From Single-Agent Reinforcement Learning to Multi-Agent Communication

Sophie Haas
Institute of Cognitive Science
Osnabrück, Germany
sohaas@uni-osnabrueck.de

Johanna Linkemeyer
Institute of Cognitive Science
Osnabrück, Germany
jlinkemeyer@uni-osnabrueck.de

Tjorven Weber
Institute of Cognitive Science
Osnabrück, Germany
tjweber@uni-osnabrueck.de

***Abstract*—In this work, we initially aimed at investigating how verbal communication can develop in a cooperative multi-agent deep reinforcement learning task. The implementations from this work lay the groundwork for this goal, while currently being limited to the single-agent case. The results from this work demonstrate that the task we created to investigate multi-agent communication in future work, is in itself solvable by a single agent. The environment is optimized to be of minimal complexity and we showed that double deep q-networks with epsilon decay is appropriate for solving the task.**

***Index Terms*—reinforcement learning, multi-agent communication, deep learning, ml-agents, Unity**

I. Introduction

For humans, communication and collaboration to solve a problem at hand is an incredibly important aspect of life and is often taken for granted. However, in the field of artificial intelligence, this is a very difficult and long debated problem.

If agents had the ability to communicate with each other or even with humans, this would be a big step not only for cooperative operations, but also for increasing the interpretability of agents' actions [1]. According to [2], the development of agents capable of using language in such a flexible way cannot be solved sufficiently well with supervised learning. Instead, they suggest to use reinforcement learning in a game approach.

In these so-called multi-agent coordination communication games, the agents are placed in a simple environment where they need to interact and coordinate in order to obtain rewards. Thereby, a common language emerges out of necessity.

In this project, the focus is on a two-agent object manipulation game, consisting of a sender and a receiver. The task is to move objects of different colors to their respective target position in a limited area. The sender is the instructing agent, which is is provided with information about a goal state and has to tell the receiver where to move each item accordingly. The receiver is the executing agent, which has to relocate the objects according to the instruction.

This is realized using the Unity Machine Learning Agents (ML-Agents) plugin, as it offers an excellent visualization of the agents' learning process and a convenient way to implement the 3D environment. Furthermore, the toolkit enables parallel training of multiple agents sharing the same neural network, which can significantly speed up and stabilize the training process [3].

II. Related Work

In order to investigate the emergence of communication, at least two entities are required to co-exist in the same environment. Such frameworks, where multiple agents interact with a common environment, are commonly referred to as multi-agent systems [4, 5]. Multi-agent system are widely spread across different domains and application fields, a few examples being resource allocation, traffic control and robotics. What unites all of them is their inherent complexity due to the presence of multiple actuators manipulating the same system. Because of the high complexity, such tasks are not easily solvable by predetermined algorithms, which require a lot of domain knowledge and are oftentimes forced to rely on heuristics [6]. Additionally, they are not well equipped to deal with a non-static environment, which is unavoidable in the case of multiple agents. The trial-and-error learning of reinforcement learning, however, is very suitable to deal with such high-complexity and dynamic environments [4].

Multi-agent setups are typically categorized as either cooperative, competitive or mixed tasks. In cooperative tasks, all agents are trying to achieve a common goal which is defined through a shared reward function ($\rho_1 = \rho_2$). In competitive tasks on the other hand, the agents are maximising their own incentives at the expense of the other agents' return (for two agents: $\rho_1 = -\rho_2$). Mixed tasks do not constrain the form of the reward function, which allows them to model both cooperative and competitive dynamics [4].

A. Partial Observability

The emergence of communication between agents is typically studied in a cooperative context with partial observability, where each agent, instead of receiving a global observation (the Markov state), only receives local observations that are in correlation with the Markov state. Such a setting is desirable for the study of language, since it requires the agents to communicate in order to achieve their goal, creating the necessary incentive for language to emerge [7].

Partially observable reinforcement learning problems, as described above, can be modelled as *Partially Observable Markov Decision Process* (POMDP) [5]. POMDPs are an extension of *Markov Decision Processes*, which are used to describe classical single-agent reinforcement learning problems with full observability [8]. An MDP is defined through a

(finite) set of states $S = \{s^1, ..., s^N\}$, that the environment can exhibit, a set of possible actions $A = \{a^1, ..., a^K\}$ the agents can take and two functions that describe the environment dynamics. The first function is the probabilistic transition function $T(s, a, s')$, which defines the mapping from the current state s to a new state s', when action a has been taken. The reward function $R(s, a, s')$ on the other hand describes the immediate reward that is received by the agent when the transition occurs. One of the basic assumptions of the MDP is that the observation of the agent encompasses the complete state of the environment. This assumption can be unrealistic in real-world settings and is not applicable to agents operating from an egocentric perspective, which is why the POMDP relaxes this very constraint in order to allow partial observability. A POMDP is therefore defined as the tuple $\langle S, A, \Omega, T, O, R, \rangle$, where S, A, T and R are defined as before described for the MDP. However, instead of observing s' directly as in the case of an MDP, the agent now perceives a partial observation $o \in \Omega$, where $\Omega = \{o^1, ..., o^M\}$ is the set of all observations that are accessible to the agent. What observations are available to the agent is dependent on the global state and defined through the observation function $O : S \times A \times \Omega \rightarrow [0, 1]$ [8].

