Forschungsprojekt B

Leon Rottler & Nicolas Mahn
University Furtwangen

Feburary 2024

Abstract

This paper presents a comprehensive study on the application of Deep Reinforcement Learning (DRL) techniques to the problem of Job Shop Scheduling (JSS). With a focus on evolving and refining DRL models, our research explores the development and assessment of various algorithms, including Deep Q-Networks (DQN), Double Deep Q-Networks (DDQN), Prioritized DDQN, and Advantage Actor-Critic (A2C) models, tailored to optimize scheduling processes under diverse constraints and operational requirements.

Central to our approach is the architecture of the RL_Resource_Manager application, a modular framework designed to facilitate the training, evaluation, and comparison of different DRL models across various JSS environments. This paper details the architectural components, algorithm implementations, and the methodologies employed for data generation and management, highlighting the adaptability of our approach to different scheduling scenarios.

We investigate several JSS environments, each characterized by unique attributes such as preemptiveness, task dependencies, processing times, and due dates, to evaluate the performance and applicability of our DRL models. Through experimentation, we analyze the effectiveness of each algorithm in navigating the complexities of job scheduling, considering factors like computational efficiency, reward structuring, and learning stability. Overall, the results of the various algorithms and environments were somewhat mixed, only performing in one environment to some extent satisfactorily.

Contents

1 Introduction

This paper presents the results of our second research project focusing on Deep Reinforcement Learning. The results of our first project can be found in Rottler & Mahn (2023). In this paper, Job Shop Scheduling (JSS) was first examined to gain a basic understanding of it. Subsequently, the fundamentals of Reinforcement Learning were explained, and some examples with Q-Learning were considered. A significant part of the paper from the first project also deals with Deep Reinforcement Learning and the implementation of Deep Q-Networks. The following subsections briefly describes the key contents and results from our first project.

1.1 Job shop scheduling

In our previous paper, we explained the basics of Job Shop Scheduling (JSS) and the challenges involved. Gabel and Riedmiller provide a straightforward explanation of Job-Shop Scheduling in their paper *Adaptive reactive Job-Shop Scheduling with Reinforcement Learning Agents*:

> "The goal of scheduling is to allocate a specified number of jobs (also called tasks) to a limited number resources (also called machines) in such a manner that some specific objective is optimized." (Gabel & Riedmiller, 2008, pg. 4)

The NP-hard complexity surrounding the topic of Job Shop Scheduling Problem (JSSP) is further explained, along with a brief, simple example of a JSSP. Additionally, it examines the information required in a JSSP about the jobs and machines. This primarily focuses on the processing route, processing time, due dates, and priority. Subsequently, it is demonstrated that there is a multitude of possible optimization criteria, with some being discussed as examples. In particular, optimization criteria for due dates are considered, for example *lateness of job*, as these are crucial for achieving customer satisfaction according to Framinan et al. (2014). Moreover, popular scheduling heuristics for JSS, such as EDD, were addressed, and towards the end of the topic, a brief mention was made of various benchmarks.

1.2 Reinforcement Learning

Another aspect of our previous work was Reinforcement Learning (RL). The paper explained, what RL is and gave an understanding of the fundamental elements of RL, including Agent, Environment, State, Action and Reward. Based on the fundamentals, the Q-learning algorithm was explained and a simple JSS application using a Q-learning algorithm was described. The description included an explanation of the State, Action and Reward representation. Building on this, a Q-table was theoretically calculated as an example, and then it was at length demonstrated how the Q-learning algorithm was implemented with Python. With this implementation, however, truly satisfactory results were not achieved. For this reason, a further approach was attempted, using Monte Carlo Learning to achieve better results, which was essentially successful. The implementation in Python was also described in detail for this approach.

1.3 DQN

A key aspect of the previous work was the engagement with the topic of Deep Reinforcement Learning. It was outlined that that traditional Q-learning is effective in many RL problems, but it is confronted limitations when handling high-dimensional state-action space characteristics of complex problems such as JSS. Deep Q-Networks (DQN) were therefore examined in detail. They provide an advanced algorithmic approach that merges Q-learning principles with deep neural networks. After delving into the theoretical foundations, it is described in great detail how and with which components the DQN model was implemented in Python. The paper explains, how the DQN model was build using different keras layers and why a Replay Buffer is used. Afterwards it is described, how the Q-learning function is implemented, considering training loops, experience storage, model update, target network updates and epsilon-greedy policy.

The first environment that was developed is the *TimeManagement Environment*, which focused on the optimization of jobs according to their due date. In this paper, the naming of environments utilizes the classification of machine scheduling models, therefore the TimeManagement environment can be found in this paper under the notation $[J, m = 1|nowait, f_j, g_j = 1|T]$. To better understand the environment itself and the implementation in Python, both were explained in the paper. Particular attention was paid to aspects such as the initialization of the environment, getting the start state, getting the next state, the DQN state and getting the rewards.

Furthermore, a second environment named *ResourceManagement* was developed. The naming of this environment has also been adapted in the context of this paper and can be found under the notation $[J|nowait, t_j, g_j = 1|D]$. The goal of this environment was to find the best possible assignment of tasks (with different execution times and no depended order) to machines such that the overall execution time is minimized. The implementation of this environment was also described following a similar scheme as the previous environment.

1.4 Results

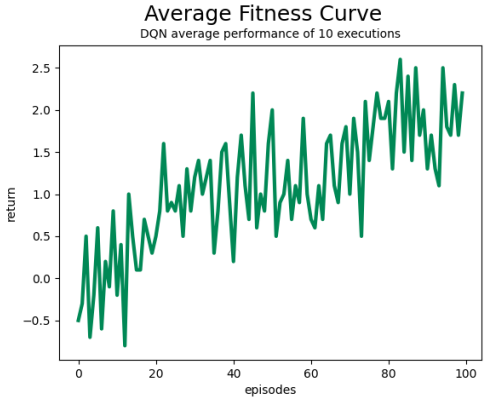

Figure 1: Fitness Curve TimeManagement Environment

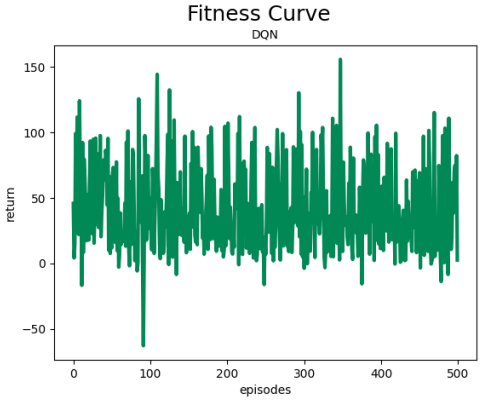

Figure 2: Fitness Curve ResourceManagement Environment

As shown in Figure 1, it was possible to get some promising results from the DQN TimeManagement environment. The algorithms ability to effectively learn the optimal scheduling policy was evaluated through its task arrangement accuracy, with an average accuracy rate of approximately 69.33%. For further details on the results of this

environment, have a look in chapter 5.4.8 of our first paper.

Figure 2 shows the fitness curve for the DQN ResourceManagement environment and as can be seen, the results are rather sub-optimal not showing significant improvements over time. To evaluate the efficacy of the algorithm, the Root Mean Squared Error metric was used. It indicated that the model has learned to schedule tasks on machines to some extent but with substantial room for improvement.

1.5 Content of this paper

The research in the second project, the results of which are described in this paper, focused on the topic of Deep Q-Networks (DQN). In the course of this, the existing DQN algorithm from the previous project was also adapted and further developed. Moreover, as part of this project, additional algorithms for solving the Job Shop Scheduling Problem were implemented, such as Double DQN and Prioritized DQN. The algorithms are described in detail in chapter 5. Furthermore, the naming of the environments was adapted to the notation for the classification of machine scheduling models. Detailed information on this will follow in the next chapter (2). Related papers were also considered as part of this project. In addition, chapter 3 also provides an overview of publicly available GitHub repositories that deal with JSS. Due to the increasing demands on the data, the management and generation of new datasets was dealt with in a more focused manner and is considered in chapter 6. As the project has progressed, the environments have also changed and new ones have been added, which is why chapter 7 deals with these in detail. The results of the respective environments are also presented in this chapter.

2 Classification of Machine Scheduling Models

There are many different problems of optimal job scheduling that vary in the nature of the jobs, the characteristics of the machines, the constraints on the schedule, and the objective function. We have already addressed this in our previous work. One of the most well-known notations to describe job scheduling problems was introduced by Graham, Lawler, Lenstra and Rinnooy Kan. A good description of the notation can be found in the publication *Sequencing and Scheduling: Algorithms and Complexity* by Lawler, Lenstra, Rinnooy Kan, and Shmoys. Lawler et al. (1993) Below, we briefly explain the core concept of the notation. For more detailed information, please refer to Chapter 3 of the publication by Lawler et al. (1993).

The basic idea of the notation is a three-field classification to describe the scheduling problem.

- The first field is denoted by α and describes the machine environment.
- The second field is denoted by β and describes the job characteristics.
- The third field is denoted by γ and describes the optimization criterion.

Thus, the notation follows the following scheme: $\alpha|\beta|\gamma$

2.1 Machine environment | α

In this subsection, the machine environment is examined in more detail. Lawler et al. differentiate between a single-stage and a multi-stage job scheduling problem. For the single-stage job scheduling problem, Lawler et al. define four values (classes):

- $\alpha_1 = \cdot$: There is a single machine.
- $\alpha_1 = \boldsymbol{P}$: There are m parallel, identical machines.
- $\alpha_1 = \boldsymbol{Q}$: There are m parallel machines with different given speeds.
- $\alpha_1 = \boldsymbol{R}$: There are m parallel unrelated machines.

For multi-stage job scheduling problems, Lawler et al. define three different classes:

- $\alpha_1 = \boldsymbol{O}$: Open-shop - every job consists of a set of operations which can be scheduled in any order.
- $\alpha_1 = \boldsymbol{F}$: Flow-shop - each job consists of a chain of operations which have to be scheduled in the given order.
- $\alpha_1 = \boldsymbol{J}$: Job-shop - each job consists of a chain of operations which have to be scheduled in the given order and processed on a dedicated machine.

In addition, the machine environment is also described with a α_2 value, which indicates the number of machines existing in the environment. For example, $\alpha_1 = \text{J}$ and $\alpha_2 = 3$ would result in $\alpha = \text{J3}$, meaning that it is a job-shop environment with three machines.

