TECHNICAL UNIVERSITY OF MUNICH

TUM Data Innovation Lab

Multi-Agent Reinforcement Learning for Logistics

Authors	Andres Becker, Iheb Belgacem, Victor Caceres, Anja Kirschner, Leo Tappe
Mentors	M.Sc. Jorrit Posor (MaibornWolff)
	Dr. Lenz Belzner (MaibornWolff)
Co-Mentor	M.Sc. Oleh Melnyk (Department of Mathematics)
Project Lead	Dr. Ricardo Acevedo Cabra (Department of Mathematics)
Supervisor	Prof. Dr. Massimo Fornasier (Department of Mathematics)

Jul 2020

Abstract

In recent years, significant progress has been made in solving challenging problems across several fields using Reinforcement Learning (RL). Projects like Deep Mind's AlphaGo Zero have showed the amazing results that RL can achieve. Also, more recent results by OpenAI [3] for multi-agent competition in the simple game of hide-and-seek, showed the potential of RL to develop complex and coordinated strategies among the agents to tackle a common objective.

Inspired by these results, a RL approach is proposed to address the logistics and resource management of a chaotic warehouse. An agent (or many) is trained so it can pick and store all in-bound items into the warehouse, so it can later deliver them as required.

Two algorithms were used, Deep Q-Learning for the single agent case and Advantage Actor-Critic (A2C) for the multi agent case. To train the models, dense and sparse reward, as well as several inbound and outbound items request schemes were tried. A curiosity-driven exploration module was implemented alleviate the sparse reward problem. To model the chaotic warehouse, an environment using GYM was developed.

For the single agent approach, a heuristic baseline was use to compare the performance of the RL Model. In the cases analyzed, the RL approach proved to achieve better performance than the heuristic baseline. For multi agent RL, we had to choose a different approach than DQN and settled for A2C, which still led to reasonable results. Overall, the results obtained showed that RL is a promising approach to manage the logistics of a Chaotic warehouse in an efficient way.

Contents

1 Introduction

1.1 Problem statement

The Challenge proposed consists in a Chaotic Warehouse which contains bins, item types and transactions of these items. In-bound transactions or deliveries of different kinds of items are received by the warehouse and must be handled by agents. Out-bound transactions or requests of these items are also received and must be also handled by agents. There are also bins inside this warehouse where items can be stored in and retrieved from.

The problem presented is solving this Chaotic Warehouse by giving the agents a strategy to handle all kinds of possible situations. Numerous heuristics are usually employed in order to optimize patterns, such as minimum walking distance.

Reinforcement Learning can be applied to this Chaotic Warehouse setup in order to discover strategies that may lead to better results by uncovering better placement or action prioritization.

The simulation of a Chaotic Warehouse offers a real-world problem scenario where Reinforcement Learning can be used and its solutions directly applicable to modern warehouses.

1.2 Chaotic Warehouse

A Chaotic Warehouse consists of the following components:

- Staging-in area: Where items are received
- Staging-out area: Where items are requested
- Bins: Multiple spaces that hold items
- Agents: Move around handling items
- Items: Different types
- Inbound-transactions: Items to be picked up from the staging-in area
- Outbound-transactions: Items to be delivered to the staging-out area

1.3 Project Goals

We set the following goals:

- Implement a working environment that resembles a Chaotic Warehouse
- Implement visualization capabilities of this environment
- Implement a single agent reinforcement learning algorithm that handles different warehouse complexities
- Implement a multi agent reinforcement learning algorithm mirroring single agent cases
- Compare the performance achieved with a heuristic baseline

2 Theory

2.1 Deep Reinforcement Learning Overview

Reinforcement Learning's (RL) idea is to learn from interaction. Instead of learning from knowledge (supervised learning) or by trying to find some structure in unlabeled data (unsupervised learning), in RL an *Agent*, who interacts with an *environment*, learns how to get a *reward* from this environment just by interacting with it. Ultimately, the aim of this agent is to maximize the reward in the long run (the *Return*). Figure 1 shows an example of this interaction between an agent and the environment.

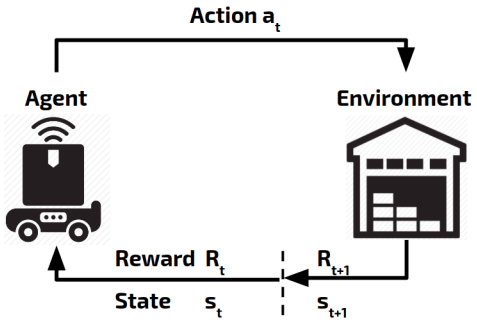


Figure 1: Interaction between an Agent and the environment.

As shown in Figure 1, on each time step t the agent observes the current state s_t of the environment, based on that and on its *policy* π, the agent decides to execute the *action* a_t in the environment, that in return feeds the agent with the next state s_{t+1} and a reward R_{t+1}.
This interaction is an example of a Markov Decision Process (MDP)[8] $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R)$, where:

- $\mathcal{S}$ is a finite set of states with Markov property (future depends only on current state s_t and not on preceding states $s_1, \ldots, s_{t-1}$).
- $\mathcal{A}$ is a finite set of actions.
- P is the transition probability function ($P(s_{t+1}, r_{t+1}|s_t, a_t)$).
- R is the reward function ($R(s_t, a_t) := \mathbb{E}[R_{t+1}|s_t, a_t] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} P(s', r|s_t, a_t)$).

The other Key elements of a RL model are:

- Return $G_t := \sum_{k=0}^{T} \gamma^k R_{t+k+1}$; reward in the long run. Where $\gamma \in [0, 1]$ is the discount factor and T is the maximum number of time steps per episode.
- Policy $\pi : \mathcal{S} \to \mathcal{A}$; Tells the agent which action to take given a state s.
- State-value $V_\pi(s) := \mathbb{E}_\pi[G_t|s_t = s]$; expected return by following π when in state s.
- Action-value $Q_\pi(s, a) := \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$; expected return when taking action a from state s by following π.

2.2 Algorithms

2.2.1 Deep Q-Learning

Recall that on each time step t we want to maximize the Return G_t, so we should chose $a \in \mathcal{A}$ such that $Q_\pi(s_t, a)$ is maximum. Since we have a finite number of actions and states, theoretically we could store all state-action pairs in a table represented by $Q_\pi(.,.)$. However, it quickly becomes computationally infeasible when the state and action space are large. Therefore instead of this, we use a Deep Neural Network as a function approximation of Q_π (see Figure 2). This is called Deep Q-Learning.
Deep Q-learning is a model-free and off-policy reinforcement learning algorithm. This means that it does not require prior knowledge of the environment (i.e. model-free) and that it updates its Q-value

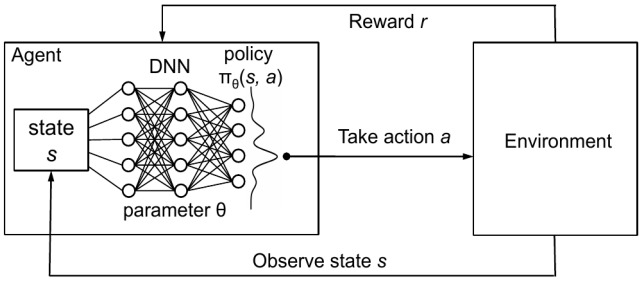


Figure 2: An artificial deep neural network to approximate Q_π. Image source: [3].

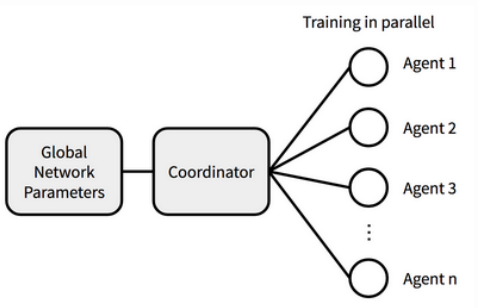
function (Q_π) using the Q-value of the next state s' and the greedy action a' (i.e. off-policy). It also uses *Temporal-Difference* (TD), which means that it updates its Q-value function, by bootstrapping a sample from its *Replay buffer* (a set o fixed size containing tuples ($s,\ a,\ r,\ a'$) from previous episodes).
The whole process is shown in algorithm 1.

Initialize replay memory D to capacity N;
Initialize action-value function Q with random weights θ;
Initialize target action-value function $\hat{Q}$ with weights $\hat{\theta} = \theta$;
for *episode=1* **to** M **do**
  Initialize state s_1;
  for t=*1* **to** T **do**
    ε-greedy: With probability ε select a random action a_t, otherwise select a_t=argmax$_a Q(s_t, a|\theta)$;
    Execute action a_t in the Environment and observe reward r_t and next state s_{t+1};
    Store transition (s_t, a_t, r_t, s_{t+1}) in D;
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D;
    $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'|\hat{\theta}), & \textit{otherwise} \end{cases}$;
    Perform gradient descent step on $(y_j - Q(s_j, a_j|\theta))^2$ w.r.t. θ;
    Set $s_t = s_{t+1}$;
    Every C steps reset $\hat{Q} = Q$;
  end
end

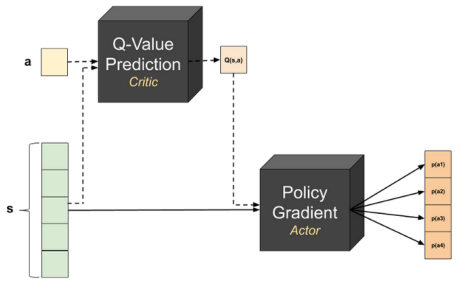
Algorithm 1: Deep Q-Learning Algorithm.