One of the difficulties with POMDPs is that the relaxation of the full-observability principle also leads to the loss of Markovian state signals and creates a situation where basing the action choices on the observation alone is insufficient to achieve optimal behavior. To evade the problem, agent in a POMDP setting are often equipped with some kind of memory that stores past observations. In its simplest form, such a memory can be created by storing previous observations in sequence with the actions that were taken in-between [8].

B. Deep Q-Networks

Deep Q-learning is one of the techniques that has been applied to partial observability settings, especially when related to the emergence of communication [7, 9].

Deep Q-Networks are based on the Q-learning algorithm and act as a function approximator of the optimal action-value function

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a). \tag{1}$$

The action-value function Q in turn is defined as

$$Q^{\pi}(s, a) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a], \tag{2}$$

which is the expected value over the discounted sum of rewards when taking action a in state s and thereafter following policy π. As Q^* complies with the Bellman Optimality Equation

$$\begin{aligned} &Q^*(s, a) = \\ &E[R_{t+1} + \gamma \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a], \end{aligned} \tag{3}$$

the Q-network approximating Q^* can be updated iteratively with the right-hand side of the equation, as shown here:

$$\begin{aligned} &Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \\ &\alpha_t[r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a') - Q_t(s_t, a_t)]. \end{aligned} \tag{4}$$

The updates are calculated based on trajectories which are created through the interaction of the agent with the environment that is attempted to be solved. To choose an action, the agent applies an ϵ-greedy policy, which selects the currently highest valued action with a probability of $1 - \epsilon$ and performs a random action otherwise [4, 7, 10].

Once a transition has been generated, it is stored in a buffer from which the samples for the Q-network update are drawn. This method is known as Experience Replay and has been introduced for increased training stability, as the sampling process effectively randomizes over the trajectories and thereby removes update variance that is due to correlations between concurrent states [10, 11].

Besides the correlations within the generated trajectories, [11] also addressed the instability that was due to the recursive definition of the update rule, where the target is calculated by the same function that is being updated. To avoid the lack of stability this formulation introduces, they created a delayed version of the q-network which they used for the calculation of the targets. After a certain number of parameter updates, the weights of the q-network are then copied to the delayed network so that the targets can be calculated based on the updated version of the q-function. Very fittingly, the delayed version of the q-network is called the target network [10].

C. Double Deep Q-Networks

A problem that has not been addressed by [11] is the chronic overestimation that occurs during Q-learning. The overestimation is due to the max operator that is used during the calculation of the targets to both select and evaluate the actions. The update rule (3) can be rewritten such that it contains both steps explicitly, θ denoting the weights of the q-network in that case:

$$\begin{aligned} &Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \\ &\alpha_t[r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_t); \theta_t)]. \end{aligned} \tag{5}$$

This way of calculating the targets becomes a problem when the max operator encounters imprecise approximations of the q-values, as it will always pick the positive noise over the realistic estimation. Since the max operator is applied during two separate steps (selection and evaluation), which rely on the same approximation, the probability of overestimation is very high and creates a self-amplifying dynamic. A suggested solution for this problem is the utilization of two separate q-value approximations, one for the selection of the action and one for the evaluation. This method is known as Double Q-learning and can be applied effectively to Deep Q-learning by using the target network as a second function approximator. The targets can then be calculated based on actions selected

by the main q-network, while the q-values associated with these actions are extracted from the target network. The update rule for Double Deep Q-learning then looks as follows, where θ_t denotes the parameters of the main q-network and θ_t^T the parameters of the target network:

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha_t[r_{t+1} + \gamma Q(s_{t+1}, \arg\max_{a'} Q(s_{t+1}, a'; \theta_t); \theta_t^T)]. \quad (6)$$

This adaption of the original Deep Q-learning algorithm has been shown to effectively reduce the overestimation bias and thereby to improve the overall performance in several scenarios. [12].

D. Training Modifications

Besides advanced modifications as Double Deep Q-learning, there are also smaller adaptations that are commonly used to improve deep reinforcement learning generally, and DQNs specifically.

1) Learning Rate Decay and Batch Size: The learning rate α determines the step size with which the q-values are updated. In the beginning of the training, it makes sense to choose a larger learning rate to make bigger steps. Over time, however, the learning rate should be decayed to stabilize the model output and to converge to an optimal policy.

[13] suggest different methods for annealing the learning rate in order to converge to global optima and avoid local minima. The idea is to start with a rather large learning rate, and gradually decreasing it towards zero over the course of the training process. This improves convergence, as smaller steps are taken near the optimum [13]. To achieve this, one approach they present is decaying the learning rate, e.g. using exponential decay or with the staircase method.

On the other hand, [14] suggest that instead of decaying the learning rate, one should increase the batch size. They show that when gradually increasing the batch size throughout training while keeping the learning rate at a constant, similar improvements in test accuracy can be seen after the same amount of training epochs. However, this approach additionally decreases the number of parameter updates significantly, which can reduce the total model training time. The downside of this is the high computational cost it takes to increase the batch size during training [13].

2) Epsilon Decay: Epsilon is used for choosing an action given the q-values. The larger the epsilon value, the more exploration, meaning that the agent tries to take more new actions. The smaller the epsilon value, the more exploitation, meaning that the agent selects the highest q-value, given a specific state. In the beginning of the training, a rather large epsilon value should be chosen, to avoid convergence to local optima and generally, because it would not be reasonable to exploit an uninformed q-function. Over time, however, epsilon can be decayed to favour actions with high q-values to increase the accumulated reward.