2.2 Job characteristics | β

Before describing the different job characteristics, Lawler et al. specify the following data for each job:

- $\boldsymbol{m_j}$: number of operations
- $\boldsymbol{p}$: processing requirements (processing time)
- $\boldsymbol{r_j}$: release date
- $\boldsymbol{f_j}$: nondecreasing real cost function
- $\boldsymbol{d_j}$: due date
- $\boldsymbol{w_j}$: weight

Typically, m_j, p, r_j, f_j, d_j and w_j are represented by integral values.

Depending on the problem, the characteristics of jobs in a JSSP can vary greatly, for example, the duration of a processing step on a specific machine. For a uniform notation of these characteristics, Lawler et al. have considered the following areas:

- β_1 = Preemption:
  - $\beta_1 = \boldsymbol{pmtn}$: Preemption - the execution of any operation can be paused and continued at a later moment.
  - $\beta_1 = \cdot$: Preemptive actions are not permitted.
- β_2 = Precedence relation:
  - $\beta_2 = \boldsymbol{prec}$: Precedence relation exists between the jobs. When there is a directed graph path from job j to k, job j precedes job k and must be completed, before job k begins.
  - $\beta_2 = \boldsymbol{tree}$: represents a rooted tree where each vertex has an outdegree or indegree of no more than one.
  - $\beta_2 = \cdot$: A precedence relation has not been defined.
- $\beta_3 = \boldsymbol{r}$: Release dates:
  - $\beta_3 = \boldsymbol{r_j}$: Release dates: Specified release dates that can vary for each job.
  - $\beta_3 = \cdot$: All $r_j = 0$
- $\beta_4 = \boldsymbol{p}$: Unit processing requirement:
  - $\beta_4 = \boldsymbol{p} = 1$: each job or operation has a unit processing requirement of 1, taking one time unit to complete.
  - $\beta_4 = \cdot$: jobs or operations have varying lengths of time to complete (arbitrary nonnegative integers).

There are other well-known ways to describe job characteristics and β, an example of which would be the notation template known in German-speaking areas by Domschke (1997). Essentially, these notations are very similar and use the same basic concepts.

2.3 Optimization criterion | γ

Typically, the aim is to reduce a specific objective value to its minimum. As an example, Lawler et al. cite the following optimization criteria:

- $\boldsymbol{C_j}$: the completion time (C_{max} = makespan)
- $\boldsymbol{L_j}$: the lateness ($C_j - d_j$)
- $\boldsymbol{T_j}$: the tardiness
- $\boldsymbol{U_j}$: the unit penalty (throughput) (in the German notation after Domschke (1997) as D_j for Durchlaufzeit)

2.4 Example

In this subsection, the notation is explained using a simple example. For this purpose, the following JSSP in the corresponding notation:

$$[\alpha|\beta|\gamma]$$

$$[\text{ J1 } | \text{ pmtn, prec } | \text{ U }]$$

This example describes a job-shop environment with one machine (α). These jobs can be preempted and resumed at a later time, plus there is a directed dependency between the jobs (β). The problem is to be optimized according to the throughput (γ).

2.5 Notation used in this paper

In this paper, the notation already described is essentially used. Due to the fact that this paper was written by native German speakers, we initially started with the notation of Domschke (1997). Essentially, the notations differ in that the symbols used can vary depending on the language, and that with Domschke (1997), β is sometimes more extensive. The main difference or additions for this paper are listed below:

- β_1 = Preemption: Domschke (1997) adds an additional option to the β_1 characteristics. The option ***nowait*** describes, that after the execution of an operation, the next operation step must begin immediately.
- $\boldsymbol{f_j}$: Domschke (1997) adds an additional job characteristic to β which describes that for each job, mandatory due dates are specified.
- $\boldsymbol{g_j}$: Domschke (1997) adds an additional job characteristic to β which describes for each job, how many operations exist for that job. For example, $g_j = 1$ would mean, that each job has exactly one operation.
- $\boldsymbol{n_j}$: Domschke (1997) adds an additional characteristic to β_3. n_j describes the lead times that are given after the completion of a work process. The job or operation must wait for a certain amount of time before it can continue to be processed again.
- $\boldsymbol{t_j}$: Domschke (1997) uses the symbol t_j for the processing time p.
- ***[α_1,m=1]***: Domschke (1997) uses a slightly different method to notate α_1 and α_2. For α_2 the notation is *m=number of machines*, for example [J,m=1] would mean a Jop shop environment with one machine.

3 Related Work

Research on JSSP has been ongoing for many years and decades and in the recent years, especially the combination of JSSP and Deep Reinforcement Learning. Therefore, there are numerous related works in this field, some of which will be briefly explained. Furthermore, there are some published GitHub repositories where code is published that deals with JSSP and Deep RL.

3.1 Related papers

Gabel & Riedmiller The article proposes an approach to production scheduling problems using multi-agent reinforcement learning. Traditional methods rely on centralized solutions and full problem knowledge, leading to computational challenges. In contrast, the proposed approach employs independent adaptive agents associated with resources, making real-time job dispatching decisions using reinforcement learning.

The system architecture facilitates reactive scheduling, allowing agents to adapt to changing environments without complete re-planning. The multi-agent learning algorithm combines batch-mode reinforcement learning, neural network-based value function approximation, and optimistic inter-agent coordination.

The evaluation demonstrates that the approach competes well with alternative methods on benchmark problems. Overall, the multi-agent reinforcement learning approach offers a promising solution for efficient and adaptive production scheduling with near-optimal results. (Gabel & Riedmiller, 2008)

Riedmiller & Riedmiller The paper introduces a neural network-based local learning approach for job shop scheduling problems. It aims to combine computational solutions with heuristic dispatching rules to achieve nearly optimal results with low computational effort. The learning agents autonomously optimize local dispatching policies based on a global optimization goal.

Experiments on single and multi-resource cases demonstrate the learning system's ability to outperform heuristic dispatching rules and adapt to new situations. The learning system's generalization ability allows it to apply knowledge to unseen scenarios without retraining. Overall, the approach provides an efficient solution for reactive scheduling and easy reconfiguration in response to changes. (Riedmiller & Riedmiller, 1999)

Zhang et al. The paper "Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning" presented at NeurIPS 2020, introduces a novel approach for solving the Job Shop Scheduling Problem, a complex optimization task in manufacturing and production planning. The authors propose a deep reinforcement learning (DRL) framework that generates Priority Dispatching Rules dynamically. This framework employs a Graph Neural Network to capture the structure of JSSPs, enabling it to learn dispatching decisions that effectively minimize scheduling time. The DRL model learns from experience, improving its scheduling policies as it encounters more scenarios, which allows it to outperform traditional, manually-crafted PDRs in both effectiveness and efficiency. The significance of this work lies in its ability to generalize across different JSSP instances, demonstrating potential for

real-world applications where scheduling plays a critical role. The results show that this DRL-based approach can adapt to various job shop sizes and complexities, offering a scalable and flexible solution to scheduling challenges. Zhang et al. (2020)

Tassel, Gebser & Schekotihin The article presents an Deep Reinforcement Learning (DRL) approach for JSS with industrial constraints. The authors introduce an optimized JSS environment and a priority-based policy architecture to guide the agent's learning process. The reward function is designed to minimize the scheduled area and maximize machine usage. The DRL-based method outperforms traditional dispatching rules on all instances. It also shows significant improvement over previous RL-based approaches. (Tassel et al., 2021)

The mentioned works were able to demonstrate that it is highly feasible to solve JSS problems of varying complexity using various Reinforcement Learning methods and Neural Networks. However, a problem with the examined related works lies in the description of States, Actions, and Rewards and the concrete implementation of the approaches in working code.

3.2 Related GitHub repositories

In the context of this work, a look was also taken at GitHub repositories that deal with a similar problem, but in part pursue different approaches to solving the JSSP.

RL-Modelling by MyMirelHub The GitHub repository *RL-Modelling* by *MyMirelHub* is dedicated to a Reinforcement Task Scheduling Project. It explores the application of reinforcement learning to task scheduling, specifically focusing on optimizing CPU and memory allocation for a set of tasks. The project uses Keras and Python to model this problem, aiming to schedule tasks based on their resource requirements efficiently. The approach involves creating an environment and an agent that interacts with it to learn optimal scheduling strategies through reinforcement learning techniques. `https://github.com/MyMirelHub/RL-Modelling`

L2D by zcaicaros The GitHub repository *L2D* is the official PyTorch implementation of the paper *Learning to Dispatch for Job Shop Scheduling via Deep Reinforcement Learning* by Zhang et al. (2020) described in the previous section. `https://github.com/zcaicaros/L2D`

wheatley by jolibrain The *wheatley* repository on GitHub by *jolibrain* is a problem solver for scheduling tasks, utilizing Graph Neural Networks (GNNs) and Reinforcement Learning. It's designed to address job-shop scheduling problems, with a focus on real-world industrial applications. The project supports training for both fixed and uncertain problem conditions and aims to generalize across various scheduling challenges. It incorporates advanced deep learning libraries, such as PyTorch and DGL, and offers features like web-based training metrics visualization and comparison to OR-Tools.

RL-Job-Shop-Scheduling by prosysscience The GitHub repository *RL-Job-Shop-Scheduling* is the official PyTorch implementation of the paper *A Reinforcement Learning Environment For Job-Shop Scheduling* by Tassel et al. (2021) described in the previous section. `https://github.com/prosysscience/RL-Job-Shop-Scheduling`

Particularly, the implementation of RL-Job-Shop-Scheduling repository was examined more closely. The idea was to look at the basic concepts of the implementation and compare it to our approach. However, the approach chosen for the implementation in the repositories differs quiet a bit from our approach. The developers decided to include many libraries and use the functions provided by them. For example, the *wandb* package was used for evaluation and monitoring. In addition, a large part of the RL implementation was done by using the open-source library *RLlib*. That said, it proved to be a great challenge to obtain further details about the corresponding functionality of the libraries used, since a version of the library that is several years old was used, for which there is hardly any documentation available today.