2.2.2 Advantage Actor-Critic

In contrast to Deep Q-Learning, Advantage Actor-Critic (A2C) does not only try to learn the advantage values but combines it with policy-gradient methods ([6], [5]). The main idea behind this is to split the agent into two subagents, one, the *critic*, for estimating the value function and another one, the *actor*, for updating the policy distribution in the direction suggested by the critic (see Figure 3b).



(a) The architecture of A2C [5]



(b) A schematic view of an Actor-Critic architecture. Dashed lines represent flows only relevant during training, so during inference (prediction) phase, only the Actor is being used [6]

Figure 3: Advantage Actor Critic

The advantage value A measures the advantage of choosing a certain action a at a certain state s and can be calculated by subtracting the state value from the action value (Q-value), $A(s,a) = Q(s,a) - V(s)$. Policy gradient algorithms learn the policy directly, meaning that the network takes the current state s as input and give as output the probability $p(a)$ to select each action a. The network weights θ are then updated in each step by

$$\theta_{t+1} = \theta_t + \alpha(Q(s_t, a_t) \cdot \nabla_\theta \ln \pi(a_t|s_t, \theta_t)). \tag{1}$$

Another big difference to Deep Q-Learning is the avoidance of replay buffers. There are several downsides to them, such as the use of memory and the amount of time needed per each real interaction, but the main reason is that as A2C is an on-policy algorithm, replay buffers can not be used since they require off-policy algorithms. To still be able to stabilize learning, multiple agents are executed in parallel (see Figure 3a) such that the parallel agents will face many different states at any given time step.
During that parallel training, the coordination module handles the agents and the global network parameters are updated periodically, i.e. after all agents have finished their segments, the global network weights are updated and the agents are reset.

2.2.3 Curiosity

The chaotic warehouse environment provides the agent only with sparse rewards: the agent doesn't get any feedback from the environment in most of the states. It receives a positive reward only after realizing a successful transaction. Figuring out which sequence of actions has led to the reward is challenging. The curiosity-driven exploration approach, developed in 2017 by Pathak D., Agrawal P., Efros A. and Darrell T., is one of the most successful approaches that have been developed in the recent years to tackle the sparse rewards problem [4].
The goal of this method is to increase the agent's general knowledge of the environment. The agent, in this approach, learns to predict using the current state and the selected action, the next state of the environment. The discrepancy between the predicted state and the true state is larger for regions the agent has not explored well yet. To incentivize the agent to explore extensively its environment, it receives a reward proportional to the prediction error.
In the formulation of the method proposed in [4], a feature representation for the states $\phi(s)$ is learned using a deep neural network. A forward model is developed to predict the feature representation of the next state given the current state and the selected action. Simultaneously, an inverse model is trained to predict from the feature representation of the state at timestep t and $t+1$ the action that led to this transition. The goal of this architecture is to make the feature extraction procedure unaffected by the aspects of the state the agent cannot control.
In the case of the chaotic warehouse environment, the state space doesn't contain by construction any information irrelevant to the task of the agent. So we have implemented a simpler version of the Intrinsic Curiosity Model proposed in [4]. We have used a deep neural network (corresponding to the forward model) that tries to predict from the observation of the current state and the choice of the action the information about the next state.
Our agent is trained to optimize the sum of the intrinsic and extrinsic rewards.

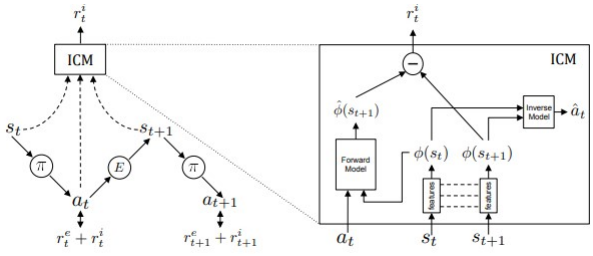
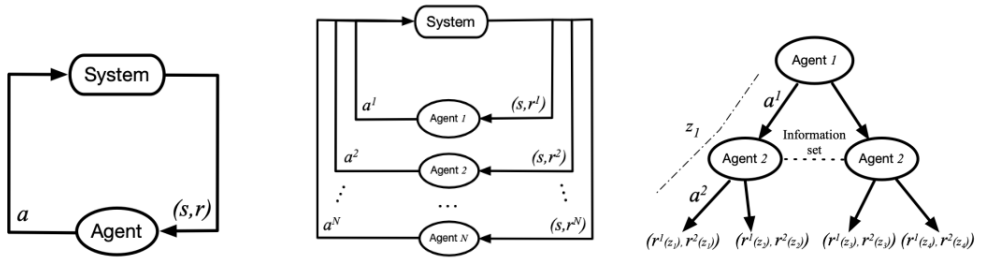


Figure 4: Intrinsic Curiosity Module. Image source [4].

2.3 Multi-Agent Reinforcement Learning

In Multi-Agent Reinforcement Learning, there are two main approaches in how to train agents, *Joint Action Learners* (JAL) and *Independent Learners* (IL). The main difference between these is that independent learning agents do not know which actions the other agents have chosen while joint action learning agents have full information about it [9].



(a) MDP (b) Markov game (c) Extensive-form game

Figure 5: Frameworks of single- and multi-agent RL.

To be more precise, in JAL scenarios, after an agent performed an action – knowing which actions the other agents chose – it receives its reward together with the information which rewards the other agents got for their actions (see Figure 5 (b)). In a *cooperative setting*, the agents share a common reward function and are also referred to as *multi-agent MDPs* (MMDPs), whereas the reward function in a *competitive setting* is also called *zero-sum* game, meaning that the rewards of all agents for one step sum up to 0 [10]. *Mixed settings*, also known as *general-sum* games, have no restriction on how to set up the reward function.
Joint action learning can be formalized by the theory of *Markovian Games*. A Markovian Game is defined as a tuple $(\mathcal{N}, \mathcal{S}, \{\mathcal{A}^i\}_{i\in\mathcal{N}}, \mathcal{P}, \{R^i\}_{i\in\mathcal{N}}, \gamma)$, where

- $\mathcal{N} = \{1, \dots, N\}$ is the set of $N > 1$ agents,
- $\mathcal{S}$ is the state space observed by all agents,
- $\mathcal{A}^i$ is the action space for an agent $i \in \mathcal{N}$,
- $\mathcal{A} := \mathcal{A}^1 \times \cdots \times \mathcal{A}^N$ is the joint action space,
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the transition probability from any state $s \in \mathcal{S}$ to any state $s' \in \mathcal{S}$ for any joint action $a \in \mathcal{A}$ and
- $R^i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function received by agent i for a transition from (s, a) to s'.

Based on the current system state s_t, at each time step t every agent $i \in \mathcal{N}$ performs an action a_t^i, leading the model to transition into state s_{t+1}. Each agent i then receives an individual reward $R^i(s_t, a_t, s_{t+1})$. In the long run, every agent wants to maximize its own long-term reward, thus aiming to find its policy π^i. In short, the other key terms of Markovian Games are:

- Agent i's policy: $\pi^i : \mathcal{S} \to \Delta(\mathcal{A}^i)$
- Agent i's state-value: $V^i : \mathcal{S} \to \mathbb{R}$
- The model's joint policy: $\pi : \mathcal{S} \to \Delta(\mathcal{A})$, $\pi(a|s) := \prod_{i\in\mathcal{N}} \pi^i(a^i|s)$
- Agent i's value function: $V^i_{\pi^i,\pi^{-i}}(s) := \mathbb{E}[\sum_{i\geq 0} \gamma^t R^i(s_t, a_t, s_{t+1}|a_t^i \sim \pi^i(\cdot|s_t), s_0 = s]$, where $-i$ stands short for $\mathcal{N} - \{i\}$.

On the other hand, an independent learner has no information about the actions that the other agents have chosen and, in fact, treats those agents as part of the environment. These learners are in general applied to non-cooperative settings.
Independent learners can be modeled as *Extensive Form Game*. In comparison to Markovian Games, the main difference is that there is no joint action space. Instead, the action space is the same for all agents, $\mathcal{A} = \mathcal{A}^1 = \cdots = \mathcal{A}^N$ [10]. Also, the agents perform their steps successively (see Figure 5 (c)).

3 Modelling & Implementation

In this section, we describe how we modelled and implemented the chaotic warehouse. In particular, we state the components of the *Markov Decision Process* that represents our warehouse, and we explain how we implemented it in *Python* by conforming to the *OpenAI Gym*[1] interface. Then, we will briefly describe a heuristic baseline that we implemented, and the library we used to run out-of-the box RL algorithms on our environment.

3.1 Mathematical modelling

The warehouse is modelled as a bounded 2D grid consisting of cells. The staging areas, bins, and agents (from now on referred to as *warehouse objects*) each occupy at every time step exactly one cell in the grid. Further, warehouse objects are capable of holding a limited number of items. This is modelled by giving each of them a fixed number of slots to store items in. Each item has the same size, regardless of type, and occupies exactly one slot. A screenshot of the rendered warehouse from our implementation can bee see in Figure 6 below.