Especially regarding sparse reward settings, a sufficient amount of exploration is critical to efficient and successful learning [15]. This is particularly true for Q-learning which suffers from poor exploration due to the bootstrapped nature of the action-value estimation.

A common approach to improve the exploration-exploitation balance is the introduction of an ϵ-decay schedule.

3) Huber Loss: [11] did not only clip their rewards to limit the range of error derivatives in their implementation of a DQN, but they also clipped the errors. According to them, this helps to improve the algorithm's stability.

Error clipping can be achieved by minimizing the Huber loss [16]. Thus, the Huber loss is less sensitive to outliers and can increase the training stability and efficiency.

E. Multi-agent DQN and Communication

1) Independent Q-learning: Independent Q-learning is one of the simplest extensions of Deep Q-learning to the multi-agent case. Instead of incorporating one agent that is controlled by a deep Q-network, there are now multiple agents controlled by a DQN. Interactions between the agents occur only indirect through the environment, such that both learning and acting are completely decentralized [17]. Due to the non-stationarity of the environment, however, the experience replay that is typically used in deep Q-learning is not an appropriate method anymore, as the policy responsible for creating the trajectory is adapting during the learning process [5]. Even though independent Q-learning with DQNs is therefore not easily solvable, [17] have been able to implement it successfully for a two-agent pong game. They chose the algorithm for its positive characteristics, which beyond its simplicity, include a low computational cost as the agents only need to keep track of their own observations and scalability due to its decentralized nature [5, 17].

2) Reinforced Inter-Agent Learning (RIAL): To extend the independent Deep Q-learning algorithm towards a multi-agent communication setup, RIAL combines the IQL algorithm with Deep Recurrent Q-Networks (DRQN) and additional communication actions. In comparison with the Vanilla DQN, the recurrent version of the network possesses an internal hidden state h^a which allows the summary of past observations into a kind of memory. As already addressed in section A, this is crucial in multi-agent settings with partial observability so that the agent can achieve optimal behavior despite the uncertainty that arises through the local observations. In addition to the environment actions u^a that are subject to all reinforcement learning algorithms, RIAL also includes so called communication actions m^a, which are selected on top on the environment action by each agent. The Q-network $Q^a(o_a^t, m_{t-1}^{a'}, h_{t-1}^a, u^a)$ is therefore not only conditioned on the current observation and action, but also on the previous hidden state of the agent (as part of its recurrent nature) and the messages that have been issued by other agents in the previous

time step. To reduce the action space, the Q-network is split into Q_u and Q_m which in each time step separately select the environment and the communication action with an ϵ-greedy policy. Similar to Independent Q-learning, the training is completely decentralized and the non-stationarity problem is solved by exlcuding experience replay. The RIAL method has been applied successfully to a multi-agent decision making problem known as the Switch Riddle [7]

III. Methods

A. Idea

1) Problem Formulation: In order to study multi-agent language emergence through reinforcement learning, we conceptualized a cooperative two-agent sender-receiver scenario in a simple environment. The environment consisted of two differently colored blocks, one of which needed to be pushed into a specific area marked as goal. A colored area at the wall would serve as instruction of which of the blocks to push, as it would exhibit either the color of the first or the color of the second block. As part of the sender-receiver setup, the instruction would be exclusively perceivable to the sender agent. Instead of taking physical actions in space, the sender agent would be able to take a communication action to relay the information from the instruction. The receiver agent on the other hand would be able to take physical actions, based on its egocentric observation of the environment and the communication issued by the other agent. Together, they would need to accomplish the goal of pushing the correct block, as indicated by the instruction into the goal area.

2) Theoretical Concept: To solve the task, we planned on adapting the RIAL algorithm to work with our sender-receiver scenario. The main difference is that instead of having identical action and observation spaces for the agents, in our case they were different and reflected the respective role of the agent. That means instead of having environment actions u^a and communication actions m^a available to both agents, our sender agent would only take communication actions $a^m = \{m_1, ..., m_N\}$ and our receiver agent only environment actions $a^u = \{u_1, ..., u_M\}$. Besides having different action spaces, our agents would also have different observation spaces. Specifically, while the sender agent would be able to observe the instructions, the receiver agent would not be able to do so. Instead, he would observe the utterances of the sender agent on top of visually-based inputs from the environment. Beyond that, we also decided to modify the type of memory our agents possessed: Instead of using the hidden state of Deep Recurrent Q-Networks, our agents would be controlled by Vanilla DQNs and simulate the memory through stacked observations. The reason for this modification was the simplicity of both the concept and the implementation, as we will explain further in the section about our Unity environment.

3) Approach: We decided to take a step-by-step approach for solving our task, which meant starting with a simplified single agent setup and concurrently building up both algorithmic and environmental complexity. The first step, therefore, was a single agent with the goal of pushing one of two blocks in the goal area. Communication was not involved and the agent would also be controlled by the simplest of deep Q-learning algorithm, the Vanilla DQN. This setup was meant to be the place for testing and exploring the environment setup and creating the code base for our DQN-algorithm. Once this task was solved to a certain degree, we could move on to advancing and exploring the implementation until the training and agent behavior was as stable and efficient as possible. From there, we believed to have a good basis for extending the setup to the multi-agent case with communication included. However, since the exploration of the single-agent case turned out to be very time-consuming, as detailed below, the following methods will only cover the single-agent case.

