiagent environments and in each of them, trained models with different levels of access to their partner's skill levels.

The experimental results suggest that while there is some promise in this approach, it likely needs to be revised to support environments with more subtle cooperation mechanisms, as well as be more robust to wide ranges of possible skill levels that could occur in those environments.

In the two environments designed to explicitly reward cooperation, the skill modeling approach worked very well in improving the agents' interactions when only two skill levels were considered. The agents were capable of correctly estimating their partner's identity and using that information to choose the right actions in order to maximize their joint reward.

These experiments also showed that intuitions about cooperation capabilities of various agents in certain environments can be quite misleading – in an environment where it seems necessary to have an internal model of the partner's skill level, even a memoryless model managed to reach a seemingly impressive result of 80% accuracy by developing a distance-based heuristic, in a way storing information in the agents' relative positions.

However, when including a wider range of partner agents, the performance improvement greatly diminished. In this case, a difficult issue has to be faced in representing the skill level – for an agent to act with a copy of itself, it needs to know its own skill level, which is not possible before actually taking some actions. What's more, if an agent acts with a partner that's more skilled than itself, it would receive a skill level input greater than anything in its training data leading to difficulties in generalization. The normalization-based method used in this work exhibits some convenient theoretical properties, but is by no means the only possible choice.

Finally, the experiment in the basic Traveling Salesman Problem environment without extra mechanisms also doesn't show major improvements over the baseline. To some extent, this is likely due to the specific task that the agents have to fulfill – while it is intuitive that perfect coordination would yield better results than a total lack of cooperation, the degree to which this impact would be noticeable is unclear. What's more, given how short the episodes are, taking even a few exploratory or coordination-oriented actions can significantly change the performance.

6.1 Future directions

There are various directions which could yield more robust results by using the architecture and the ideas described in this work. They can be grouped in three main categories:

- Modifications of the core algorithm
- Modifications of the peripheral elements
- More robust evaluation environments

Regarding the core algorithm, the briefly explored training procedure where the skill modeling component is trained separately from the agents themselves showed some promising results, which likely could lead to a better performance and generalization across various agents, avoiding the non-stationarity of multiagent training.

The peripheral elements primarily refer to the issue of skill representation and normalization – in order to use an agent's skill level for decision-making, we must first define what a skill level is, and then represent it in the form of a number or a vector, and this turns out to be nontrivial, especially given additional constraints on the expected behaviour of those values. It could also involve changing the exact self-play training procedure, e.g. by using the previous version as a partner for self-play rather than a copy of the newest agent, or introducing some kind of a schedule to the training.

Finally, using environments that put more emphasis on ad-hoc cooperation could exhibit stronger results even with the current approach. Notable examples include the Overcooked environment [55] introduced to accentuate the mechanisms of human-AI cooperation, and the game of Hanabi [1] which serves as an important benchmark in partially observable cooperative multiagent problems. What's more, pursuing the skill-modeling angle further, environments with mechanically difficult components could also be suitable, like the game of StarCraft 2 [56].

In the end, this work has shown that the proposed idea displays promising results in facilitating ad-hoc cooperation between RL agents, and with additional experiments could be refined to a versatile MARL algorithm.

References

[1] Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The Hanabi Challenge: A New Frontier for AI Research. *Artificial Intelligence*, 280:103216, March 2020.

[2] Wilko Schwarting, Alyssa Pierson, Javier Alonso-Mora, Sertac Karaman, and Daniela Rus. Social behavior for autonomous vehicles. *Proceedings of the National Academy of Sciences*, 116(50):24972–24978, December 2019.

[3] David Premack and Guy Woodruff. Does the chimpanzee have a theory of mind? *Behavioral and Brain Sciences*, 1(4):515–526, 1978.

[4] Neil C. Rabinowitz, Frank Perbet, H. Francis Song, Chiyuan Zhang, S. M. Ali Eslami, and Matthew Botvinick. Machine Theory of Mind. *arXiv:1802.07740 [cs]*, March 2018.

[5] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[6] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *arXiv:1712.01815 [cs]*, December 2017.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*, December 2013.

[8] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint*, 2019.

[9] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent Reinforcement Learning: An Overview. In Dipti Srinivasan and Lakhmi C. Jain, editors, *Innovations in Multi-Agent Systems and Applications - 1*, Studies in Computational Intelligence, pages 183–221. Springer, Berlin, Heidelberg, 2010.

[10] Georgios Papoudakis, Filippos Christianos, Arrasy Rahman, and Stefano V. Albrecht. Dealing with Non-Stationarity in Multi-Agent Deep Reinforcement Learning. *arXiv:1906.04737 [cs, stat]*, June 2019.

[11] Sanyam Kapoor. Multi-Agent Reinforcement Learning: A Report on Challenges and Approaches. *arXiv:1807.09427 [cs, stat]*, July 2018.