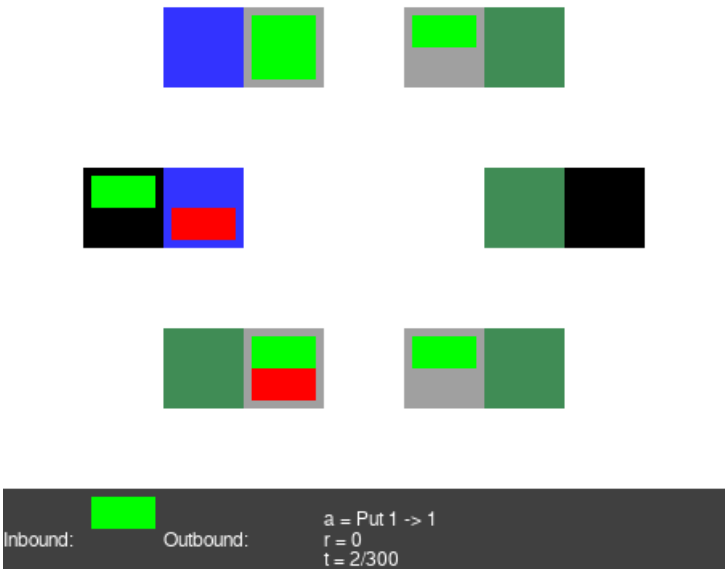


Figure 6: A warehouse frame rendered using `pygame`.

The environment is episodic, that is, an episode lasts for a fixed number of timesteps, and the goal for the agents is to complete as many transactions in an episode as possible. There are a few parameters of the warehouse which influence the size of the state and action spaces described below. These parameters are listed in table 1.

3.1.1 States

A state $s \in \mathcal{S}$ contains the following information:

- The positions of the agents
- The holding status of the agents
- The holding status of the staging areas
- The holding status of the bins
- The current time step (to ensure the Markov property)

Parameter	Description
R	The number of rows in the grid.
C	The number of columns in the grid.
T	The number of time steps per episode.
N_{agent}	The number of agents in the warehouse.
N_{bin}	The number of bins in the warehouse.
N_{item}	The number of distinct item types.
S_{agent}	The number of slots of the agents.
S_{bin}	The number of slots of the bins and staging areas.

Table 1: Parameters of a warehouse.

We represent the state as a discrete vector $s \in \mathbb{Z}^N$, where

$$N = \underbrace{N_{\text{agent}}}_{\text{for each agent}} \times \underbrace{(2 + S_{\text{agent}})}_{\text{position + holding status}} + \underbrace{(N_{\text{bin}} + 2)}_{\text{for bins + staging areas}} \times \underbrace{S_{\text{bin}}}_{\text{holding status}} + \underbrace{1}_{\text{time step}}$$

Each slot is encoded by an integer corresponding to the type of item in that slot, where an empty slot is represented by 0. Moreover, transactions are implicitly encoded via the holding status of the staging areas: an item in the staging-in area needs to be moved out of there, and an item in the staging-out area represents a request for that item, i.e. the requirement that such an item should be moved there. For example, the state of the warehouse in Figure 6 is given by the vector

$$s = (\underbrace{1, 2, 0, 0}_{\text{Agent 1}}, \underbrace{3, 2, 0, 1}_{\text{Agent 2}}, \underbrace{2, 0, 0, 0}_{\text{Staging areas}}, \underbrace{2, 2, 2, 0, 2, 1, 2, 0}_{\text{bins}}, \underbrace{2}_{\text{time step}})^{\mathrm{T}}$$

Note that it is unnecessary to include the positions of the bins or the staging areas in the state, as these are constant. Also, for the purposes of deep reinforcement learning, a flat discrete vector is not necessarily the most useful representation, as neural networks usually have trouble with categorical inputs. Therefore, a learning algorithm can first transform the discrete vector it receives from the environment into a suitable format, for instance via a one-hot encoding.

3.1.2 Actions

The action space $\mathcal{A}$ of each agent consists of two general types of actions: movement actions and pick/put actions.

Movement actions are used to navigate the warehouse. We implemented two movement models which differ in the motion primitives available to the agents.

- In the *low-level* movement model, an agent picks a direction $d \in \{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ which causes it to move from its current cell into the adjacent cell in the chosen direction, provided that cell is not occupied by an obstacle. Obstacles can be the boundary of the grid, the staging areas, the bins, and other agents.
- In the *high-level* movement model, an agent picks a goal location p from a selected subset of goal locations in the grid and simply moves there, again provided that cell is not occupied. This subset of goal locations is chosen to be the set of locations in which agents can interact with the staging areas and bins (see *access spots* below). The benefit of this model is that the agents will only ever move to "interesting" locations without aimlessly wandering around the warehouse.

It should be noted that high-level movement has different implications in the single-agent and the multi-agent scenario. If there is only a single agent, it never hurts to take the shortest path to a desired goal location. As efficient algorithms for finding shortest paths exist, high-level movement can be regarded as delegating the task of finding such a path to a subroutine. It is then also easy to account for the time it takes to move to a desired goal location by incrementing the time step by the length of the shortest path to the goal.

In contrast, for multiple agents, the situation is not quite as simple. First, the shortest paths of the individual agents might lead them to collide with each other, and so are not necessarily the optimal choice. Further, as the environment progresses in discrete time steps, it is not easy to account for the time the agents take to reach their destination with a single step in the high-level movement model, as they might cover different distances. Therefore, in the multi-agent case, the high-level movement model should be treated as the ability to "teleport" in a single time step to the desired goal location.

Pick/put actions are used to interact with the staging areas and the bins (from now on referred to as *containers*). In order to interact with a container, an agent has to be in a designated *access spot*. Each container has such a set of access spots associated with it, i.e. a set of positions in the grid that an agent has to be in so that it can interact with the container. Once an agent is in such an access spot, it can interact with the corresponding container by providing three pieces of information:

- which slot of the container to use for the interaction
- which slot of itself to use for the interaction
- whether to pick an item from the container, or to put an item into the container

Then, the agent will either pick the item currently located in the selected container slot into its selected agent slot, or put the item currently located in its selected agent slot into the selected container slot. This only has an effect if the "target slot" is free at the time of performing the action, so for instance, an agent can't put an item into a slot if there already is an item there. Furthermore, agents can't put items into the staging-in area, they can't pick items from the staging-out area, and they can only put items into the staging-out area that are requested there.

In total, we get an action space of size

$$|\mathcal{A}| = \underbrace{N_{\text{agent}}}_{\text{For each agent}} \times \left(\underbrace{M}_{\text{Movement}} + \underbrace{2 \times S_{\text{bin}} \times S_{\text{agent}}}_{\text{Picking/Putting}} \right)$$

where M is either 4 (in case of low-level movement) or the total number of access spots over all containers in the warehouse (in case of high-level movement).

3.1.3 Rewards

We implemented a variety of different reward schemes, the two most useful ones are these:

- *Dense Reward*: The agents get a reward of 1 whenever they pick up an item from the staging-in area or put a (requested) item into the staging-out area. Otherwise, they get a reward of 0.
- *Sparse Reward*: The agents only get a reward when they complete an entire transaction, i.e. completely clear the staging-in area or staging-out area.

Additional variations could for instance incorporate the current time step or negative rewards for illegal actions.

3.2 The `WarehouseEnv` Gym environment

In order to implement the warehouse such that it is easy to use and run standardized algorithms on, we decided to conform to the Gym[1] interface. Gym is a Python toolkit by OpenAI for developing and comparing reinforcement learning algorithms. It provides a simple interface for reinforcement learning environments via the class `gym.Env`. To use it, one simply has to subclass it and implement a few methods, as in the snippet below:

```
import gym
```

```
class WarehouseEnv(gym.Env):

    def __init__(self, filename):
        """
        Initializes the environment.
        """
        ...

    def step(self, action):
        """
        Advances the environment by one time step, given an action
        by the agents. Returns the new state and a reward.
        """
        ...

    def reset(self):
        """
        Resets the environment to an initial state. To be called when
        an episode ends. Returns the new initial state.
        """
        ...

    def render(self):
        """
        Renders a visualization of the environment to the screen.
        """
        ...
```

As we wanted to keep the design flexible, we didn't hard-code a specific warehouse. Instead, we developed a JSON schema that lets one store all the necessary information about a warehouse in a file. Such a file can then be used to instantiate a concrete `WarehouseEnv` in Python. The schema can be found in the appendix.

3.2.1 The `__init__` method

This is the constructor that is executed when the environment gets created. To be able to create different warehouse environments, the name of a JSON file conforming to the schema above is passed as an argument to the `__init__` method, which then reads the warehouse information from the file.

Furthermore, the `action_space` and `observation_space` member variables have to be set. These let agent code know about the shape and type of observation it can expect, and about the type of action the environment expects from the agent code. Mainly, this is used to distinguish between discrete and continuous states and actions. As we have discrete vectors as states and a discrete number of actions for each agent, we settled on `gym.spaces.MultiDiscrete` observation and action spaces. However, to more easily interact with existing algorithm libraries, we chose a `gym.spaces.Discrete` action space for the single agent case.

3.2.2 The `step` method

The `step` method gets the actions chosen by the agent as a parameter, and modifies the environment accordingly. This means that an agent's position is updated in case of a movement action, or the holding status of the agent and some container are updated in case of a pick/put action. This results in a new state stored in a `numpy.array`, as specified by the observation space.