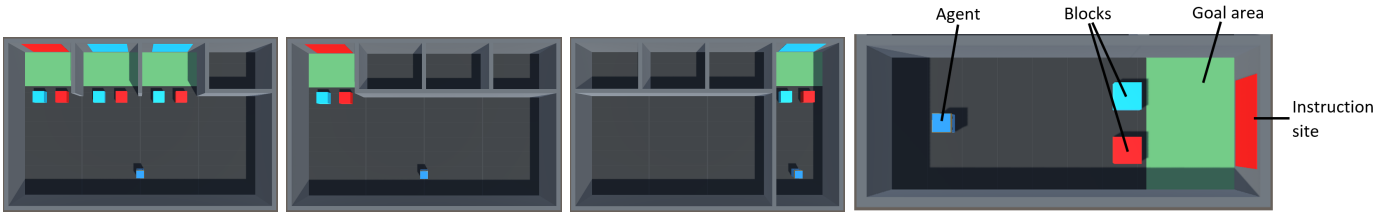
B. Unity Environment

1) Area: We created an area within the Unity scene in which the agent was able to move in. The area was surrounded by walls to give the agent a limited space to move in and thus reduce the project complexity. Apart from the agent, the goal, as well as the two blocks and an instruction site were present within one area.

Building a reinforcement environment in Unity using the ML-Agents toolkit has the advantage that it allows to test the entire setup using existing implementations of proximal policy optimization (PPO) with multiple areas at the same time. In this way, each agent learns from each area, whereas when training outside of Unity, one is limited to a single agent at a time. Using the build-in implementation of PPO, we optimized the area setup to allow us to start training with our own implementation with the most simple environment (Fig. 1d).

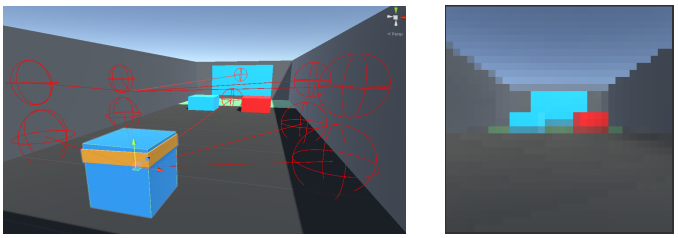
2) Observations: Following the suggestions by the ML-Agents toolkit documentation, we initially decided for raycast observations. The agent was able to perceive its environment by sending out ten rays into its environment (Fig. 2a). The raycast observations were summarized into an observation vector that contained the following information: For each ray, it was determined whether that ray hit a blue or red block, a wall, the goal, or a red or blue instruction site. If it hit one of those objects, the distance from the agent towards the object was stored, else a zero was saved. As a proof of concept, we tested, using the existing PPO implementation, whether the agent was able to learn to push the correct block into the goal area with the raycast observation.

Since training the agent with our own implementation using raycast observations was initially unsuccessful, we reconsidered the observation choice. We tested visual observations, meaning that the agent was able to perceive the environment as if it would see (Fig. 2b). We trained with the existing PPO implementation for 15 minutes and



(a) (b) (c) (d)

Fig. 1: Over time, we decreased the complexity of the task the agent had to fulfil. Our initial idea was to create a task in which an agent has to score at multiple goal areas (a). We realized that training with multiple areas using the PPO implementation from ML-Agents required a very long training time. We therefore decided to reduce the number of goal areas to one (b). With this simplified environment we managed to train the pre-implemented PPO in approximately 20 minutes with nine areas training simultaneously. Considering that we were only able to train with one area outside of Unity, we decided to further simplify the area setup by including another wall such that the agent was allowed to move only within very limited space (c). Subfigure (d) provides a detailed overview of our final area setup.



(a) (b)

Fig. 2: Different observation types: (a) Raycast observations: The agent sends out ten rays to perceive its environment. The observation is encoded in a vector which stores if a ray hits an object, what object type is hit, as well as the distance of the object to the agent. (b) Visual observations: The agent can perceive its environment through a camera that simulates its eyes. The resolution is set such that objects can be differentiated briefly, while choosing a resolution that is as small as possible to keep the observation space small. For this particular environment, the resolution was set to be $32 \times 32 \times 3$.

nine simultaneous areas until the agent learned to score with the correct block. Despite this success, we again were unsuccessful in training the agent with our own implementation. We decided to change our implementation instead of changing our environment further, and tried to improve our implementation using the raycast observation since they had an observation size of 164×1, while the visual observations were of size $32 \times 32 \times 3$ per observation and required a convolutional neural network instead of a simple neural network with two dense layers.

3) Actions: We decided for a discrete action space. At each timestep, the agent was able to perform one out of five actions: do nothing, turn left, turn right, move forward, or move backward.

4) Rewards: Rewards were initially given for scoring a goal with a block: If the block color matched the instruction site color, a positive reward of one was given. If the agent scored with the block of the wrong color, a negative reward of -0.1 was given. We defined those rewards based on reference scenes provided in the ML-Agents toolkit, as well as the general guideline to only provide rewards in the range $[-1; 1]$. For each timestep, a slight negative reward of -0.00005 was given to enforce the agent to act quickly. We tested different reward settings to get to convergence faster and tackle the sparse reward setting: We tried giving small negative rewards for crashing into walls, and slight positive rewards for touching boxes to get the agent to move towards them faster. Instead of improving the time until convergence, our agent learned to avoid blocks too close to walls, and pushed against blocks in the wrong direction. We therefore stuck to the initial reward settings and tried to improve those instead. In particular, we tested different values for scoring with the wrong block. We realized that with our initial negative reward, the agent did not learn properly. Training started with a fairly high average score but did not improve significantly. After changing the reward to 0 for scoring falsely, an actual learning curve could be observed, which paved the way for further improvements that were needed to boost the average score. We also tested whether a small positive reward of 0.1 for scoring falsely would improve the learning further, however it did not. Thus, we ended up with a setup in which a positive reward of one was given for scoring with the correct block, as well as a slight negative reward for each timestep.