4 Architecture

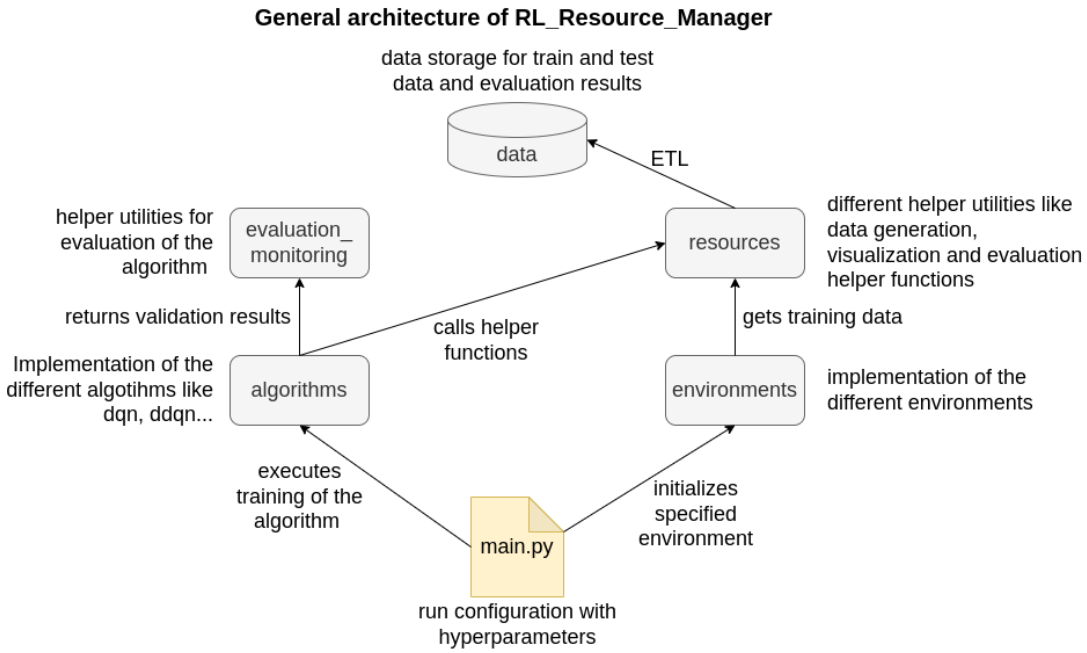

Figure 3: General architecture of RL_Resource_Manager

The basic architecture of the developed application RL_Resource_Manager can be seen in Figure 3. The figure shows the essential components of the architecture with a brief description and the main dependencies. The gray rectangles each symbolize a directory, the yellow file symbolizes a single Python file. The following short descriptions will provide more detailed information on the individual components.

main.py The core of the architecture is the Python file *main.py*. In this file, the entire configuration for the training of the models takes place. Here, the algorithm, the environment and the training and test data can be set. In addition, many other parameters such as the number of episodes, epsilon, epsilon_decay, batch_size, and many more can be adjusted. Moreover, the file orchestrates the initialization of the environment, the training and subsequent evaluation and, if desired, storage of the model.

algorithms In the *Algorithms* component, the various implemented algorithms are located. Each algorithm is implemented in its own Python file. Furthermore, within this component, there are Python files with functions that are used by all algorithms. These include the creation of the Keras models and the implementation of the Replay_Buffer. The specific implementation of the various algorithms is explained in chapter 5.

environments The implemented environments can be found in the *Environments* component. As with the algorithms, the environments are each implemented in their own Python file. In addition, there is a generic environment where functions are outsourced that are needed by all environments and are then called by them. The specific implementation of the various environments is explained in chapter 7.

evaluation_monitoring After the code was refactored to some extent, there is only a little functionality left in the component. Essentially, it contains the Python script *validation.py*, which is required to calculate the loss and accuracy for some algorithms.

resources A lot of utility functionality can be found in the *resources* component. Among other things, this component contains the Python scripts for data generation. Further information on the generation of the data is provided in the following chapter. Also, the Python scripts with functions for evaluating the algorithms are located in this component and are described in more detail in chapter 6.6. In addition, the component provides functions such as monitoring the resources used during the training (CPU and GPU) and general utility functions, which have been outsourced to a *utils.py* file.

data The *data* component, as indicated by the symbol, represents the storage location for the application's data. This component is kept very simple; it is not a database or similar. The data is stored in files in various directories. On one hand, the training and test data are saved, and on the other hand, the results of the evaluations are also stored. Therefore, the data directory contains three sub-directories: test, train, and evaluation.

5 Algorithms

To solve these JSS Problems several Algorithms where designed. The first one is the Depp Q-Network (DQN) algorithm, which had already been designed in our previous paper but has undergone some changes. To improve on the DQN a Double DQN (DDQN), and a Prioritized DDQN was designed. A different approach to DQN was also attempted with an Advantage Actor-Critic (A2C) algorithm. An A2C algorithm is gradient-based where the algorithm learns from both the policy and the value function. In comparison Q-Learning is a gradient-free method, that strongly focuses on the q-value. On top of this a Supervised approach was also conceived, so that the accuracy can be calculated and compared between reinforcement algorithms. Though so far, the correct answers can only be determined for the $[J, m = 1|nowait, f_j, g_j = 1|T]$ algorithm.

5.1 Deep Q-Network (DQN)

The DQN algorithm, previously introduced in our work, is revisited here due to several important modifications and foundational contributions to subsequent algorithms. Our implementation defines the DQN model as a sequential neural network, beginning with an input layer reflecting the environment's state dimensions, followed by two dense layers of 64 and 32 neurons, respectively, employing ReLU activation for non-linearity. A flatten layer then converts the multidimensional data into a one-dimensional action vector, with the output layer's size matching the number of possible actions. We adopt Mean Squared Error (MSE) for loss calculation, using the Adam optimizer, a common choice in RL for its efficiency in handling sparse gradients and adaptive learning rates. The learning rate is adjustable via the α hyperparameter, allowing for fine-tuning to specific environments.

```
def create_dqn_model(input_shape, num_actions, alpha):
    model = Sequential([
        Input(shape=input_shape),
        Dense(64, activation="relu"),
        Dense(32, activation="relu"),
        Flatten(),
        Dense(num_actions, activation="linear")
    ])
    model.compile(loss='mse', optimizer=Adam(learning_rate=alpha),
                                        metrics=['accuracy'])
    return model
```

After the DQN model is initialized, it is duplicated to create the target DQN model. This target model plays a crucial role in stabilizing the learning process by providing a consistent benchmark for updating the DQN model's weights. It will be discussed in more detail later in this chapter.

```
# Initialize DQN and target models
dqn_model = create_dqn_model(env.dimensions, len(env.actions), alpha)
target_dqn_model = keras.models.clone_model(dqn_model)
target_dqn_model.set_weights(dqn_model.get_weights())

# Create Replay Buffer
replay_buffer = ReplayBuffer(10000)
```

In addition to the target model, the Replay Buffer is initialized. The Replay Buffer is a crucial component in reinforcement learning that stores experiences collected during training. Each experience, or transition, consists of a tuple containing the state, action

taken, reward received, and the next state. This buffer allows the algorithm to learn from a wider range of experiences by replaying these transitions in mini-batches, thereby breaking the correlation between consecutive learning steps and improving the stability and efficiency of the learning process.

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.buffer, batch_size)

    def __len__(self):
        return len(self.buffer)
```

The episode begins by obtaining the initial state from the environment. This marks the start of the episode's iterations, where the algorithm endeavours to navigate the environment's challenges, aiming to maximize the reward received.

```
# Main training loop
for episode in range(episodes):
    state = env.get_start_state(episode)

    # If not final state
    while not env.done(state):
```

Following the initialization, an action is selected for the agent to take in the environment. This selection process utilizes an epsilon-greedy policy, balancing exploration and exploitation to optimize the agent's learning.

```
        # Action selection and masking
        possible_actions, impossible_actions = \
            env.get_possible_actions(state, index=True)

        # Epsilon-greedy policy
        if rnd() < epsilon:
            action_index = \
                possible_actions[randint(0, len(possible_actions))]
        else:
            q_values = \
                dqn_model.predict(np.array([state]))[0]
            if len(impossible_actions) > 0:
                q_values[impossible_actions] = -1e6  # Mask with a large
                                                     negative value
            action_index = util.argmax(q_values)
        action = env.actions[action_index]
```

In this process, the agent first identifies possible and impossible actions given the current state. With a probability of epsilon, a hyperparameter, the agent explores by selecting a random action from the set of possible actions, encouraging the discovery

of new strategies. Conversely, with a probability of 1 - epsilon, the agent exploits its current knowledge by selecting the action with the highest estimated reward (Q-value), as predicted by the DQN model. To ensure the agent only selects feasible actions, Q-values of impossible actions are masked with a significantly large negative value, effectively removing them from consideration. This method ensures that the agent's decisions are both informed by past experiences and open to new possibilities, fostering a balance between exploring new actions and exploiting known rewards.

```
        # Take action, observe reward and next state
        next_state = env.get_next_state(state, action)
        reward = env.get_reward(state, action, next_state)

        # Store experience to the replay buffer
        replay_buffer.push(state, action_index, reward, next_state)
```

Once an action is selected, the environment calculates the corresponding reward. This calculation is based on how well the action aligns with the goals of the task, reflecting the immediate benefit or cost of that action. The reward calculation is a critical component that influences the learning process. How the reward functions where implemented for each environment can be found in the corresponding chapter about them.

Then, the current state, the action taken, the received reward, and the subsequent state are collectively recorded as an experience and stored in the replay buffer.