Further, the reward for the agents is calculated. In case of a single agent, the reward is just a single number, whereas for multiple agents, we implemented both the option of separate rewards for the agents,

as well as the option of a single reward for all agents. In the latter case, the individual rewards simply get summed up.

Then, the current timestep is incremented and compared to the time limit to check whether the episode is over. New inbound and outbound transactions are generated according to a chosen transaction scheme, see the next section for details.

Finally, the new state, the rewards, a flag indicating whether the episode is over, and a dictionary containing additional metatada are returned as a tuple `(obs, reward, done, info)`. The agent can then use this information to do reinforcement learning.

3.2.3 The `reset` method

The `reset` method is called upon termination of an episode. It resets the warehouse either to its initial state, or to a random initial state. The current time step is reset to 0, and the new initial state is then returned.

3.2.4 The `render` method

The `render` method allows the user to visualize the warehouse. In particular, it renders a current representation of the environment to the screen. We chose to use `pygame`[7] to do this. It shows the warehouse as a 2D grid, where the color of a grid cell indicates whether the cell is occupied by a bin, staging area, agent, and whether it's an access spot or simply empty. The holding status of a warehouse object is visualized by subdiving the corresponding cell into slots and coloring each slot according to the type of item in it. Finally, a panel at the bottom of the screen contains information about the current transactions, time step, action, and reward. See Figure 6 for an example of a frame rendered using the `render` method.

3.3 Heuristic baseline

For the purpose of having a baseline to compare our reinforcement learning agents against, we implemented a simple heuristic algorithm. The agents act independently from each other. Each agent, at any given time step, chooses its action according to the following simple algorithm:

1. If it can perform a good pick action, do so.
2. Else, if it can perform a good put action, do so.
3. Else, if it can move somewhere where it could perform a good pick action, do so.
4. Else, if it can move somewhere where it could perform a good put action, do so.
5. Else, perform a random action.

The word "good" in this context means the following: picking from the staging-in area and and putting into the staging-out area is always good. Picking from a bin is only good if the item is needed at the staging-out area, and putting into a bin is only good if the item is not needed at the staging-out area.

3.4 Stable-Baselines

Over the course of this project, we have used for the Deep Reinforcement Learning algorithms the implementations from Stable-Baselines [2], which is a popular open source Python library in the RL research community.
Stable Baselines is a a fork of OpenAI Baselines. It is a repository that provides implementations of many popular Deep Reinforcement Learning algorithms such as Deep Q-Learning, Advantage Actor-Critic, and Deep Deterministic Policy Gradient.
It is a very well documented repository that incorporates many of the recent improvements to the Deep RL algorithms. The Deep Learning models are implemented with Tensorflow and support for Tensorboard is provided. The algorithms support environments that inherit the Env class of OpenAI Gym.

4 Results & Evaluation

4.1 1 Slot 1 Item Case

The first simple case implemented was used as a baseline and as a starting point. We used a normal movement scheme of up, down, right, left, a sparse reward setup, giving a +1 reward for every out-bound transaction completed and a continuous episode setup where the agent tried to finish as many transactions as possible. An initial state of the warehouse was also fixed.

Number of Bins	4	Neural Network	32x16
Item types	1	Number of different states	Aprox 7 000
Item slots	1		

Table 2: 1 Slot 1 Item Case.

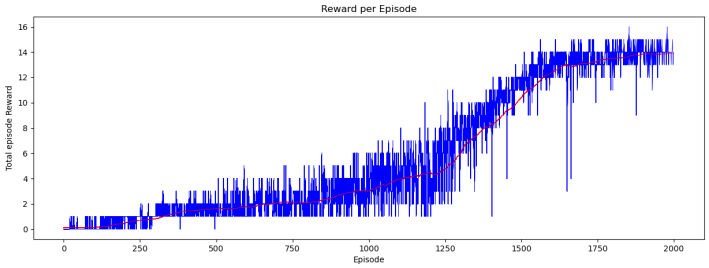


Figure 7: Reward Plot 1 Slot 1 Item.

The Reward Plot (Figure 7) indicates that the training was successful and the agent is able to carry out transactions in this environment.

4.2 Growing Complexity

This setup struggled with increasing the number of bins, the number of items or the number of item slots, as the total number of different states grew exponentially. Here we tried the following scenario:

Number of Bins	4	Neural Network	512x128x32
Item types	2	Number of different states	Aprox 60 000 000
Item slots	2		

Table 3: 2 Slot 2 Item Case Attempt.

The Reward Plot (Figure 8) indicates that the training was not successful and the agent is unable to carry out transactions in this environment.

4.3 Low and High Level Movement

We understood that the agent, in order to execute transactions, first had to learn how to move around. The movement part was holding back the agent and adding unnecessary complexity to the already existing complex task. Thus, we switch to the high-level movement scheme. This movement part is then a subtask which can be solved and incorporated by another module later on. Distance can be also taken into account by making the environment track the distance between spots and adjust rewards proportionally to the remaining time.

4.4 Sequential and Episodic Transactions

We understood that reducing complexity was crucial in order to train successful models. We decided to cut the long sequential episode into one single transaction episode. Each episode started with a completely

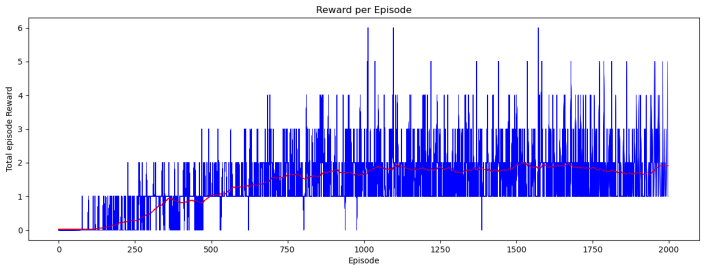


Figure 8: Reward Plot 2 Slot 2 Item Case Attempt.

random state of the warehouse including a random in-bound or out-bound transaction and, as soon as this transaction was completed, the episode ended and a new one, again random, would begin. The agent then had to solve this mini-setup which enabled it to also solve a continuous flow of episodes without random reinitializations later in testing.

4.5 Curiosity

The Curiosity Module described in implementation is used to attempt to improve training of the agent's neural network, whether making it faster or possible in larger complexity scenarios.

4.6 Other Considerations

We also attempted other strategies such:

- Variations of a continuous state input with some variables using a one-hot encoding instead of a completely discrete input.
- Various picking-putting schemes that made it easier for the agent instead of the basic per-slot basis.
- Various reward schemes, giving punishments for illegal action attempts or intermediate positive rewards.

All these strategies did not have good results and thus were discarded for the later scenarios.

4.7 Scaling: bins, items, slots

4.7.1 2 Slot 2 Items Case

By the combination of high-level movement scheme and episodic transactions, we were able to solve this more difficult scenario. The amount of time steps needed for each episode is now relevant, because as the model optimizes its learning, it requires less time steps to fulfill a transaction.
The reward plot is not as relevant as it can be deduced that when the time step does not reach its maximum then the agent was able to get the reward. The mean time step per episode reducing over time means an optimization of the agent's behaviour (Figure 9).

The time step plot shows that the training works and that it continuously optimizes the agent's behaviour as the training progresses, meaning ti takes less time for the agent to carry out a transaction.

Number of Bins	4	Neural Network	512x128x32
Item types	2	Number of different states	Aprox 1 000 000
Item slots	2		

Table 4: 2 Slot 2 Item Case.

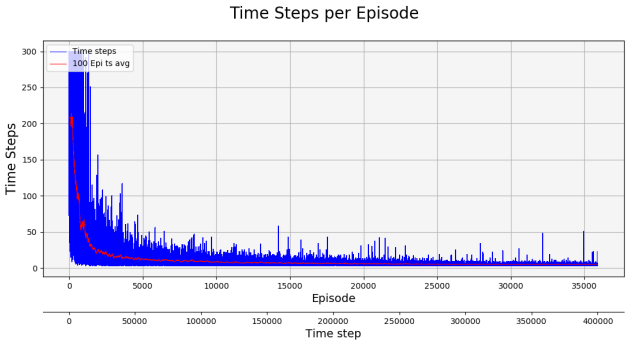


Figure 9: Time step Plot 2 Slot 2 Item Case.

4.7.2 2 Slot 2 Items Case - Distance

To incorporate distance optimization, the environment must take into account the distance between movements and the reward being proportional to the remaining time. The above experiment is repeated. Training takes longer but now the agent also prioritizes short distances. The reward plot is relevant as the time steps does not always have the same maximum, however they lower over time (Figure 10). The agent is able to find an optimum strategy that also optimizes distances and thus the reward increases (Figure 11)..

The time step plot once again shows that it takes progressively less time for an agent to complete a transaction. The reward plot indicates the the agent get more reward as it optimizes itself, as the reward is proportional to the time it takes for each transaction. It means the the agent successfully prioritizes moving short distances over longer distances.

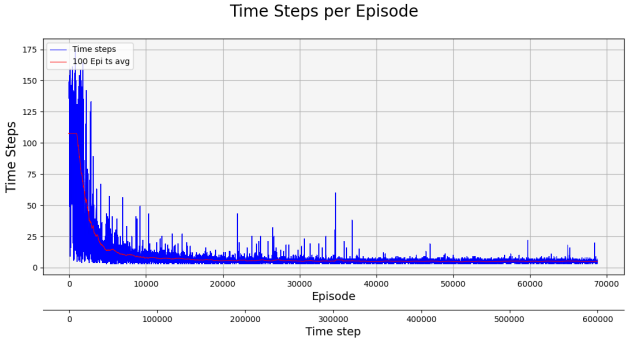


Figure 10: Time step Plot 2 Slot 2 Item Case - Distance.