C. Tensorflow Implementation

While the code controlling the environment and agent behavior inside the Unity framework has been written in C#, our deep Reinforcement Learning algorithms have been implemented with Python and Tensorflow.

1) ML-Agents Python API: In order to access and communicate with the Unity Environment from a Python script, the ml-agents framework provides the mlagents_envs package and a low-level Python API. With the API, the environment can be loaded from a binary file into a

`UnityEnvironment` object, which then controls the communication flow between Unity and Python. The interface then allows to reset the environment, as well as to request observations and to set actions. Observations and actions are always accessed for a specific agent type and agent ID, which can also be requested from the environment.

2) Vanilla DQN: Our basic implementation is a Vanilla DQN algorithm, which iterates through a specified number of environment episodes. During each episode, the agent takes ϵ-greedy actions until it solved the task, failed the task or the maximal number of steps has been reached. The task is solved, when the correct block has been pushed into the goal area and failed, when the same is true for the incorrect block. In each step, the current observation, the selected action, the resulting observation and the received reward are stored in the experience replay buffer, together with the information of whether a terminal state has been reached. The agent is controlled by a deep Q-network, which is defined and initialized before the start of the main loop, together with a duplicate version that will function as target network. With a specified frequency of steps, the main Q-network is trained for a specified number of training epochs on trajectories that are sampled from the experience replay buffer. Also with a (different) specified frequency of steps, the target network is updated, which means that the weights from the main network are copied to the target network.

To train the Q-network, the Mean Squared Error (`tf.keras.losses.MSE`) between the current prediction and the target is calculated for each trajectory from the sampled batch. The target is computed as $target \leftarrow rewards + \gamma * q'$, where q' is the max q-value for the next observation o' as predicted by the target network. The MSE loss is then backpropagated through the network using the Adam optimizer from the `tf.keras.optimizers` module.

Algorithm 1 Vanilla DQN

Initialize Buffer B, DQN Q_θ and Target Network $Q_{\theta'}$
for N epochs **do**
  Reset env and get initial observation o_t
  while not done **do**
    Select action a_t with ϵ-greedy policy
    Observe o_{t+1}, r_{t+1} and $done$
    Store $(o_t, a_t, o_{t+1}, r_{t+1}, done)$ in B
    if train frequency **then**
      for M train epochs **do**
        $q' \leftarrow max(Q_{\theta'}(o_{t+1}))$
        $target \leftarrow rewards + \gamma * q'$
        $\theta \leftarrow \theta + \alpha * MSE(prediction, target)$
      end for
    end if
    if update frequency **then** $\theta' \leftarrow \theta$
  end while
end for

Different from the original implementation [11], we implemented our DQN as a simple feed-forward network, as it does not need to deal with multi-dimensional visual inputs. The network consists of overall three dense layers, with both the input and the hidden layer having 64 units and a ReLU activation function. The output layer has five units for each possible action and uses a linear activation.

```
HYPERPARAMS = {
    'learning_rate': 0.001,
    'batch_size': 128,
    'discount_rate': 0.99,
    'epsilon': 0.1,
    'train_frequency': 5,
    'train_episodes': 3,
    'target_update_frequency': 5000
}
```

Fig. 3: Hyperparameter configuration for Vanilla DQN.

With this setup, there were multiple hyperparameters that needed to be tuned, both on the deep learning and on the reinforcement learning side. To optimize the deep learning part of the algorithm, we explored different configurations of the learning rate and the batch size. To optimize the reinforcement learning mechanism, we tuned the discount rate γ, the exploration factor ϵ, the frequency of training per environment step, the number of training episodes per train step and the target update frequency per environment step. The best configuration we were able to find can be seen in Fig. 3. Overall, we found that the correct ratio between collecting experiences, training and updating the target network needed to be balanced correctly for the agent to be able to learn.

3) Double DQN: Once we had explored the Vanilla DQN setup sufficiently with different hyperparameter configurations, we looked into algorithmic improvements to our implementation, as the agent had not yet learned to solve the task consistently. One of the most widely used modifications to Vanilla DQNs is Double Deep Q-learning, because it reigns in the innate overestimation bias in DQNs. As overestimation belongs to the biggest problems for DQNs, we hoped to significantly increase the agents ability to learn through the implementation of DDQNs. As explained in detail above, the DDQN algorithm calculates the targets based on both the main q-network and the target network in order to not base both the selection and the evaluation of the next actions on the same q-value. The predictions of the main q-network were then used to obtain the indices of the actions with the maximal q-values (action selection), while the q-values at those indices were extracted from the target network's predictions.

For the implementation we therefore needed to modify the training of the agent such that the parameter update would rely on the new method of target calculation. To this end, we created a subclass of our `DeepQAgent` class called `DoubleDeepQAgent` which overwrites the `learn()`

method that implements the network training. Instead of computing the approximation of the next q-value q' by maxing over the target network's predictions for the next observations, we obtained the q-values for the next observation from both the main q-network and the target network. The predictions from the main q-network were then used to obtain the indices of the actions with the maximum q-value (action selection), while the q-values themselves where taken from the target network at the specified indices (action evaluation).