```
        # Start training when there are enough experiences in the buffer
        if len(replay_buffer) > batch_size:
            batch = replay_buffer.sample(batch_size)

            states = np.array([b_state for b_state, _, _, _ in batch])
            next_states = np.array([b_next_state for _, _, _,
                                                    b_next_state in batch])

            q_values_batch = target_dqn_model.predict(states)
            next_q_values_batch = target_dqn_model.predict(next_states)

            for i, (b_state, b_action, b_reward, b_next_state) in
                                                    enumerate(batch):
                q_values = q_values_batch[i]
                next_q_values = next_q_values_batch[i]

                # Calculate the updated Q-value for the taken action
                q_value = q_values[b_action]
                q_value = (b_reward + gamma * np.max(next_q_values)) -
                                                        q_value
                q_values[b_action] = q_value

                dqn_input[i] = np.array(state)
                dqn_target[i] = q_values

            dqn_model.fit(np.array(dqn_input), np.array(dqn_target),
                          use_multiprocessing=True,
                          batch_size=batch_size)
```

This code segment highlights the process of training the DQN model using experiences stored in a replay buffer, updating the model with a target network, and employing vectorization for efficient computation.

The use of a target network is a critical innovation in stabilizing the training of DQN

models. Normally, in Q-learning, an agent learns by continually updating its Q-values based on the reward received from the environment and the estimated future rewards. However, using the same network for both current and next state Q-value estimation can lead to unstable training due to the moving target problem, where the Q-values keep changing as the network weights are updated. To mitigate this, the target network concept was introduced. The target network is a separate model with the same architecture as the primary DQN model but with its weights frozen on a previous version of the primary model. By using the target network to predict the next state Q-values, the training process becomes more stable because the target Q-values are kept constant for a number of steps, reducing the variance in the updates.

Vectorization is another important concept illustrated in this code. It refers to the process of converting operations that would normally be executed in a loop into operations that can be performed on entire arrays or matrices at once. This is achieved through the use of libraries like NumPy, which is designed for efficient mathematical operations on large arrays of data. In the context of this code, vectorization is used when the states and next states are converted into NumPy arrays and when Q-values are calculated for batches of experiences. This approach significantly speeds up the computation, as operations on large arrays can be parallelized and executed much faster than iterating through each item individually.

In summary, the code segment is a part of a DQN training loop where experiences stored in a replay buffer are used to update the model. The target network is employed to provide stable Q-value targets for learning, preventing the destabilizing effects of rapidly changing estimates. Vectorization is utilised to efficiently process batches of experiences, enhancing the computational efficiency of the training process. These techniques combined allow for the effective training of a DQN model, enabling it to learn optimal policies for decision-making tasks in complex environments.

```
        # Update state
        state = next_state.copy()
```

Finally the state is updated and with that the next episode can start.

```
    # Target network update
    if episode % update_target_network == 0:
        target_dqn_model.set_weights(dqn_model.get_weights())

    # Epsilon decay
    epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

At the conclusion of each episode, the target network is updated at intervals defined by the hyperparameter update_target_network_times. This ensures that the target network gradually aligns with the learned policy. Subsequently, the exploration rate epsilon is decayed according to a predetermined schedule, which strategically reduces the amount of random exploration as the agent becomes more knowledgeable about the environment. This cycle marks the transition to the next episode, allowing the learning process to proceed with updated parameters. This Sequence diagram shown in Figure 4 highlights, how the algorithm learns.

To demonstrate the effectiveness of the algorithm it was executed in combination with the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment. This resulted in an accuracy (it is basically impossible to get 100% accuracy) of 2.20% and an MSE of 7.19.

The diagram in Figure 5 clearly shows that the algorithm, hasn't finished learning (the regressed curve is probably deceptive at the end). The problem with this model is, that

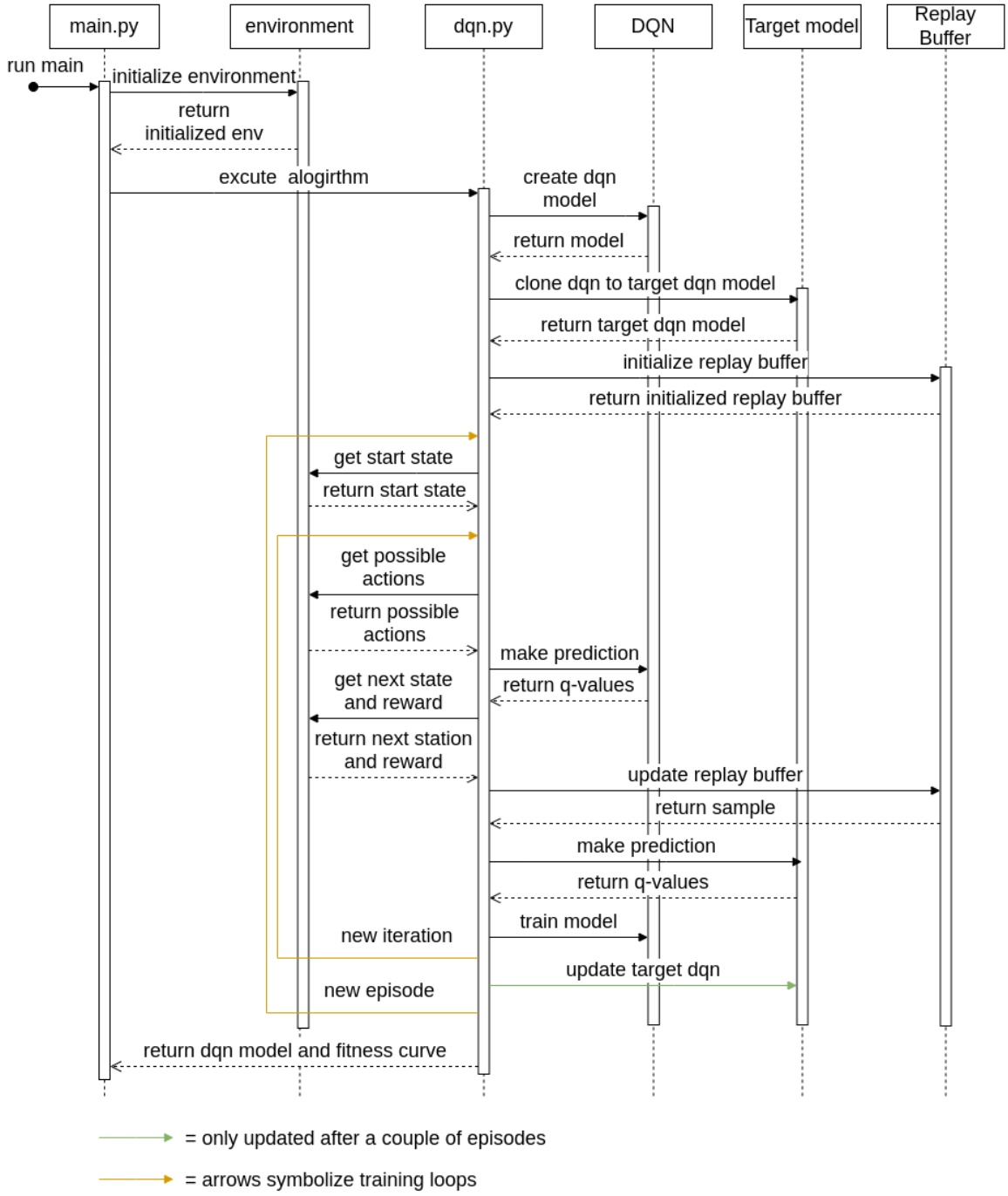

Figure 4: Sequence diagram DQN algorithm

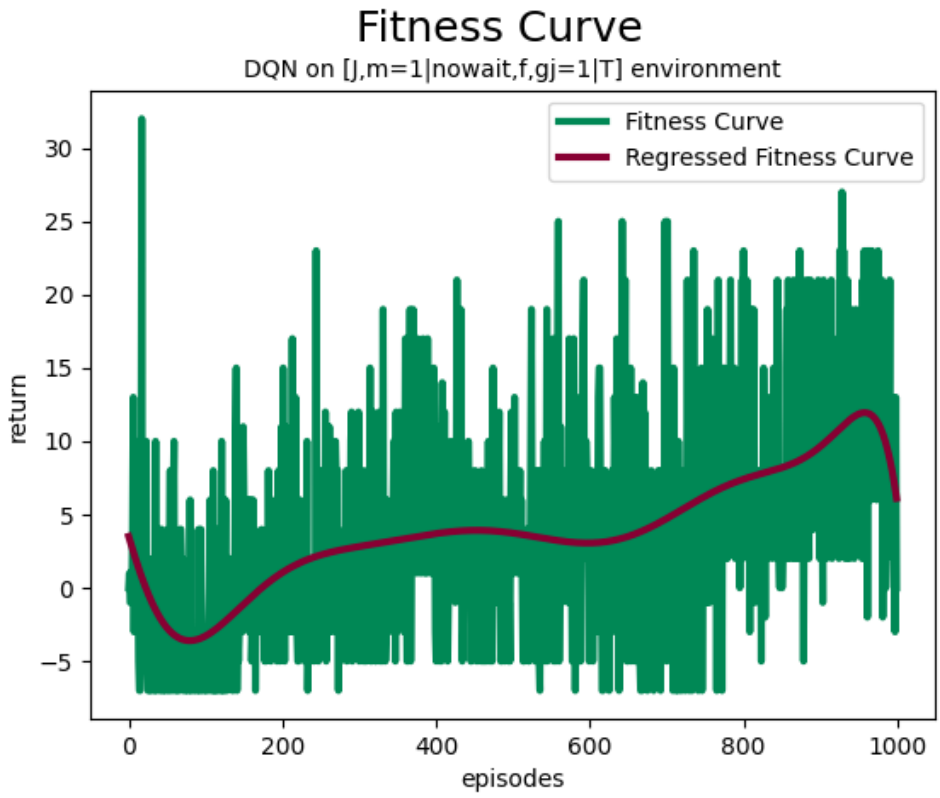

Figure 5: Fitness curve of the DQN algorithm on the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment

it gets progressively more complex. A big step towards improving the time complexity of the algorithm was the vectoring of the predictions. It is still unclear if the algorithm increases in other ways and needs further research, but a definite problem is the increased greedy selection of the actions, which requires an extra prediction for each episode. This seems very plausible as the in- and decrease of the epsilon decay hyperparameter strongly impacts the number of episodes that can be analysed inside a 2-3 hours training period.

5.2 Double Deep Q-Network (DDQN)

Building upon the foundation laid by the Deep Q-Network (DQN), the Double Deep Q-Network (DDQN) introduces a critical advancement to mitigate the overestimation of action values—a noted limitation within the DQN framework. DDQN refines the action evaluation process by employing a dual-network architecture, effectively separating the action selection from its subsequent evaluation. This division is crucial, as it addresses the overestimation bias inherent in DQN, where a single network is responsible for both selecting and evaluating actions.

In DDQN, the primary network is tasked with action selection, while a separate target network, which is periodically updated to reflect the primary network's state, evaluates these actions. This methodological shift ensures a more accurate estimation of Q-values by reducing the propensity for overestimation, leading to enhanced policy performance and more reliable decision-making in complex environments.