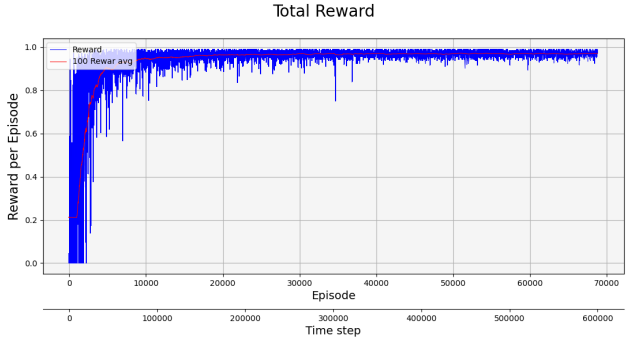


Figure 11: Reward Plot 2 Slot 2 Item Case - Distance.

4.7.3 2 Slot 3 Items Case

This scenario was also solved using the same strategy as before. However, the training time started to be significant. While the 2 slot 2 item case took around 4 hours to train, this case took around 14 hours to solve and required much more time steps due to the increased complexity and a larger neural network required (Figure 12).

The plot results once again show the the agent is able to handle this type of environment as is able to optimize its behaviour as it trains.

Number of Bins	4	Neural Network	1024x512x128x32
Item types	3	Number of different states	Aprox 200 000 000
Item slots	2		

Table 5: 2 Slot 3 Item Case.

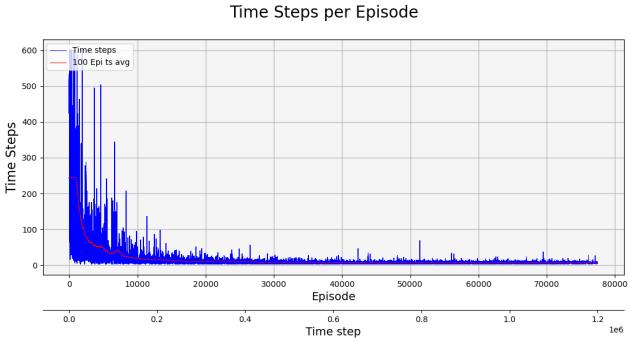


Figure 12: Time step Plot 2 Slot 3 Item Case.

4.7.4 2 Slot 3 Items Case - Curiosity

The previous experiment was repeated now incorporating curiosity. The adjustment made was to increase the transaction completed reward from 1 to 100 in order to make the agent prioritize it instead of the curiosity reward (Figure 13).

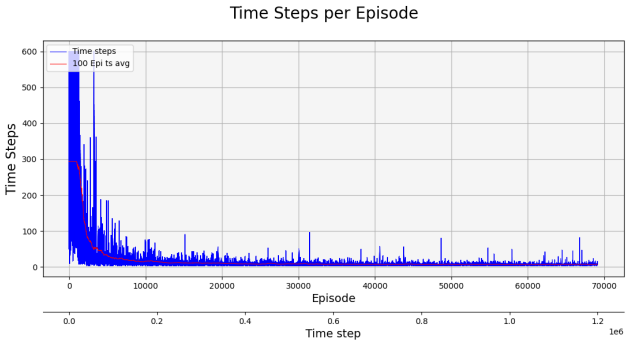


Figure 13: Time step Plot 2 Slot 3 Item Case with Curiosity.

A simulation (Figure 14) was made to compare both models. They were each ran 100 times.
The non-curiosity seems to be more stable which may be due to the reason that it requires a shorter time to train, as the curiosity network needs to converge before the performance of the agent's network starts to increase. It may also suggest that perhaps curiosity might not have as big of an influence in this warehouse scenario. Here, the environment does not change with progress unlike other scenarios where curiosity has been implemented, where the environments change as an agent progresses through it.

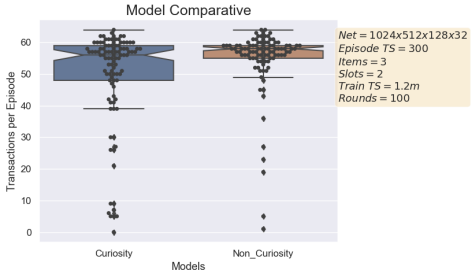


Figure 14: Time step Plot 2 Slot 3 Item Case Model Comparison.

4.8 Boosting

Ensemble methods are popular machine learning techniques that help achieve a high prediction accuracy by combining several base models. We tried to apply this technique to our reinforcement learning problem. We trained several RL agents to compare the performance of each agent to the model combining all of them. To obtain this model, we have summed the action selection probabilities for the different agents and selected the action with the highest probability.
The figure shows that the boosting agent has a more stable performance. In opposition to the base agents, the boosting never has rounds with only few successful transactions.
We expect to see more noticeable improvements if we have a bigger variety of models as ensemble methods are in general more efficient when the algorithms' predictions are less correlated.

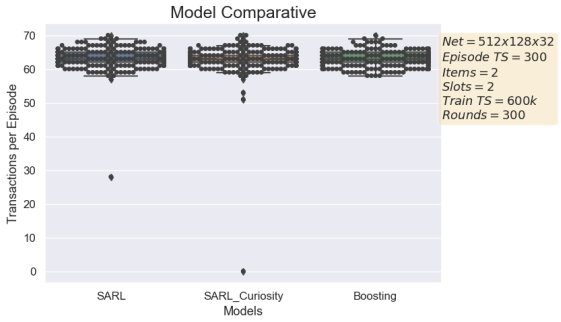
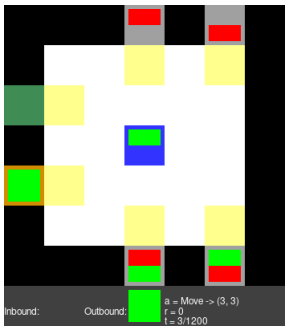


Figure 15: Boosting

4.9 Low Level Movement Environment

Using a low-level movement environment (the agent can move $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$ across the warehouse instead of jumping into the access spots), we conducted 3 simulations increasing gradually the complexity and using the follow general environment and model conditions:



- 7x7 grid (but only 5x5 for movement).
- 4 bins.
- Single agent.
- Episodic transactions[1] and limit of 1200 time steps.
- Deep Q-Network architecture: [128, 64, 32].
- Sparse Rewards[2].

The number of items, number of bin slots, number of agent slots and training time steps, vary among the 3 simulations showed in figures 16, 17 and 18.

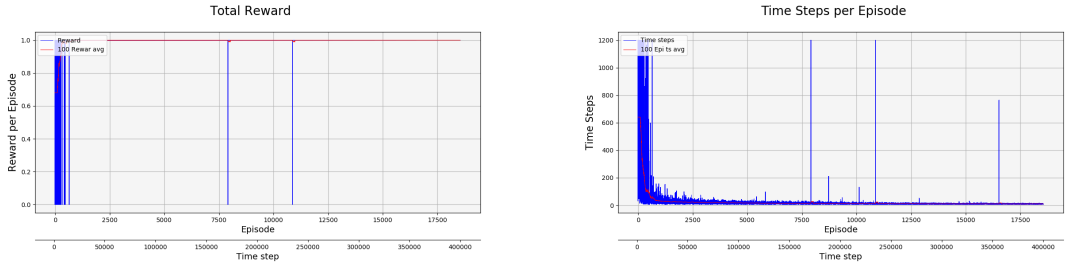


Figure 16: Level 1 Complexity. 400k time steps of training; 1 bin and agent slot, 3 Items; $25 \cdot 4^7 \approx 410$k environment states; $4 + 2 \cdot 1 \cdot 1$=6 actions.

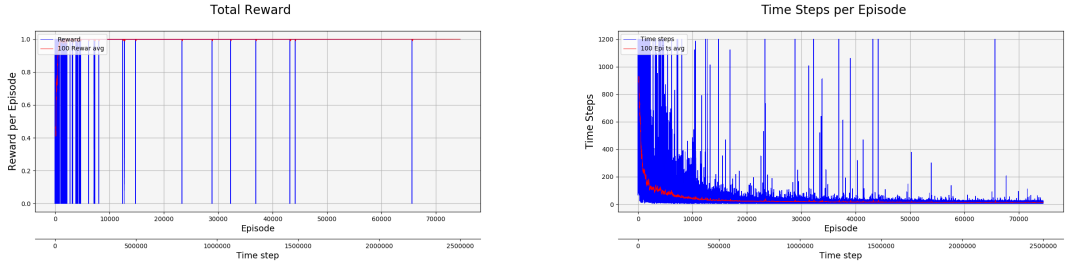


Figure 17: Level 2 Complexity. 2.5m time steps of training; 2 bin slots, 1 agent slot, 2 Items; $25 \cdot 3^{13} \approx 40$m environment states; $4 + 2 \cdot 2 \cdot 1$=8 actions.