Algorithm 2 Double DQN

Vanilla DQN
...
if train frequency **then**
  for M train epochs **do**
    $q' \leftarrow Q_{\theta'}(o', argmax(Q_\theta(o_{t+1})))$
    $target \leftarrow rewards + \gamma * q'$
    $\theta \leftarrow \theta + \alpha * MSE(prediction, target)$
  end for
end if
...
Vanilla DQN

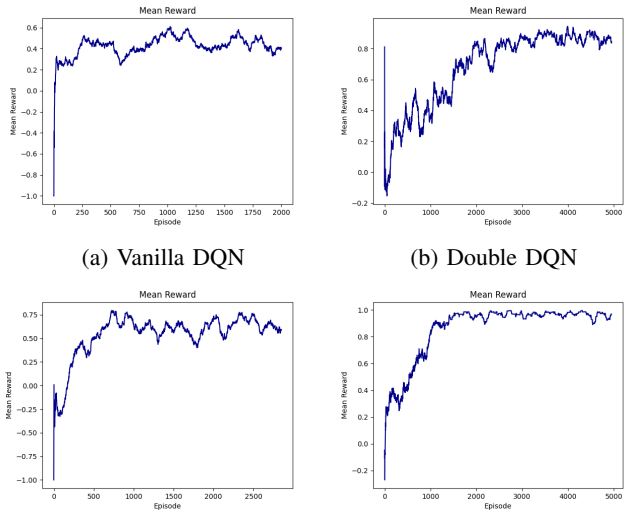
4) Training Modifications: With the implementation of DDQNs, our agent was generally able to solve the task. From this point, we applied several smaller modifications to the training in the hopes of increasing the efficiency of the training.

Learning Rate Decay: To stabilize the model output and improve convergence, we created a learning rate decay schedule using the `ExponentialDecay` option function from the `tf.keras.optimizers.schedules` module. We set the initial learning rate to 0.001 and chose 10000 decay steps and a decay rate of 0.98.

Batch Size Increase: As an alternative to the learning rate decay, we also tried the approach of increasing the batch size. To achieve this, we doubled the batch size after every 500 episodes, starting with an initial batch size of 128.

Huber Loss: The Huber loss clips the errors, which makes it less sensitive to outliers. Thus, we hoped for increased training stability and efficiency when switching the MSE loss for the Huber loss. The switch was conducted very easily by setting the loss function to Tensorflow's `tf.keras.losses.Huber` instead of the previously used MSE loss.

Epsilon Decay: Due to the fact that our task is set in a sparse reward setting and we are applying Q-learning to solve the problem, it is especially important to have a high exploration rate at the start of the task while later on it needs to sufficiently low so that the experience can be exploited. To achieve this exploration-exploitation balance, we added an ϵ-decay schedule to our training. We chose to decay ϵ linearly with a factor of 0.99, starting with ϵ equal to 1 and reducing it step-wise until 0.001.



(a) Vanilla DQN (b) Double DQN

(c) DDQN with Huber loss (d) DDQN with ϵ-decay

Fig. 4: Average scores x episodes that we achieved with our implementation of the named methods.

You can find the implementation of our project here: https://github.com/jlinkemeyer/DRL-MultiAgent

IV. Results

A. Vanilla DQN

With our most basic implementation, the Vanilla DQN algorithm, the agent was able to achieve a maximal average score of 0.6 (see Fig 4a), which is still considerably less then the perfect score of 1.0. Additionally, the agent seems to learn efficiently only during the first 200 episodes, which is followed by a very long phase of oscillating around the sub-optimal averages between 0.4 and 0.6 instead of improving further. Overall, the training exhibits only moderate stability and does not achieve the desired performance.

B. Double DQN

The results of the Double DQN implementation (Fig. 4b) show that the modification indeed significantly improves the performance of the learning algorithm. Instead of plateauing very fast at a sub-optimal score, the agent now learns more steadily until it almost achieves the perfect score. At the end of the training, the agent is now able to successfully push the instructed block into the goal. Even though the training is successful, it still takes the agent 4000 to 5000 episodes to learn how to solve the task. Additionally, the learning curve still includes oscillations.

C. Learning Rate Decay and Batch Size Increase

We initially added the learning rate decay and batch size increase to give more stability to the Vanilla DQN implementation and improve convergence, as the average score quickly dropped again after reaching its peak. There, it seemed to help in the sense of stabilizing the model output, however there still was much room for improvement for the average score.

With the Double DQN implementation, we did not have the issue of the average score dropping anymore and adding learning rate decay or batch size increase did not produce significantly better results.

D. Huber Loss

Although the Huber loss seemed to be a promising method, instead of improving the performance it impacted the training negatively, as shown in Fig. 4c.

E. Epsilon Decay

As can be seen in Fig. 4d, decaying epsilon over the steps taken in the environment lead to major improvements in the training efficiency. The learning curve is much steeper than before and contains considerably less oscillations compared to the previous results. Overall, the number of episodes necessary to solve the task well consistently could be halved from four or five thousand episodes to only two thousand.

V. Discussion

A. Ml-Agents Toolkit

The Unity ML-Agents toolkit for reinforcement learning is an extremely valuable tool for creating one's own three-dimensional environment. Even though, while working with it we not only noticed the many positive features, but also some disadvantages, which is why we will evaluate the toolkit and its usability for reinforcement learning here.