Empirical assessments underscore DDQN's effectiveness, showcasing its ability to provide more precise value estimates and improved performance across a suite of Atari

2600 games, as documented by (van Hasselt et al., 2016).

```
        if len(replay_buffer) > batch_size:

            batch = replay_buffer.sample(batch_size)

            states = np.array([b_state for b_state, _, _, _ in batch])
            next_states = np.array([b_next_state for _, _, _,
                                                b_next_state in batch])

            q_values_batch = \
                target_dqn_model.predict(states)
            next_q_values_batch = \
                target_dqn_model.predict(next_states)
            next_q_values_model_batch = \
                dqn_model.predict(next_states)

            for i, (b_state, b_action, b_reward, b_next_state) in
                                                enumerate(batch):
                # Get the Q-values of the state, and next state from the
                                                    target model
                q_values = q_values_batch[i]
                next_q_values = next_q_values_batch[i]
                next_q_values_model = next_q_values_model_batch[i]

                # Calculate the updated Q-value for the taken action
                q_value = q_values[b_action]
                    next_q_value = \
                        next_q_values[util.argmax(next_q_values_model)]
                q_value = (b_reward + gamma * next_q_value) - q_value
                q_values[b_action] = q_value

                dqn_input[i] = np.array(state)
                dqn_target[i] = q_values

            dqn_model.fit(np.array(dqn_input), np.array(dqn_target),
                          use_multiprocessing=True,
                          batch_size=batch_size)
```

The DDQN model commences training once the replay buffer accumulates a sufficient number of experiences. It leverages the bifurcated approach where the primary network identifies the most rewarding action, and the target network evaluates this action's Q-value. This separation crucially mitigates the overestimation bias observed in DQN, fostering a more stable and accurate learning process. The utilization of two networks in DDQN for action selection and evaluation represents a significant evolution from DQN, aiming to rectify the bias in Q-value estimation and improve the overall learning efficacy in complex decision-making environments.

Though DDQN is built upon the principles established by DQN, including the use of a target network, it distinguishes itself through the specific application of this network to decouple action selection from evaluation. This methodological advancement, while improving upon DQN's framework, demonstrates performance metrics that highlight the nuanced improvements over DQN, with DDQN achieving a modest accuracy increment to 0.60% and an MSE reduction to 17.10, thereby affirming its value in refining reinforcement learning strategies.

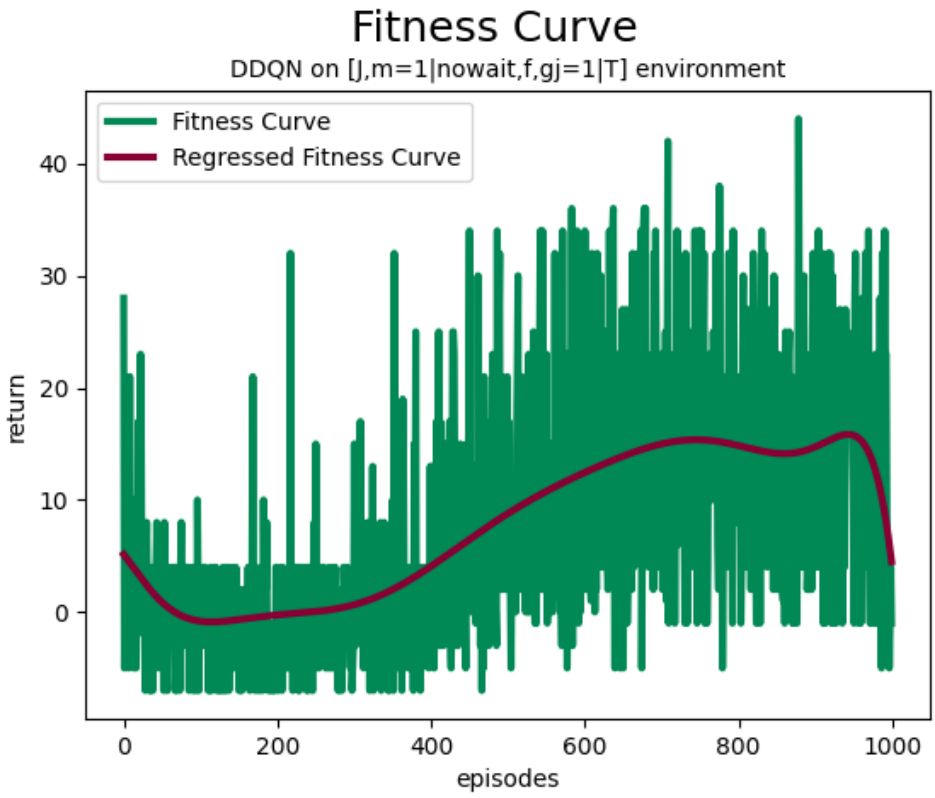

Figure 6: Fitness curve of the DDQN algorithm on the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment

5.3 Prioritized Double Deep Q-Network

Building on the foundation of Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN), the Prioritized Double Deep Q-Network (Prioritized DDQN) introduces a significant advancement by incorporating a prioritized experience replay mechanism. This section highlights the key differences between DQN/DDQN and Prioritized DDQN, particularly focusing on the novel aspects of prioritization in experience replay, as underscored by Schaul et al. (2016) in their ICLR paper on "Prioritized Experience Replay."

In traditional DQN and DDQN, experiences are stored in a replay buffer and sampled uniformly at random for learning. This approach treats all experiences as equally important for learning, which is not always efficient. Prioritized DDQN, however, introduces a method where experiences are sampled based on their importance, as measured by the magnitude of their temporal difference (TD) error. This prioritization ensures that experiences which are likely to offer more significant learning opportunities are sampled more frequently.

As a quick reminder the TD error is the reward (R) plus the best Q-Value of the next state ($Q(S_{t+1}, a)$) minus the current Q-value ($Q(S_t, A_t)$).

$$\delta_t \leftarrow R_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)$$

In our python implementation it looks as follows:

```
td_error = b_reward + gamma * next_q_value - q_value
```

Key Differences and Advancements

- **TD Error as a Proxy for Importance:** Unlike DQN and DDQN, where all transitions are considered equally for sampling, Prioritized DDQN uses the TD error to gauge the importance of each transition. Transitions with higher TD errors, indicative of being more surprising or informative, are given higher priority for being replayed. This approach aligns with the intuition that agents can learn more effectively from transitions that defy their current expectations.
- **Stochastic Sampling:** To balance between focusing on high-priority transitions and maintaining diversity in the experience replay, Prioritized DDQN employs stochastic sampling. This method ensures that even transitions with lower priority have a chance of being sampled, thereby mitigating the risk of overfitting to a subset of experiences.
- **Importance Sampling Weights:** Given that prioritized sampling changes the distribution of experiences from which the agent learns, there's a potential introduction of bias. Prioritized DDQN corrects for this bias using importance-sampling weights, ensuring that the learning updates are unbiased estimates of the true expected value.
- **Annealing the Bias:** The use of importance sampling introduces variance into the learning updates. Prioritized DDQN addresses this by annealing the importance-sampling weights, gradually reducing their effect as learning progresses. This strategy helps to stabilize learning in the face of changing priorities and sampling distributions.
- **Improved Learning Efficiency and Performance:** The incorporation of prioritized experience replay in DDQN has been shown to enhance learning efficiency significantly. By focusing on more informative experiences, Prioritized DDQN achieves faster learning and, in many cases, superior performance compared to its predecessors. Schaul et al. (2016) reported that incorporating prioritized replay into DQN resulted in a new state-of-the-art performance across a suite of Atari 2600 games.

5.3.1 PrioritizedReplayBuffer

In the path of adapting the DDQN algorithm to the Prioritized DDQN algorithm a new replay buffer had to be designed, where the batch entries have a priority associated with them. Schaul et al. recommend a binary tree object to save the batch samples, as it is communally more efficient. The 'SumTree' class implements a binary tree data structure where the value of a parent node is the sum of its children's values. For the sake of brevity it's exact implementation will be skipped.

```
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6):
        self.tree = SumTree(capacity)
        self.alpha = alpha
        self.capacity = capacity
```

The PrioritizedReplayBuffer leverages the SumTree to manage the storage and retrieval of experiences in a way that prioritizes certain experiences over others, based on their importance, to enhance learning efficiency in reinforcement learning algorithms. It introduces a novel method for managing the replay memory, a critical component of deep reinforcement learning that enables algorithms to remember and learn from past experiences. The use of α_{rb}, distinct from the learning rate, plays a pivotal role in determining how experiences are prioritized within this buffer.

The prioritization mechanism is built on the foundation of addressing issues related to greedy TD-error prioritization. Traditional approaches that rely solely on the magnitude of the temporal difference (TD) error to prioritize experiences can lead to scenarios where transitions with initially low TD errors are seldom revisited. This approach can result in a lack of diversity in the experiences replayed, leading to slower learning and potential overfitting. The PrioritizedReplayBuffer seeks to mitigate these challenges by adopting a stochastic prioritization method that balances between prioritizing transitions with high TD errors and ensuring every transition has a non-zero probability of being replayed. The formula for computing the priority of a given transition i, shown as:

$$P(i) = \frac{p_i^{\alpha_{rb}}}{\sum_k p_k^{\alpha_{rb}}}$$

where p_i represents the priority of transition i, illustrates this balance by adjusting the priority level through the exponent α_{rb}, which controls the degree of prioritization applied (Schaul et al., 2016).

```
    def _get_priority(self, td_error):
        return (np.abs(td_error) + 1e-5) ** self.alpha
```

This prioritization scheme is realized in the PrioritizedReplayBuffer through the _get_priority method. The method computes the priority of an experience based on its TD error. The formula used in the implementation is:

$$priority = (|td\_error| + \epsilon_{rb})^{\alpha_{rb}}$$

where $|td\_error|$ is the absolute value of the TD error, ensuring the priority is always positive, and $\epsilon_{rb} = 1e - 5$ is a small constant added to prevent the priority from being zero even when the TD error is zero. This implementation choice reflects the stochastic prioritization approach by assigning a non-zero probability for every transition's selection, thus ensuring a minimal level of exploration and preventing the algorithm from ignoring potentially valuable experiences due to small or zero TD errors.