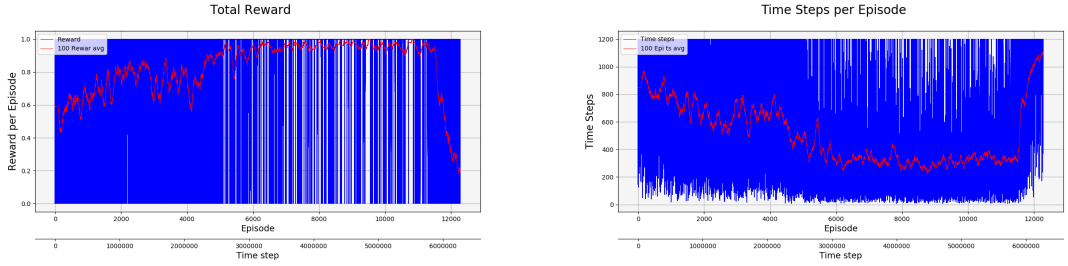


Figure 18: Level 3 Complexity. 7m time steps of training; 2 bin and agent slots, 2 Items; $25 \cdot 3^{14} \approx 120$m environment states; $4 + 2 \cdot 2 \cdot 2$=12 actions.

[1] Episode starts with a random transaction, bin status and agent position; episode finishes after the transaction completion or the time limit is reached.

[2] Reward obtained only after completing the current transaction.

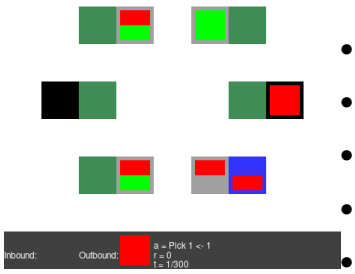
The most complex functional environment trained was the Level 2 (Figure 17):

- 2.5m time steps of training.
- 2 bin slots, 1 agent slot, 2 Items.
- $25 \cdot 3^{13} \approx 40$m environment states.
- $4 + 2 \cdot 2 \cdot 1 = 8$ actions.

As we can see in Figures 16 and 17, for complexity level 1 and 2 the RL model was able to learn the environment, since at the end of the training the agent was always able to get the reward in a small number of time steps. However, this was not the case for the complexity level 3. As we see in Figure 18, despite that the episode reward was moving towards 1 (and the episode time steps were decreasing), at the end of the training the total reward fell almost to 0 and the time steps reached almost the limit per episode. A similar behavior was observed while training smaller environments when the number of training time steps was not sufficiently large, which suggest that 2.5m training time steps were not enough for this environment. However, due to limited computational resources, it was not possible to train this model longer. Training environments more complex than Level 3 have a similar behavior to this. To make the training process more robust, a curiosity module was implemented. However, the observed results were very similar to the ones obtained with no curiosity, which reinforces our belief that this limitation may be caused by a lack of training.

4.10 Comparison with Heuristic baseline

Using a high-level movement environment (the agent can jump between the access spots) and two scenarios, we conducted several simulations to compare the RL model performance against the heuristic baseline. As a performance metric, we used the number of completed random transaction by each agent in 300 time steps. For each scenario, we performed 100 repetitions of the experiments. The follow general environment and model conditions were used:

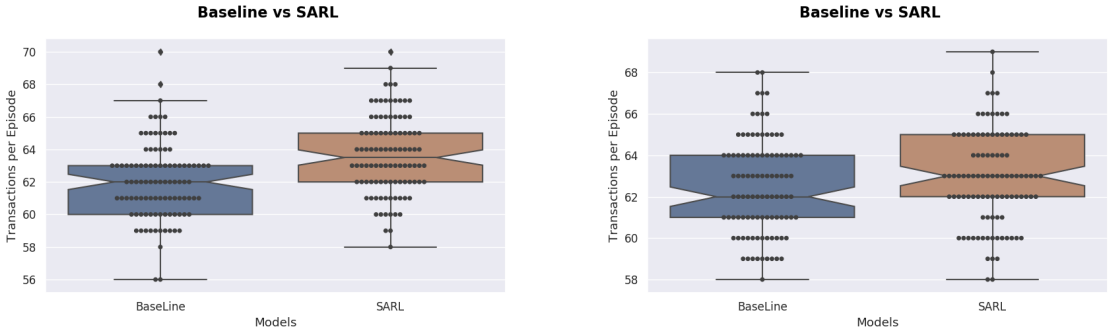


- Single agent with 2 slots.
- 4 bins with 2 slots each.
- Episodic transactions[1].
- 300 time steps limit.
- Sparse Rewards[2].

For the first scenario, we used 2 items and trained for 1.4 million time steps using a network architecture of [512, 128, 32] (see figure 19a). For the second scenario, we used 3 items and trained for 2.4 million time steps using a network architecture of [1024, 512, 128, 32] (see figure 19b).

[1] Episode starts with a random transaction, bin status and agent position; episode finishes after the transaction completion or the time limit is reached.

[2] Reward obtained only after completing the current transaction.



(a) First scenario: 2 items.

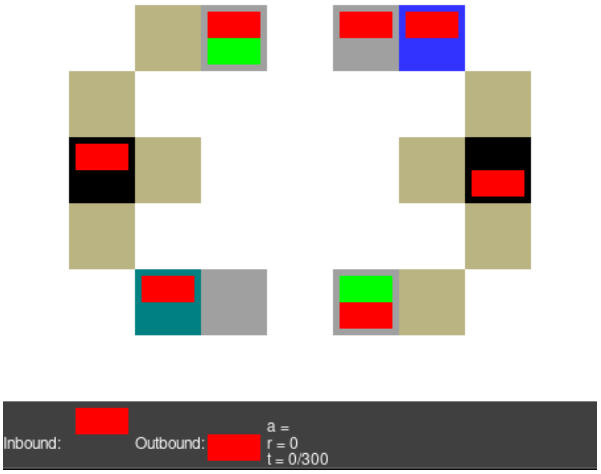
(b) Second scenario: 3 items.

Figure 19: Completed random transactions in 100 rounds of 300 ts.

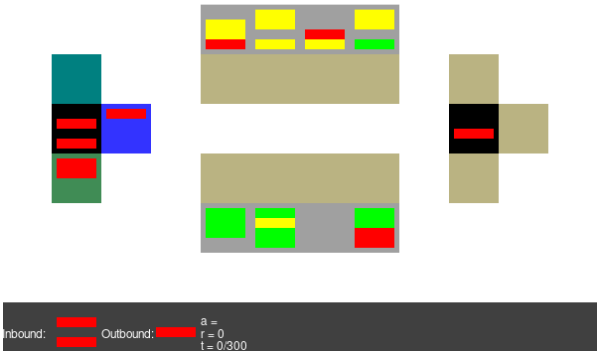
As we can see in figures 19, the RL approach performed better than the heuristic baseline in both cases. These results work as a proof of concept to show the viability of a RL model to manage the logistics and resources of a chaotic warehouse. This shows that with sufficient training time and/or computational power, and a big enough network architecture, it is possible to achieve good behavior in bigger and more complex environments.

4.11 Multi-Agent Results

For trying the models in a multi-agent environment, the simplest change was to simply add another agent to the "2 Slot 2 Item Case". As DQN usually takes longer to compute we decided to train our models first with A2C. Another reason for choosing A2C at the very beginning was that it can handle `gym.MultiDiscrete`-action spaces. In the implementation of `stable_baselines`, DQN can only handle `gym.Discrete`-action spaces, such that we had to implement a wrapper from MultiDiscrete to Discrete. All our A2C models where trained with the standard configuration of `stable_baselines` using 4 vectorized environments, while the DQN models used a 512x128x32 network with an exploration fraction of 0.8, decaying epsilon from 1 to 0.02.
We trained all our MARL models for 600,000 steps. While our Joint Action Learner (JAL) was able to move all agents within one step, our Independent Learner (IL) could only predict the action for one agent, thus it wasn't able to fulfill as many episodes as the JAL model within the given amount of training steps.



(a) **Medium** MARL setup (2 items, 4 bins, 2 bin slots, 2 agents, 2 agent slots



(b) **Large** MARL setup (3 items, 8 bins, 4 bin slots, 3 agents, 2 agent slots

Figure 20: Different MARL setups

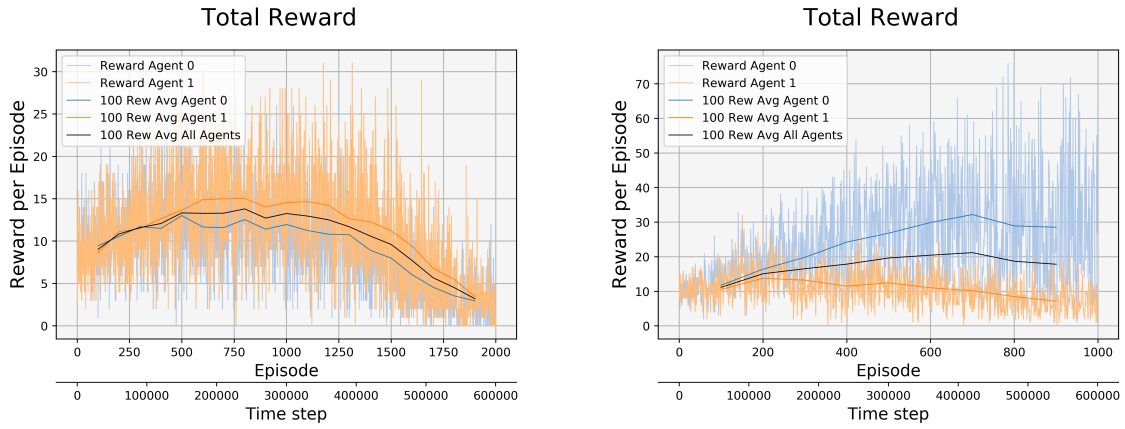
4.11.1 Episodic Transaction Scheme

The episodic transaction scheme differs a bit from the transaction scheme used in the single agent models. Here, an episode lasts for 300 time steps and an agent receives a reward of 1 as soon as it correctly puts an item to the staging out area. Also, transactions are generated after every time step with a probability of 50%.
In the **medium** (see Figure 20a) scenario, one can already see the challenges that one is facing with Multi-Agent Reinforcement Learning.
First we trained our models with A2C. As we can see in Figure 21, both learners, JAL and IL, improve their strategy over the training time with A2C towards fulfilling around 25 (JAL) resp. 20 (IL) transactions per episode. Trained using A2C, JAL slightly outperforms IL.