Unity is a game development platform which enables the creation of 3D-scenes of any desired complexity and detail. The Unity ML-Agents toolkit makes those environments available for deep reinforcement learning, which has several advantages. First of all, using Unity makes it incredibly easy and fast to build an appropriate environment for the agent to interact with. The creation of the visual scene itself does not require to write any code and allows to work with visuals directly from the start. It also provides the internal physics engine, such that realistic environment dynamics do not have to be implemented manually. Specific to reinforcement learning, the ML-Agents toolkit also provides a large pool of options for observation types and action spaces, together with helpful guides of how to use them. In our case, as mentioned above, it enabled us to explore both raycast-observation and camera-observations with minimal effort.

The toolkit also provides in-built implementations of PPO and soft actor critics (SAC) which can be run with a simple command. Those algorithms are especially nice for beginners that are just starting out with deep reinforcement learning. However, they are also very useful to test new environments and different reward settings. As described above, this is exactly how we used the PPO algorithm: it allowed us to optimize the environment independent from our implementation which added a considerable amount of efficiency to our working process. The in-built algorithms are highly optimized for not only usability, but also efficiency. They include for example the functionality to run multiple environments within one Unity scene (called areas in this case). The different areas

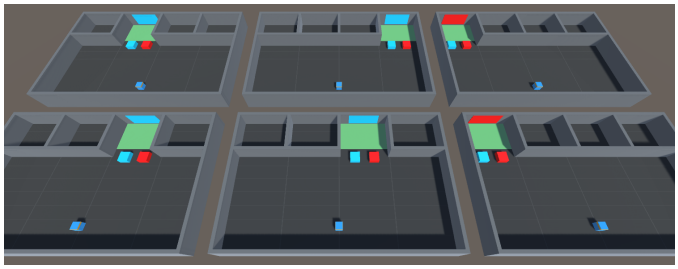


Fig. 5: Training directly in Unity allows to train with multiple areas at the same time, drastically decreasing training time compared to training with a single area.

are run in parallel and asynchronously, which reduces training time drastically. Although those algorithms can easily be used, ML-Agents also comes with a corresponding python package and an API such that own algorithms can be implemented for Unity environments. The possibility to load the environment from a binary file additionally makes sure that no Unity dependencies besides the ml-agents python package, are needed.

While the toolkit overall is a truly great asset, because of all the reasons mentioned above, there were still some problems we encountered while working with the low-level API. It became obvious to us quite fast that the toolkit is optimized for the usage with the in-built algorithms rather than own implementations. This is for example mirrored in the much more extensive documentation on the usage of Unity with the provided algorithms than there is documentation on the Python API. Additionally, the functionality that the API supports is limited. The training of multiple areas in one scene is not possible, as the API does not provide functions that allow to reset individual areas. Overall, the control over the environment and the agent is restricted as they rely on predefined Unity implementations. If such detailed control is wanted, the usage of the toolkit can therefore lead to problems that require to either adapt expectations or change Unity functionality through code on a rather deep level. This leads us to the conclusion that while the Unity ML-Agents toolkit is a fantastic tool for deep reinforcement learning, it might not be the first choice for advanced projects that require a high level of control over all aspects of the implementation. We recommend anyone to explore Unity and ML-Agents, but especially so if the project can profit from the in-built functionalities.

B. Sparse Reward Setting

Coming back to our project, the setup we chose based on [3] required us to handle a sparse reward setting: Final rewards were only provided when scoring a goal with the correct color, and a small negative reward for each passed timestep. [3] provide a solution to this problem: They advise to use the built-in imitation learning functionality. In imitation learning, the agent is provided with trajectory information from those trajectories a human player has obtained when manually controlling the environment. Since this was too complex to

implement within this project, we first decided to use reward shaping instead, even if not advised by [3]. We tested different options such as providing negative rewards for crashing into walls and touching blocks. However, no further rewards we added helped to the agent to solve the task faster but rather introduced undesired behaviors such as avoiding blocks that are positioned too close to walls. We implemented another idea to shape rewards, namely to propagate the last reward from an episode through the trajectory, decaying that reward further moving towards the trajectory's start. Investigations on how well this approach would work to improve training time were stopped due to time constraints.

C. Algorithm Choice

The initial choice to use DQN to solve the given task was based on DQN being used in multi-agent setups. We thus considered it to be sensible to implement DQN also for the single-agent case, to allow for an easy expandability to the multi-agent case. Due to long training times and unstable training, we decided to implement Double DQN to see whether training stability can be improved as proposed by [12]. Our results demonstrate the following: With DQN, the mean reward sum remained unstable between 0.4 and 0.6 for over 1500 episodes without improving further (Fig. 4a). While requiring to train for more episodes, Double DQN was able to reach a much higher mean reward sum and only fluctuated only between 0.8 and 0.9, thus demonstrating the strength of Double DQN. Training results got even more stable and reached convergence faster when we introduced epsilon decay (Fig. 4d), demonstrating that over time, less exploration should be included as the network proceeds to learn.

D. Outlook

There are multiple aspects in which the project could be expanded. The most practical aspect would be the introduction of parallel asynchronously trainable environments, which would help massively in reducing the overall training time. Shorter training times would then allow for more thorough explorations of different hyperparameter configurations and training modifications, possibly leading to better results.