The difference in implementation from the theoretical model described earlier lies in the explicit handling of edge cases through the addition of ϵ_{rb} and the practical computation of priority using the absolute TD error. This approach ensures that all experiences, regardless of their initial TD error, have a chance to influence the learning process, addressing the limitations of greedy prioritization and fostering a more robust and efficient learning environment. The use of α_{rb} allows for the fine-tuning of the prioritization process, enabling the balancing act between on high-error transitions and maintaining diversity in the experiences replayed.

```
    def add(self, state, action, reward, next_state, td_error):
        priority = self._get_priority(td_error)
        self.tree.add(priority, (state, action, reward, next_state))
```

The actual batch samples can now be added with the add function. After enough samples where added the first batch will be requested from the algorithm.

The sample method uses the concept of importance sampling (IS) weights and their integration into the prioritized replay mechanism. This mechanism is crucial for addressing the inherent bias introduced by prioritized replay, which alters the distribution of experiences sampled for learning.

The formula for calculating the importance sampling (IS) weights is given by:

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^{\beta_{rb}}$$

This equation adjusts the impact of each experience during the learning process, based on its probability of being sampled. Here, N is the total number of experiences in the replay buffer, $P(i)$ is the probability of sampling the i-th experience, and β_{rb} is a parameter that controls the degree of compensation for the non-uniform sampling.

The importance sampling weights, w_i, correct for the bias introduced by the non-uniform probabilities of sampling different experiences. This correction is necessary because prioritized replay skews the distribution towards experiences with higher TD errors, potentially leading to biased estimates of the expected value function. The weights are designed to scale down updates for experiences that are sampled more frequently than their occurrence in a uniform distribution, ensuring that the learning updates are more representative of the entire experience distribution (Schaul et al., 2016).

```
    def sample(self, batch_size, beta=0.4):
        states = []
        actions = []
        rewards = []
        next_states = []
        idxs = []
        segment = self.tree.total_priority() / batch_size
        priorities = []

        for i in range(batch_size):
            a = segment * i
            b = segment * (i + 1)
            s = random.uniform(a, b)
            (idx, p, data) = self.tree.get(s)
            priorities.append(p)
            states.append(data[0])
            actions.append(data[1])
            rewards.append(data[2])
            next_states.append(data[3])
            idxs.append(idx)

        sampling_probabilities = priorities / self.tree.total_priority()
        is_weight = np.power(self.tree.total_priority() * \
                             sampling_probabilities, -beta)
        is_weight /= is_weight.max()

        return np.array(states), np.array(actions), np.array(rewards),
               np.array(next_states), np.array(idxs)
```

The 'sample' method implementation illustrates how these concepts are applied in practice:

1. **Segmentation:** The total priority sum is divided by the batch size to create segments within the priority sum. This ensures a stratified sampling approach, where each batch sample is drawn from a different segment of the total priority range, promoting diversity in the sampled experiences.

2. **Sampling and Collecting Data:** For each segment, a random value within the segment's range is chosen to select an experience based on its cumulative priority. This process not only selects experiences based on their priority but also ensures every part of the priority spectrum is represented in the batch.

3. **Calculating Sampling Probabilities:** The probability of each sampled experience is calculated by dividing its priority by the total priority sum. This step quantifies the bias introduced by prioritized sampling.

4. **Computing IS Weights:** The IS weights are computed using the formula above, with β_{rb} as an adjustable parameter, which encays as the episode count increases. These weights are then normalized by the maximum weight in the batch to ensure they only scale the learning updates downwards, stabilizing the learning process.

5. **Return**: The method returns the sampled states, actions, rewards, next states, and indexes, along with the IS weights, ready to be used in learning updates.

The use of IS weights serves two main purposes: correcting for the sampling bias and stabilizing updates in the presence of non-linear function approximation. By ensuring that the magnitude of learning updates is inversely proportional to the frequency of sampling, IS weights help in achieving more stable and efficient learning. This approach allows the algorithm to effectively navigate the highly non-linear optimization landscapes characteristic of deep reinforcement learning, by ensuring that high-error experiences contribute more to learning while their impact is carefully moderated to prevent disruptive updates.

In summary, the integration of IS weights into the sampling process of a PrioritizedReplayBuffer addresses the bias towards high-priority experiences, ensuring a balanced and effective learning process. This methodology underpins the efficiency and success of reinforcement learning algorithms in diverse applications.

Finally there is the update function, which is needed to update the priority whenever a new TD error is calculated for a sample.

```
    def update(self, idx, td_error):
        priority = self._get_priority(td_error)
        self.tree.update(idx, priority)

    def __len__(self):
        return self.tree.size
```

5.3.2 Prioritized DDQN

This new replay buffer also necessitates some changes in the algorithm. At the start of the algorithm the beta increment is calculated. This is the amount by which beta increases each episode.

```
beta_increment_per_sampling = (rb_beta_end - rb_beta) / episodes
```

'rb_beta_end' symbolises the maximum 'rb_beta' value.

The next big difference is that the TD error needs to be calculated earlier, so it can be used to priorities batch sample. Sadly this increases the computational complexity, which makes this algorithm quit ehard to run.

```
        next_q_values_model = dqn_model.predict(next_state)[0]
        next_q_values = target_dqn_model.predict(next_state)[0]

        # Calculate the updated Q-value for the taken action
        q_values = actual_q_values.copy()
        q_value = q_values[action_index]
        next_q_value = next_q_values[util.argmax(next_q_values_model)]
        td_error = (reward + gamma * next_q_value) - q_value

        # Store experience to the replay buffer
        replay_buffer.add(state, action_index, reward, next_state,
                          td_error)
```

The fitting process is resolved in a similar fashion as before, the main difference is, that the replay buffer is updated with the newly calculated TD error.

```
        # Start training when there are enough experiences in the buffer
        if len(replay_buffer) > batch_size:
            b_states, b_actions, b_rewards, b_n_states, idxs = \
                               replay_buffer.sample(batch_size, rb_beta)

            q_values_batch = target_dqn_model.predict(b_states)
            next_q_values_batch = target_dqn_model.predict(b_n_states)
            next_q_values_model_batch = dqn_model.predict(b_n_states)

            for i, (b_state, b_action, b_reward, b_next_state, idx) in \
                    enumerate(zip(b_states, b_actions, b_rewards,
                                  b_n_states, idxs)):

                q_values = q_values_batch[i]
                next_q_values = next_q_values_batch[i]
                next_q_values_model = next_q_values_model_batch[i]

                # Calculate the updated Q-value for the taken action
                q_value = q_values[b_action]
                next_q_value = next_q_values[
                                    np.argmax(next_q_values_model)]
                td_error = b_reward + gamma * next_q_value - q_value

                replay_buffer.update(idx, td_error)
                q_values[b_action] = td_error

                dqn_input[i] = np.array(state)
                dqn_target[i] = q_values

            dqn_model.fit(np.array(dqn_input), np.array(dqn_target),
                          use_multiprocessing=True,
                          batch_size=batch_size)
```

Finally the 'rb_beta is updated at the end of the episode together with epsilon.

```
    # Epsilon decay
    epsilon = max(min_epsilon, epsilon * epsilon_decay)
```

```
    # Beta encay
    rb_beta = min(rb_beta_end, rb_beta + beta_increment_per_sampling)
```

A big problem with the algorithm is its execution time. Although the performance issues aren't solely tight to the increased number of predictions that need to be maid per iteration, but it certainly does play a role. While we where able to execute the previous algorithms over 1000 episodes on the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment, in the same time this algorithm could only run for 500 episodes. Although the training period wasn't as long for the prioritized algorithm. A learning process can be observed. In the end the algorithm can boast of a 1.70% accuracy and an MSE of 7.77.

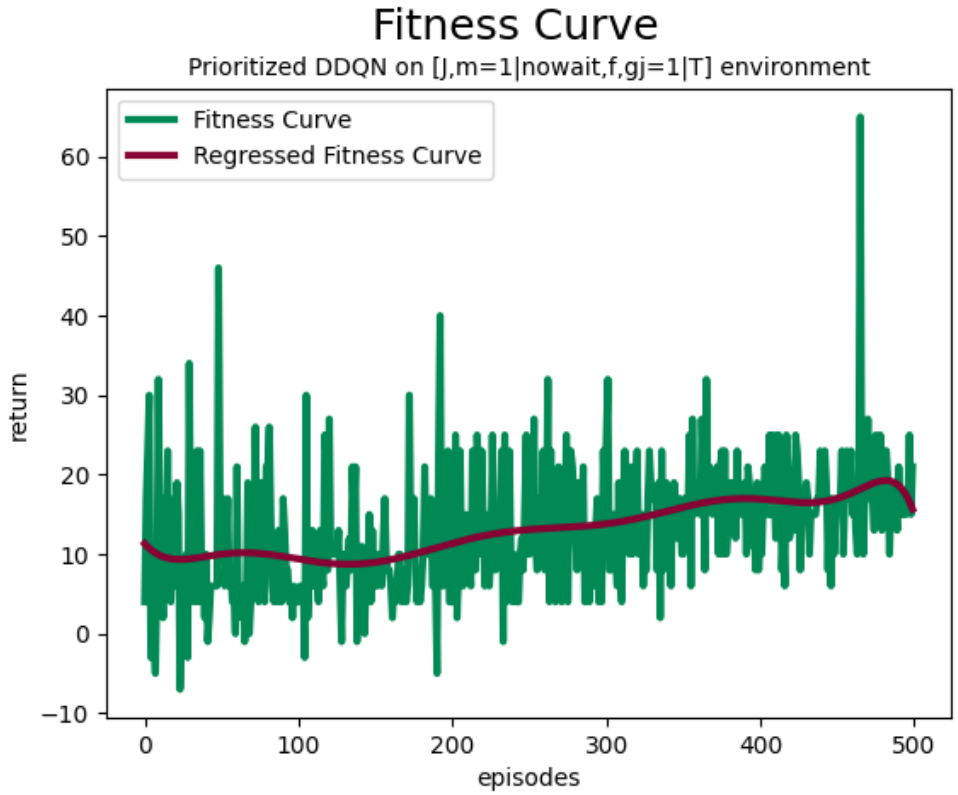

Figure 7: Fitness curve of the Prioritized DDQN algorithm on the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment

5.4 Advantage Actor-Critic (A2C)

The A2C algorithm is a different approach to reinforcement learning from Q-Learning. A2C or actor-critic methods in general are gradient-based, compared to gradient free. Actor Critic methods don't have a single neural net that attempts to optimize, but rather two. An actor and a critic. This division allows for a more nuanced approach to learning, leveraging the strengths of both components to optimize the policy and value functions concurrently (Mnih et al., 2016; Yoon, 2019; Simonini).