(a) JAL (b) IL

Figure 21: Models of medium warehouse environment trained with A2C

The results of the models trained with A2C differ highly from those trained with DQN. Looking at Figure 22, one can see that both models do not perform as expected when being trained with DQN. While in IL, there is still some improvement over time considering the mean of both agents, the rewards decrease in JAL after some improvement in the first 750 episodes.



(a) JAL (b) IL

Figure 22: Models of medium warehouse environment trained with DQN

The observations from above are verified by looking at Figure 23: While JAL can complete most transactions in the medium case when trained with A2C, it is not able to complete many transactions when trained with DQN. IL performs not as good as JAL trained with A2C but still is able to fulfill several transactions per episode, regardless of training algorithm.

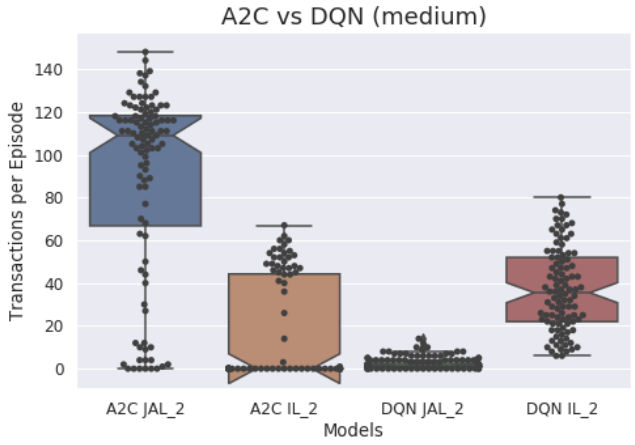
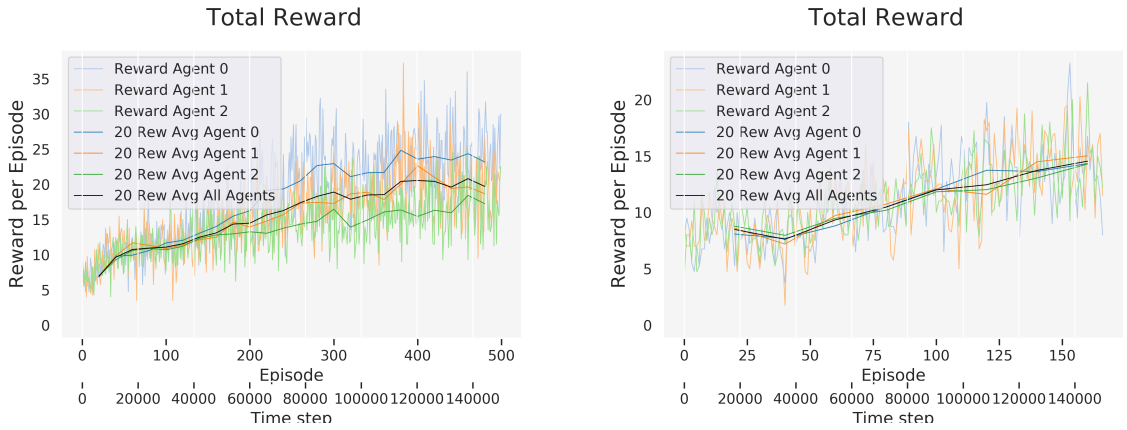


Figure 23: Completed random transactions in 100 rounds of 300 time steps.

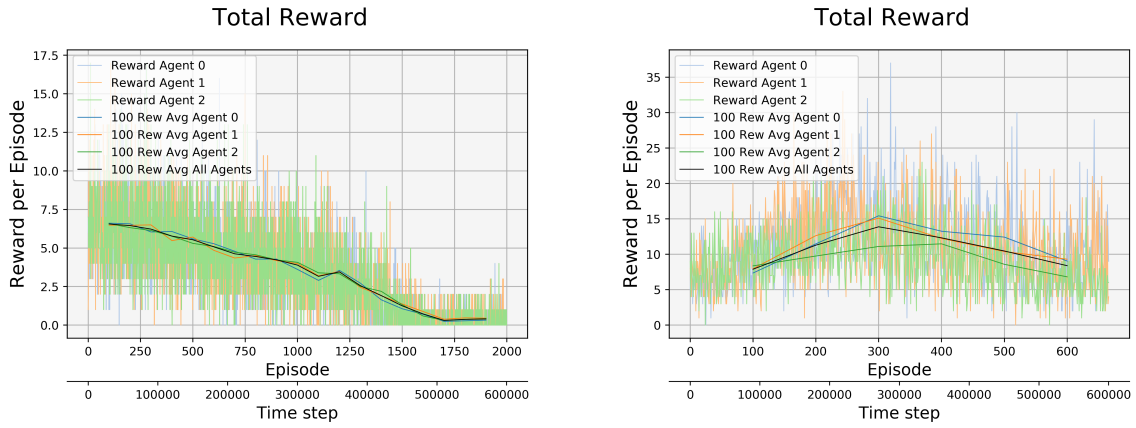
Switching to the **large** MARL scenario (see Figure 20b), we can observe that all models need more time to learn. Considering JAL and IL trained with A2C (see Figures 24), both models are behaving similar to the models in the **medium** scenario (see Figures 21). Both models learned to fulfill more transactions per episode, up to around 20 (JAL) compared to 15 (IL) transactions per episode.



(a) JAL (b) IL

Figure 24: Models of large warehouse environment trained with A2C

Similar to the **medium** case, the IL trained with DQN improves its policy, but after around 300 episodes, rewards start decreasing, whereas the JAL trained with DQN shows only decreasing rewards.



(a) JAL (b) IL

Figure 25: Models of large warehouse environment trained with DQN

Compared during testing, the models behave as in the **medium** case, but in a more extreme way. Also, while JAL outperforms IL when trained with A2C, IL outperforms JAL when trained with DQN.

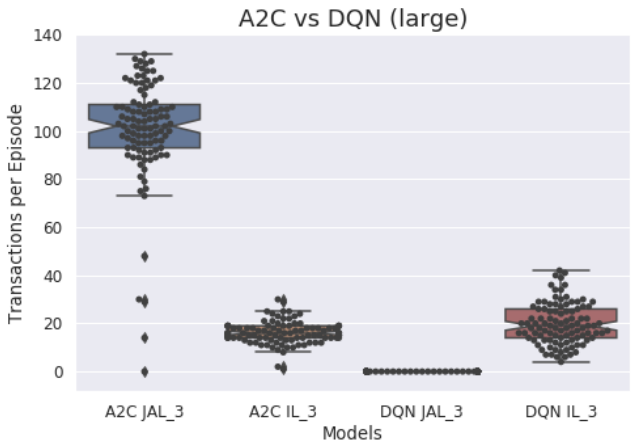


Figure 26: Completed random transactions in 100 rounds of 300 time steps.

4.11.2 Sequential Transaction Scheme

Additionally, we conducted experiments using the sequential transaction scheme described in Section 4.4. The simplest way to transition from the single-agent to the multi-agent setting in this case is to simply reuse trained single agents and run them in parallel. To be more precise, we first train a single agent in a warehouse. Then, when we want to run n agents in the same warehouse, we simply copy the agent n times and embed these copies into a coordinator module. This coordinator module has the purpose of

- converting observations of the multi-agent environment into observations that the single agents can deal with
- communicating these observations to the single agents and getting corresponding actions back
- synthesizing the individual actions of the single agents into an action vector to be passed to the environment.

The benefit of this method is that no new training is needed, and training time does not increase with the number of agents. However, there are significant drawbacks: The agents essentially think that they're alone in the environment, and thus they ignore each other completely. This already bounds the synergy

effects one can expect. Even worse, as the agents are not aware of each other, they might even obstruct each other, e.g. by blocking an access spot that another agent would like to get to. These considerations are confirmed by Figure 27 below: While still being able to complete a significant number of transactions, two agents coordinated in the way described above perform worse than just a single agent in the 2 slots, 2 items scenario.

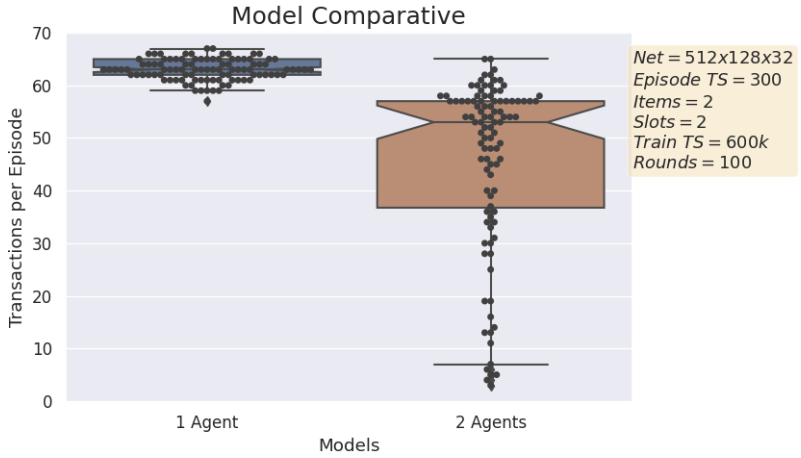


Figure 27: The single agent outperforms the coordinator module.