The most relevant extension of our project would be the actual implementation of the multi-agent case, as presented in the original idea. This would require the addition of the multi-agent functionality on the Unity side as well as the implementation of the modified RIAL algorithm with Python and Tensorflow. As we designed the Unity environment and implemented functionalities with the multi-agent case in mind, the extension of the environment would not be too complex. The adaptation would require the following changes: To adapt the overall coordination of the two agents, the C# scripts would have to be modified, especially concerning the rewards that are not given independently, but are shared between the agents. Practically that means that instead of giving the separate agents rewards, an agent group would have to be initialized and rewards would have to be controlled using group rewards. Additionally, the observation spaces would have to be adapted: The agent that pushes the blocks into the goal area (push-block agent) should not be able to perceive the instruction site. This can be realized by simply removing the instruction site tags from the detectable tags in the raycast observation. The action space for the push-block agent remains the same as in the single-agent setup. For the communication agent, the raycast observation can remain the same as the one in the single-agent case. For the communication agent, the action space would have to change: Instead of just being able to move around without being able to push the blocks, this agent would also require to have an option to communicate. Keeping in mind to limit task complexity, we could allow the communication agent to have a vocabulary size of two which should suffice to communicate the relevant information whether the push-block agent should push either the red or the blue block into the goal area. The performed action passed as observation to the push-block agent could thus be either 0 or 1. The two agents would have to develop their own communication pattern, which could be either (0 = red and 1 = blue) or (0 = blue and 1 = red). Once this task would be solved, it could be investigated how communication patterns would evolve with larger vocabulary size in more complex environments e.g. the setup shown in Fig. 1a. [1] report that when given the option, non-verbal communication could evolve, too. It would be interesting to observe whether this would happen for our specific setup.

Besides studying the emergence of communication as described above, the implementation of the RIAL framework would allow to investigate how the type of memory used in the method (DRQN or stacked observations) influences the performance regarding (i) the quality of the results and (ii) regarding the consumption of computational resources. Going one step further, the RIAL framework could be expanded to the Differentiable Inter-Agent Learning (DIAL) framework, which has been proposed by the same authors as RIAL. The DIAL method addresses one of RIAL's disadvantages: The lack of a rich feedback loop that gives the sending agent more immediate information for the evaluation of his utterances. The feedback loop is realized by backpropagating gradients from the the receiver agent to the sender agent through the communication channel. To allow the computation of gradients while maintaining discrete communication during the decentralized execution phase, the activations of the q-network are processed by a so-called discretise/ regularise unit (DRU), which regularises the network output during training and creates discrete communication signals from it during execution. The extension to the DIAL method would be interesting for the study of language emergence, as it more closely resembles the way humans are communicating with each other [7].

VI. Conclusion

With our project, we wanted to investigate the emergence of communication through multi-agent reinforcement learning. To this end, we created a concept for a cooperative multi-agent scenario and implemented a simplified single-agent version as a Unity environment. Using this environment, we were able to

show that the task is solvable with a single agent that receives all necessary information to fulfill the task. We consider this to be important groundwork with regard to the goal to get to multi-agent communication. In the future, we hope that this work is extended to the multi-agent case to contribute to both language studies and deep reinforcement learning by combining both in the setup that we presented here.

REFERENCES

[1] I. Mordatch and P. Abbeel, "Emergence of grounded compositional language in multi-agent populations," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[2] A. Lazaridou, A. Peysakhovich, and M. Baroni, "Multi-agent cooperation and the emergence of (natural) language," *arXiv preprint arXiv:1612.07182*, 2016.

[3] Unity Technologies. Unity ml-agents toolkit. [Online]. Available: \url{https://github.com/Unity-Technologies/ml-agents},lastvisited2022-09-14

[4] L. Busoniu, R. Babuska, and B. De Schutter, "A comprehensive survey of multiagent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 38, no. 2, pp. 156–172, 2008.

[5] A. OroojlooyJadid and D. Hajinezhad, "A review of cooperative multi-agent deep reinforcement learning," *arXiv preprint arXiv:1908.03963*, 2019.

[6] A. M. Hafiz and G. M. Bhat, "Deep q-network based multi-agent reinforcement learning with binary action agents," *arXiv preprint arXiv:2008.04109*, 2020.

[7] J. Foerster, I. A. Assael, N. De Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," *Advances in neural information processing systems*, vol. 29, 2016.

[8] M. T. Spaan, "Partially observable markov decision processes," in *Reinforcement Learning*. Springer, 2012, pp. 387–414.

[9] D. Xie, Z. Wang, C. Chen, and D. Dong, "Iedqn: Information exchange dqn with a centralized coordinator for traffic signal control," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[11] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[12] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[13] K. Nakamura, B. Derbel, K.-J. Won, and B.-W. Hong, "Learning-rate annealing methods for deep neural networks," *Electronics*, vol. 10, no. 16, p. 2029, 2021.

[14] S. L. Smith, P.-J. Kindermans, C. Ying, and Q. V. Le, "Don't decay the learning rate, increase the batch size," *arXiv preprint arXiv:1711.00489*, 2017.

[15] H. Yang, D. Shi, C. Zhao, G. Xie, and S. Yang, "Ciexplore: Curiosity and influence-based exploration in multi-agent cooperative scenarios with sparse rewards," in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, 2021, pp. 2321–2330.

[16] A. Patterson, V. Liao, and M. White, "Robust losses for learning value functions," *arXiv preprint arXiv:2205.08464*, 2022.

[17] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.