- **The Actor:** The actor is responsible for choosing actions given the current state of the environment. It does so based on a policy function, $\Pi(a|s)$, which maps states to actions. The goal of the actor is to learn a policy that maximizes expected rewards over time, effectively improving the decision-making strategy as learning progresses.

- **The Critic:** The critic evaluates the actions taken by the actor by estimating the value function, $V(s)$, or the expected return from a given state. This estimation helps in assessing the quality of the actions chosen by the actor, providing a feedback mechanism. The critic's objective is to minimize the difference between the estimated values and the actual returns, refining its value function estimation over time.

- **Advantage Function:** A critical component of the A2C algorithm is the advantage function, $A(s, a) = Q(s, a) - V(s)$, where $Q(s, a)$ is the action-value function representing the expected return of taking action a in state s. The advantage function measures the relative benefit of taking a specific action compared to the average outcome of all possible actions in a given state. This metric guides the actor's policy updates, focusing on actions that yield higher than expected returns.

For our implementation the previously designed DQN model needs to be adapted for both the actor and the critic.

```
def create_actor_model(input_shape, num_actions, alpha):
    model = Sequential([
        Input(shape=input_shape),
        Dense(64, activation="relu"),
        Dense(32, activation="relu"),
        Flatten(),
        Dense(num_actions, activation="softmax")
    ])
    model.compile(loss='categorical_crossentropy', optimizer=Adam(
                                        learning_rate=alpha))
    return model
```

The actor models main difference is its activation function of the output. Its activation functions 'softmax' gives the probabilities of what action should be selected.

```
def create_critic_model(input_shape, alpha):
    model = Sequential([
        Input(shape=input_shape),
        Dense(64, activation="relu"),
        Dense(32, activation="relu"),
        Flatten(),
        Dense(1, activation="linear")
    ])
    model.compile(loss='mse', optimizer=Adam(learning_rate=alpha))
    return model
```

The model of the critic is also very similar to the DQN model. The main difference is, that the critic only evaluates. The evaluation is a score, so only one output layer is necessary.

The A2C algorithm starts of in a very similar fashion to the DQN algorithm. The outlined models are initialized and the episode and iteration is started.

```
    def a2c(env, episodes, gamma, alpha, progress_bar=True):
    actor_model = create_actor_model(env.dimensions, len(env.actions),
                                        alpha)
    critic_model = create_critic_model(env.dimensions, alpha)

    # Main training loop
    for episode in range(episodes):
```

```
        state = env.get_start_state(episode)

        # If not final state
        while not env.done(state):
```

Now the action selection process has started. As before first all possible actions are retrieved from the environment. Instead of using an epsilon greedy policy, the algorithm asks the actor for the action probabilities. The action probabilities state what the actor would choose for actions. These action probabilities are masked with the possible action, so no impossible actions can be selected.

```
        # Step 1: Getting possible and impossible actions
        possible_actions, impossible_actions = \
            env.get_possible_actions(state, index=True)

        # Step 2: Get action probabilities
        action_probabilities = actor_model.predict(np.array([state]))[0]

        # Step 3: Normalizing probabilities
        masked_probabilities = np.copy(action_probabilities)
        for idx in range(len(action_probabilities)):
            if idx not in possible_actions:
                masked_probabilities[idx] = 0
        masked_probabilities = np.copy(action_probabilities)
        normalized_probabilities = masked_probabilities \
                                   / np.sum(masked_probabilities)

        # Step 4: Choosing an action
        action_index = np.random.choice(
                        np.arange(len(normalized_probabilities)),
                                   p=normalized_probabilities)
        action = env.actions[action_index]
```

Then as with the DQN the reward and next state are calculated

```
        # Take action, observe reward and next state
        next_state = env.get_next_state(state, action)
        reward = env.get_reward(state, action, next_state)
        return_ += reward
```

This very simple implementation of the algorithm does not have a replay buffer, so the actor and critic are updated in the iteration sequence. To update the actor the TD error needs to be calculated. Instead of using the Q-value to update the TD error the evaluation of the critic is used, since he tries to optimize the value function.

$$\text{DQN: } \delta_t \leftarrow r + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)$$

$$\text{A2C: } \delta_t \leftarrow r + \gamma \text{next\_ctritic\_value} - \text{ctritic\_value}$$

The TD error is the stand in for the advantage function in this case. In the code it is implemented as follows:

```
        critic_value = critic_model.predict(np.array([state]))[0]
        next_critic_value = critic_model.predict(np.array([next_state]))
                                                 [0]

        target = reward + (gamma * next_critic_value * (1 - int(env.done
                                                 (state))))
        td_error = target - critic_value
```

Now the models are updated. First the critic is updated using the TD target estimate.

```
        # Update critic
        critic_model.fit(np.array([state]), target)
```

To updated the actor a one-hot encoded representation of the action selected by the actor is created. this is used for the back propagation of the model. the TD error is used as sample weights. It is to be researched if such an approach of updating the model could be used for a DQN.

```
        actions = np.zeros([1, len(env.actions)])
        actions[np.arange(1), action_index] = 1

        # Update actor
        actor_model.fit(np.array([state]), actions, sample_weight=
                                              td_error.flatten())
```

The A2C algorithm has many advantages over a classical DQN algorithm, it was selected in the drive to work towards the rainbow algorithm (Hessel et al., 2018), an algorithm that combines many RL approaches. Sadly this could not be achieved in the given timeframe.

- **Stability and Efficiency:** By separating the policy and value function estimations, A2C achieves a more stable learning process. The critic's feedback helps in smoothing the learning updates, reducing the variance in policy updates and leading to more efficient learning.
- **Better Exploration:** The advantage function enables the actor to identify and prioritize actions that are likely to yield higher rewards, promoting a more focused exploration of the action space. This targeted exploration is more efficient than random strategies, such as epsilon greedy, leading to faster convergence.
- **Applicability to Continuous Action Spaces:** Unlike Q-Learning, which struggles with continuous action spaces, A2C is well-suited for environments where actions are not discrete. This makes A2C applicable to a broader array of problems, including robotics and autonomous vehicles.

6 Data

As part of this project, ideas were developed on how to generate and manage data for training and testing the algorithms. The fundamental idea is that there is a data generator that can generate new training or test data and can be configured via different parameters. The data is so comprehensive that it can be used for all algorithms and environments (each adapted to its needs). This is intended to create comparability of algorithms and environments, as they can be trained and tested on the exact same data.

6.1 Job data

Every job is described by the following data points. All data points are implemented as integers:

- **ids**: unique id per job (not used for learning)
- **child_foreign_keys**: not used for learning, needed to avoid cyclical dependencies of the jobs (-1 = no child job)
- **nonpreemtive_flag**: is the job preemtive or not? [0,1]
- **lead_time_total**: total lead time (in time units)
- **lead_time_todo**: remaining lead time (in time units)
- **processing_time_total**: total processing time (in time units)
- **processing_time_todo**: remaining processing time (in time units)
- **deadline (due_date)**: due date
- **done_flag**: is task done? [0,1]
- **is_task_ready**: is task ready? [0,1]

6.2 Data generation

There are two ways in which data generation can be executed. In the standard version, a new dataset is created for each episode. By calling the method *generate_new_dataset*, the data gets created according to the parameters passed.

```
def generate_new_dataset(episodes: int, numb_of_tasks: int,
                                    threshold_split: int,
                                    episode_repetition: bool = False):
    episode = 0
    index = 0
    training_distribution_list = list()
    if episode_repetition:
        training_distribution_list, threshold_episode =
                    generate_list_with_training_distribution(episodes,
                                                             threshold_split)

    while episode < episodes:
        ids = np.arange(0, numb_of_tasks)
        child_foreign_keys = dgu.validate_child_elements(numb_of_tasks)
```

```
        nonpreemtive_flag = random_choice(np.arange(0, 2),
                                                  size=numb_of_tasks)
        lead_time_total = random_choice(np.arange(0, 5),
                            size=numb_of_tasks, p=[1 / 2] + [1 / 8] * 4)
        lead_time_todo = lead_time_total
        processing_time_total = random_choice(np.arange(1, 10),
                                                  size=numb_of_tasks)
        processing_time_todo = processing_time_total
        deadline = dgu.generate_deadlines_with_target_average(
                                            numb_of_tasks, 40, (10, 50))
        done_flag = np.zeros(numb_of_tasks, dtype=int)
        is_task_ready = np.ones(numb_of_tasks, dtype=int)

        result_list = list(
            [ids, child_foreign_keys, nonpreemtive_flag, lead_time_total
                                                    , lead_time_todo,
                                                    processing_time_total,
                                                    processing_time_todo,
                                                    deadline, done_flag,
                                                    is_task_ready])
```

As it can be seen in the code section above, the generation of the job data es described in the previous subsection is straightforward and kept simple by only using integer values. The results of the generation of the different data points are temporarily stored in a results list.

The second option for data generation provides the possibility to train a certain number of episodes on the same, already known data, and to gradually add unknown new data step by step. For this, there is the *episode_repetition* flag, which, when set to True repeats the episodes according to a certain schema that can be easily adjusted in the code. An example for such a schema could be that the first 50 episodes will all be the same dataset, then a new dataset would be created for the next 40 episodes, a new one for the following 30 episodes and so on. From a certain point onwards there would be a new dataset for every episode.

```
if episode_repetition:
    if index < len(training_distribution_list):
        for temp_episode in range(int(training_distribution_list[index])
                                            ):
            dgu.save_training_data_as_pkl_file(str(episode), result_list
                                                    , episodes,
                                                    numb_of_tasks,
                                                           episode_repetition


            dgu.save_training_data_as_txt_file(str(episode), result_list
                                                    , episodes,
                                                    numb_of_tasks,
                                                           episode_repetition


            episode += 1
        index += 1
    else:
        dgu.save_training_data_as_pkl_file(str(episode), result_list,
                                            episodes, numb_of_tasks,
                                                   episode_repetition)
```

```
        dgu.save_training_data_as_txt_file(str(episode), result_list,
                                           episodes, numb_of_tasks,
                                                  episode_repetition)
        episode += 1
else:
    dgu.save_training_data_as_pkl_file(str(episode), result_list,
                                       episodes, numb_of_tasks,
                                       episode_repetition)
    dgu.save_training_data_as_txt_file(str(episode), result_list,
                                       episodes, numb_of_tasks,
                                       episode_repetition)
    episode += 1
```

The code section above is part of the *generate_new_dataset* function and in the case of the repetition of episodes realises exactly this. Furthermore the generated data for the episode gets saved as a pickle- and a text-file. Saving as a text-file was added later and is not used by the system. It is meant to provide the possibility for the developer to visually check the data used for training or testing.