[12] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer Series in Statistics. Springer-Verlag, New York, 2 edition, 2009.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[15] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

[16] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27–48, April 2016.

[17] Balázs Csanád Csáji. Approximation with Artificial Neural Networks, 2001. Library Catalog: www.semanticscholar.org.

[18] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. *arXiv:1703.03400 [cs]*, July 2017.

[19] Joel Z. Leibo, Edward Hughes, Marc Lanctot, and Thore Graepel. Autocurricula and the Emergence of Innovation from Social Interaction: A Manifesto for Multi-Agent Intelligence Research. *arXiv:1903.00742 [cs, q-bio]*, March 2019.

[20] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete Problems in AI Safety. *arXiv:1606.06565 [cs]*, July 2016.

[21] Louis Augustin Cauchy. Methode générale pour la résolution des systemes d'équations simultanées. *Comptes Rendus Hebd. Seances Acad. Sci.*, 25:536–538, October 1847.

[22] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, October 1986.

[23] P.J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, October 1990. Conference Name: Proceedings of the IEEE.

[24] Guillaume Chevalier. LARNN: Linear Attention Recurrent Neural Network. *arXiv:1808.05578 [cs, stat]*, August 2018.

[25] Adam Santoro, David Raposo, David G. T. Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. A simple neural network module for relational reasoning. *arXiv:1706.01427 [cs]*, June 2017.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, November 2018. Google-Books-ID: 6DKPtQEACAAJ.

[27] Karl Johan Åström. Optimal Control of Markov Processes with Incomplete State Information I. *Journal of Mathematical Analysis and Applications*, 10:174–205, 1965. Publisher: Elsevier.

[28] Daniel S. Bernstein, Shlomo Zilberstein, and Neil Immerman. The Complexity of Decentralized Control of Markov Decision Processes. *arXiv:1301.3836 [cs]*, January 2013.

[29] Richard Bellman. On the Theory of Dynamic Programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716–719, August 1952.

[30] Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. AI Safety Gridworlds. *arXiv:1711.09883 [cs]*, November 2017.

[31] Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In S. A. Solla, T. K. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems 12*, pages 1057–1063. MIT Press, 2000.

[32] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, August 2017.

[33] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937, June 2016. ISSN: 1938-7228 Section: Machine Learning.

[34] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-Dimensional Continuous Control Using Generalized Advantage Estimation. *arXiv:1506.02438 [cs]*, October 2018.

[35] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep Reinforcement Learning for Multi-Agent Systems: A Review of Challenges, Solutions and Applications. *IEEE Transactions on Cybernetics*, pages 1–14, 2020.

[36] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. *arXiv:1706.02275 [cs]*, March 2020.

[37] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, July 2019.

[38] Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic Motivation and Automatic Curricula via Asymmetric Self-Play. *arXiv:1703.05407 [cs]*, April 2018.

[39] Aditya Grover, Maruan Al-Shedivat, Jayesh K. Gupta, Yura Burda, and Harrison Edwards. Learning Policy Representations in Multiagent Systems. *arXiv:1806.06464 [cs, stat]*, July 2018.

[40] Jurgen Schmidhuber. *Evolutionary Principles in Self-Referential Learning. On Learning now to Learn: The Meta-Meta-Meta...-Hook.* Diploma Thesis, Technische Universitat Munchen, Germany, 1987.

[41] Evan Hubinger, Chris van Merwijk, Vladimir Mikulik, Joar Skalse, and Scott Garrabrant. Risks from Learned Optimization in Advanced Machine Learning Systems. *arXiv:1906.01820 [cs]*, June 2019.

[42] Henry M. Wellman and David Liu. Scaling of Theory-of-Mind Tasks. *Child Development*, 75(2):523–541, March 2004.

[43] S. Baron-Cohen, A. M. Leslie, and U. Frith. Does the autistic child have a "theory of mind"? *Cognition*, 21(1):37–46, October 1985.

[44] Jakob N. Foerster, Francis Song, Edward Hughes, Neil Burch, Iain Dunning, Shimon Whiteson, Matthew Botvinick, and Michael Bowling. Bayesian Action Decoder for Deep Multi-Agent Reinforcement Learning. *arXiv:1811.01458 [cs]*, September 2019.

[45] Hengyuan Hu and Jakob N. Foerster. Simplified Action Decoder for Deep Multi-Agent Reinforcement Learning. *arXiv:1912.02288 [cs]*, December 2019.

[46] Michael Taylor. *The Possibility of Cooperation.* Cambridge University Press, August 1987. Google-Books-ID: ioGuQgAACAAJ.

[47] Alexander Schrijver. On the History of Combinatorial Optimization (Till 1960). In *Handbooks in Operations Research and Management Science*, volume 12, pages 1–68. Elsevier, 2005.

[48] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009.

[49] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[51] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015.

[52] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. Publisher: IEEE COMPUTER SOC.

[53] DeepMind. pycolab, 2017.