We also tried to adapt our IL and JAL approaches to the sequential transaction scheme, but this proved to be very difficult. Whereas the time required to finish an episode clearly diminishes over time for our single agent model, indicating that the agent learns to complete transactions faster and faster, we were not able to obtain similar results for multiple agents. Figure 28 illustrates this for both JAL and IL models trained using A2C.

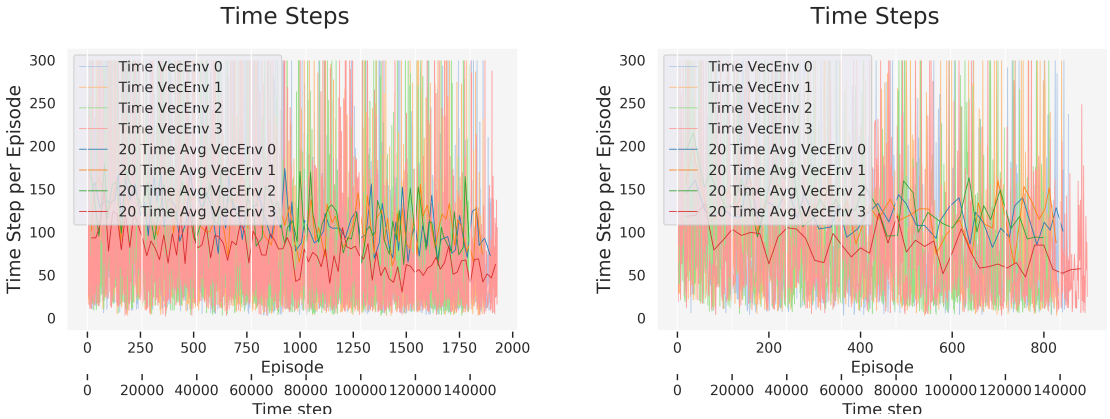


Figure 28: Both JAL (left) and IL (right) fail to significantly reduce the time needed to finish an episode.

This might be due to the possibility of the agents not getting meaningful rewards: As the agents only get a reward upon completion of an entire transaction, which will be triggered by a single agent putting the last inbound item into a bin or the last outbound item into the staging-out area, the majority of the agents will be rewarded for whatever action they did at the time of completion, which might not have been a good action at all. Further work is needed to devise a cleverer way of returning rewards to the agents.

5 Conclusions

Through this report we show that Reinforcement Learning is a promising approach and could applied to the Chaotic Warehouse scenario, albeit with some difficulties. We show that as complexity grows, the demand for resources in training grows exponentially and the key to solving this is to reduce it as much as possible.

We used a high level movement scheme in order to extract an easily solvable movement problem from the main problem that consists in the handling of transactions. Distances, however, can also be incorporated to the training so that the agent could still take them into account during its optimization.

We also used episodic transactions in order to reduce a large problem of consecutive transactions into a smaller problem of one single transaction. This way the agent can concentrate on solving a smaller setup. However, this might come with a negative consequence of finding a local minimum. This should be explored in further research.

We also introduced a curiosity module. However, we can only conclude that in this Chaotic Warehouse scenario, it might not be as useful in training. A possible reason could be that this scenario does not radically change as the agent progresses through it. In other environments where curiosity has been used, the environment changes a lot as the agent solves parts of it.

We compared the single agent case against a well thought out heuristic baseline and still are able to uncover some patterns that were not considered, enabling the Reinforcement Learning agent to outperform this baseline.

Finally, we expanded our approach to multiple agents. While some of our results are promising, we also encountered significant difficulties, which range from simple scaling issues to fundamental uncertainties about modeling the multi-agent reinforcement learning problem. Future work is needed in this area to obtain greater clarity and better results.

References

[1] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.

[2] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[3] Hongzi Mao et al. "Resource Management with Deep Reinforcement Learning". In: Nov. 2016, pp. 50–56. DOI: 10.1145/3005745.3005750.

[4] Deepak Pathak et al. *Curiosity-driven Exploration by Self-supervised Prediction*. 2017. arXiv: 1705.05363 [cs.LG].

[5] *Policy Gradient Algorithms*. https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#a2c. Accessed: 2020-07-22.

[6] *Qrash Course II: From Q-Learning to Gradient Policy Actor-Critic in 12 Minutes*. https://towardsdatascience.com/qrash-course-ii-from-q-learning-to-gradient-policy-actor-critic-in-12-minutes-8e8b47129c8c. Accessed: 2020-07-22.

[7] Pete Shinners. *PyGame*. http://pygame.org/. 2011.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.

[9] Ermo Wei and Sean Luke. "Lenient Learning in Independent-Learner Stochastic Cooperative Games". In: *Journal of Machine Learning Research* 17.84 (2016), pp. 1–42. URL: http://jmlr.org/papers/v17/15-417.html.

[10] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. "Multi-Agent Reinforcement Learning: A Selective Overview of Theories and Algorithms". In: *ArXiv* abs/1911.10635 (2019).

Appendix

JSON schema for warehouses

In order to store a warehouse in a file, one has to provide all the necessary information about it, which includes the parameters from table 1:

```
{
    "time_limit": 200,
    "num_rows": 7,
    "num_cols": 12,
    "num_bin_slots": 4,
    "num_agent_slots": 2,
    "num_items": 3,
    ...
}
```

Moreover, information about the warehouse objects has to be included. For instance, the agents are stored in a list, and each agent is described by its initial position and holding status:

```
{
    "agents": [
        {
            "position": [2, 4],
            "status": [0, 0]
        },
        ...
    ],
    ...
}
```

The bins and staging areas also have position and status, and additionally a list of access spots. Below, you can see how the staging areas are stored. The bins are stored analogously, but in a list, like the agents.

```
{
    "staging_in_area": {
        "position": [3, 1],
        "status": [2, 0, 0, 0],
        "access_spots": [
            [2, 1], [3, 2], [4, 1]
        ]
    },
    "staging_out_area": {
        ...
    },
    ...
}
```

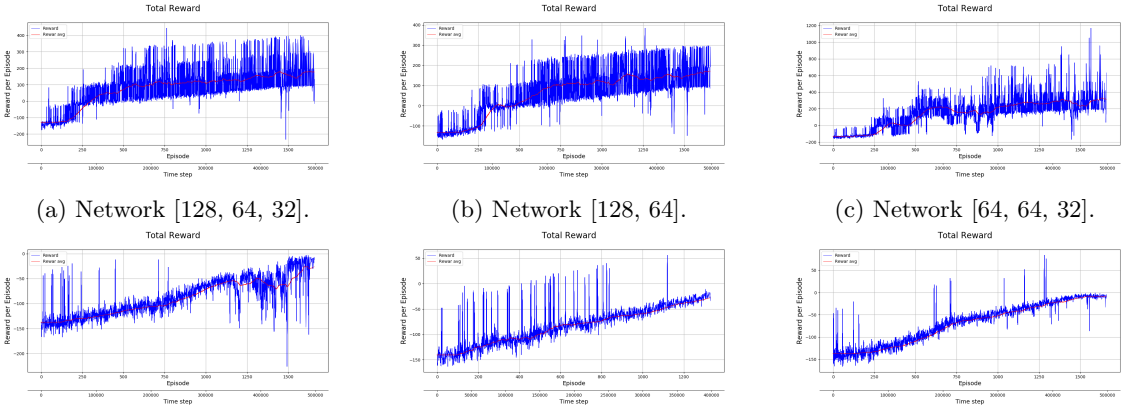
Other Results

Different Network sizes

In order to find a suitable Network size for our model, several architectures were tested. The general environment and model conditions were:

- Environment conditions: 7x9 grid, 4 bins, 1 slot (bin & agent) and 1 Item. Low level movement.
- Dense reward: Completed outbound transaction +100, Completed inbound transaction +10, Any kind of illegal interaction (move, drop, etc) -1
- Transaction scheme: Fixed episode duration, i.e. episode start always with same conditions, then new transactions were generated randomly until the end of the episode.

The reward plot for each tested architecture is shown in figure 29.



(a) Network [128, 64, 32]. (b) Network [128, 64]. (c) Network [64, 64, 32].

(d) Network [64, 32]. (e) Network [16, 16, 16, 16]. (f) Network [16, 8, 8].

Figure 29: Episode reward plots for different network architectures.

As we can see in the subfigures of Figure 29, a too small network architecture (subfigures d, e and f) perform poorly (the mean episode reward never surpass 0), while bigger networks performed better. However, note that the best results were not achieved by the biggest network (subfigure a, with mean episode reward close to 200), but by an intermediate one (subfigure c, with mean episode reward close to 300). A good explanation for this is the Variance-Bias trade-off, the simplest models have a smaller variance but perform poorly, while bigger models perform better but have bigger variance. Also, recall that bigger models mean more parameters to estimate, and therefore more training steps.