[54] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016.

[55] Micah Carroll, Rohin Shah, Mark K. Ho, Thomas L. Griffiths, Sanjit A. Seshia, Pieter Abbeel, and Anca Dragan. On the Utility of Learning about Humans for Human-AI Coordination. *arXiv:1910.05789 [cs, stat]*, October 2019.

[56] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv:1708.04782 [cs]*, August 2017.

[57] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, and Skye Wanderman-Milne. JAX: composable transformations of Python+NumPy programs, 2018.

[58] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

A Backpropagation Derivation

Backpropagation is the fundamental algorithm making it possible to train deep neural networks in an end-to-end manner. [22] Consider a network consisting of L layers, defined by the equation:

$$\begin{aligned} z^l &= w^l a^{l-1} + b^l \\ a^l &= \sigma(z^l) \end{aligned} \tag{A1}$$

where w^l, b^l are the weights and biases at layer l, z^l is the weighted input to layer l, and a^l are the activations. Note that z^l, a^l, b^l are vectors and w^l are matrices of appropriate dimensions.

Let's also decide that we're optimizing a quadratic loss function defined as follows:

$$C = \frac{1}{2n} \sum_x ||y - a^L(x)|| \tag{A2}$$

It could easily be replaced with any differentiable loss function, but this one is used for simplicity of the proof.

First, let's define the error of a neuron j in layer l as

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{A3}$$

This corresponds to the intuition that we're seeking the derivative with respect to the weights, and serves as a stepping stone to that goal.

The main idea of backpropagation, as suggested by the name, is that given the gradients at a certain layer, we can compute the gradients of the previous one, in a way sending the signal backwards. Therefore as a first step we need to find the error term of the last layer:

$$\begin{aligned} \delta_j^L &= \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \\ \delta^L &= \nabla_a C \odot \sigma'(z^L) \end{aligned} \tag{A4}$$

where $\odot$ is the Hadamard (component-wise) product, and the second equation is the matrix-based form of the previous, component-based equation.

Now, with this starting condition, we can build a recurrence relation using the chain rule of differentiation. Since we know how a^l depends on a^{l-1}, we can use this to obtain the following expression:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma(z^l) \tag{A5}$$

The only remaining step is actually computing the gradients with respect to the weights, using the error terms δ^l. For the bias vectors, this turns out to be quite simple:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{A6}$$

And for the weight matrices, it also follows simply Equation A1:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{A7}$$

Now all that's needed is putting all these components together. The entire process is as follows:

1. Compute all weighted inputs and activations z^l, a^l (Equation A1)
2. Compute the loss function C (Equation A2)
3. Compute the last error term δ^L (Equation A4)
4. Recursively compute the remaining error terms δ^l (Equation A5)
5. Compute the bias and weight gradients (Equations A6 and A7)

This procedure is implemented in a highly-efficient manner using the vectorized formulation in various autograd libraries such as PyTorch [50], TensorFlow [51] or Jax [57] and as such can be used in training the neural networks used in this work.

B Policy Gradient Theorem Derivation

Policy Gradient-based algorithms such as PPO work by directly optimizing the reward function by changing the policy's weights with gradient descent [58]. Formally speaking, we're trying to find a policy $\pi_\theta \colon S \to \Delta A$ that maximizes the expected reward obtained over the course of an episode:

$$\theta^* = \arg\max_\theta J(\pi_\theta) = \arg\max_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}} [R(\tau)] \tag{B1}$$

A necessary building block for this is an estimate of the gradient $\nabla_\theta J(\pi_\theta)$. In order to obtain it, let's explicitly write out the expectation from Equation B1:

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \underset{\tau\sim\pi_\theta}{\mathbb{E}} [R(\tau)] = \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) = \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) \tag{B2}$$

To take this further, something called a log derivative trick is necessary: for any differentiable function $f \colon \mathbb{R}^n \to \mathbb{R}$ it is true that:

$$\nabla_x f(x) = f(x) \nabla_x \log f(x) \tag{B3}$$

Indeed, it can be verified as follows:

$$RHS = f(x)\nabla_x \log f(x) = f(x)\frac{1}{f(x)}\nabla_x f(x) = \nabla_x f(X) = LHS \tag{B4}$$

With this, setting $f \colon \theta \mapsto P(\tau|\theta)$ we can continue the derivation:

$$\begin{gathered}\nabla_\theta J(\pi_\theta) = \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) = \\ \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) = \\ \underset{(s_t,a_t)\sim\pi_\theta}{\mathbb{E}} [\nabla_\theta P(\tau|\theta) R(s_t)]\end{gathered} \tag{B5}$$

Because a trajectory is just a sequence of states and actions, this can be expanded into the final form:

$$\nabla_\theta J(\pi_\theta) = \underset{\tau\sim\pi_\theta}{\mathbb{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta (a_t|s_t) R(\tau) \right] \tag{B6}$$

□