For easy identification of the training and test datasets, they are stored in a dedicated folder under the directory *data/test* or *data/train*. For each generated dataset, a new folder is created in these directories following the naming scheme: day of generation - number of episodes - number of tasks - repetition True/False. The folder for the generated example dataset for the next subsection would have this naming: *2024-02-23_episodes-10_tasks-10_repetition-False*. In this folder, for each episode, a separate pickle and text file is saved. The naming starts with 0 for the first episode and increasing sequentially by 1.

6.3 Example dataset

The following example shows one episode containing ten jobs.

ids - [0 1 2 3 4 5 6 7 8 9]
child_foreign_keys- [4 -1 4 9 -1 4 2 5 3 2]
nonpreemtive_flag - [1 0 1 0 0 1 1 0 1 1]
lead_time_total - [0 0 1 4 0 1 0 0 3 4]
lead_time_todo - [0 0 1 4 0 1 0 0 3 4]
processing_time_total - [1 9 6 6 6 1 6 4 8 1]
processing_time_todo - [1 9 6 6 6 1 6 4 8 1]
deadline (due_date) - [50, 24, 15, 50, 50, 42, 34, 35, 50, 50]
done_flag - [0 0 0 0 0 0 0 0 0 0]
is_task_ready - [1 1 1 1 1 1 1 1 1 1]

6.4 Environments configuration

An important component for ensuring that the various environments can use the same datasets is the *env_dict* dictionary. This dictionary is where the data for the different environments is configured. It works according to the following principle: Each environment is identified by its own identifier through the respective notation. This identifier serves as the key in the dictionary. As the value in the dictionary, a list of integers is defined for each environment. These integers reference the index positions of the attributes in the datasets; therefore, in this case, the values 0 to 9 are permissible.

For instance, for the environment $[J, m = 1|nowait, f_j, g_j = 1|T]$, this means that only *deadline (due_date)* is needed as an attribute from the dataset. This environment will then only see the attribute *deadline (due_date)*; the other attributes will not be taken into account. However, environment $[J, m = 1|pmtn, nowait, tree, n_j, t_j, f_j, g_j = 1|T]$ utilizes all attributes from the dataset.

```
env_dict = {
    "$[J|nowait,t_j,g_j=1|D]$": [5],
    "$[J,m=1|nowait,f_j,g_j=1|T]$": [7],
    "$[J,m=1|pmtn,nowait,tree,n_j,t_j,f_j,g_j=1|T]$": [0, 1, 2, 3, 4, 5,
                                                       6, 7, 8, 9]
}
```

6.5 Getting the start state

During the training of the algorithm, new states for the environment are required for each iteration (episode). The environment loads these states using the *get_start_state* function. In order for this function to be able to provide the corresponding correct data, the information about the environment is required. It must be specified how many jobs are being trained with, as well as which episode it is in, and the directory name of the data to be used for training must also be passed.

The function loads the corresponding pickle file for the episode, checks in the *env_dict* which attributes from the dataset are required for the calling environment, temporarily stores these, and finally returns them to the environment, or the training algorithm, respectively.

```
def get_start_state(env_name: str, number_of_tasks: int,
                                    num_episode: int, dir_name: str):
    result_list = dgu.read_list_from_pkl_file(str(num_episode),
                                                          dir_name)
    temp_result_list = []
    for item in env_dict.get(env_name):
        temp_result_list.append(result_list[item][:number_of_tasks])
    if 1 in env_dict.get(env_name):
        temp_result_list =
                dgu.remove_child_element_keys_that_are_too_high(
                                                    temp_result_list)
    return temp_result_list
```

6.6 Execution log

The application provides the option to save information about the training process and the achieved results in a log file via the *save_log_file* flag. For this to happen, the *save_log_file* flag must be set to True in *main.py*. If this is the case, the results will be saved in a JSON file named *execution_log.json* in a new folder within the */data/evaluation/* directory after the algorithm has been trained. The structure of the *execution_log.json* file will be explained subsequently. The following extraction shows the essential structure of the JSON document.

```
{
    "Execution Time": "2024-02-22 12:16:56",
    "Duration (seconds)": 1763.9739382266998,
    "System Configuration": {...},
    "Hyperparameters": {...},
    "Result": {...},
    "Model Path": "models/[J-nowait,t,gj=1-D]
       _DQN_20240222_124619"
}
```

The document contains information about the time when the training was executed and how long it took to train the model (duration). Furthermore information about the system configuration and resource utilization is logged. The hyperparameters are logged to be able to recreate the model or at least understand the configuration used for training. In addititon, the results also get logged in the file as well as the path, where the model is saved. Due to the fact that the *execution_log.json* can contain an substantial amount of data the following subsection briefly describe what can be found in which section of the JSON document.

6.6.1 System Configuration

The system configuration includes information about the hardware used and how it was utilized during the course of the training.

```
{
    "System Configuration": {
        "CPU Model": "12th Gen Intel(R) Core(TM) i7-12700H",
        "CPU Usage (%)": {
            "50": 5.5,
            "95": 6.2,
            "99": 6.6
        },
        "CPU Usage": [...],
        "TF GPU": [
            "PhysicalDevice(name='/physical_device:GPU:0',
               device_type='GPU')"
        ],
        "GPU ID": 0,
        "GPU Name": "NVIDIA GeForce RTX 3080",
        "Total Memory (MB)": 10240.0,
        "Driver": "546.01",
        "GPU Load": {
            "50": 10.0,
            "95": 12.0,
            "99": 18.0
        },
        "GPU Memory Used (MB)": [...],
        "GPU Temperature (C)": [...],
        "Total RAM": 7.598011016845703,
```

```
        "RAM Usage (%)": {
            "50": 75.55,
            "95": 98.8,
            "99": 99.0
        },
        "RAM Usage": [...],
    }
}
```

The information about the GPU can only be displayed for Nvidia graphics cards. Should there be a decrease in performance, these logged data might be of interest. They make it possible to check which component of the hardware was particularly stressed (CPU, GPU, or RAM). The data on *CPU Usage*, *GPU Memory Used (MB)*, *GPU Temperature (C)*, and *RAM Usage* are logged after a definable unit of time (for example, every 2 seconds).

6.6.2 Hyperparameters

The hyperparameters with which the model was trained are also saved in the *execution_log.json*. Except for the episode count, the environment, and the algorithm, the hyperparameters listed below are the same ones used for the training of all graphs or other evaluation results shown in this paper.

```
{
    "Hyperparameters": {
        "environment": "[J|nowait,t,gj=1|D]",
        "max_numb_of_machines": 3,
        "max_numb_of_tasks": 9,
        "max_task_depth": 10,
        "fixed_max_numbers": true,
        "high_numb_of_tasks_preference": 0.35,
        "high_numb_of_machines_preference": 0.8,
        "algorithm": "DQN",
        "episodes": 200,
        "epochs": 1,
        "gamma": 0.85,
        "epsilon": 1,
        "alpha": 0.0001,
        "epsilon_decay": 0.95,
        "min_epsilon": 0.01,
        "rb_alpha": 0.6,
        "rb_beta": 0.4,
        "rb_beta_end": 1,
        "batch_size": 128,
        "update_target_network": 100,
        "numb_of_executions": 1,
        "model_name": "[J-nowait,t,gj=1-D]_DQN_20240222_124619",
        "save_final_dqn_model": false,
        "save_log_file": true,
        "training_dir_name": "2024-02-04_episodes-690000_tasks
            -100",
```

```
        "test_dir_name": "2024-02-21_unlabeled-dir-date
            -2024-01-17_epochs-1000_tasks-9_env-[J,m=1-nowait,f,
            gj=1-T]"
    }
}
```

All hyperparameters are saved in the file. This includes information such as the environment used, the algorithm used, as well as the training and test data used. In addition, parameters such as the number of episodes, gamma, epsilon, epsilon_decay, batch_size, and a few others are logged. The purpose of this is to ensure comparability between the different models and potentially reproducibility of the model.

6.6.3 Result

The result section from the *execution_log.json* contains the following information:

```
{
    "Result": {
        "fitness_curve": [
            74.00000000000003,
            138.33333333333331,
            -2.9999999999999574,
            131.66666666666669,
            157.33333333333334,
            .
            .
            .
            22.666666666666615
        ]
    }
}
```

Here, the data points for visualizing the fitness curve are saved to enable its regeneration later if needed. However, the fitness curve is also saved separately as a PNG file in the same folder alongside the *execution_log.json*.

7 Environments

As part of this work, the two environments from the last paper were revised and a new environment was created. These are explained in the following sub-chapters.

7.1 [J,m=1|nowait,f_j,g_j=1|T]

Previously: TimeManagement

This environment already existed in the our previous paper. Since then small adaptations where made to it. The most noticeable on is the name change. Instead of naming the environment arbitrarily it is now named after machine scheduling standards (see chapter 2). This environment optimises the due date (f_j) of one machine ($m = 1$). The main goal of this environment is to meet all due dates (T).

This is a very simplistic environment. A states in environment encapsulate the entire scenario where several tasks are to be sorted according to their due dates. Each state is defined by a pair of arrays: the tasks array and the result array.

The tasks array is a list of jobs, with each job represented by an integer indicating its due date. A smaller integer denotes an earlier due date. For example, in the initial state of an episode, the tasks array may look as follows:

```
tasks: [2, 3, 4, 1]
```

Conversely, the result array is intended to hold the sorted tasks. This is essentially the history from the tabular Q-learning example. At the beginning of an episode, it is initialized with zeros:

```
result: [0, 0, 0, 0]
```

A full state would look as follows:

```
[
[2, 3, 4, 1],  # the tasks
[0, 0, 0, 0]   # the result
]
```

As actions are executed, tasks are moved from the tasks array to the result array. For instance, let's take the action *[0, 1]*. This action moves the task from index *0* of the tasks list to index *1* of the result list. The corresponding position in the tasks list is then replaced with a zero, symbolizing that the task's position has been decided:

After executing the action *[0, 1]*:

```
[
[0, 3, 4, 1],  # the tasks
[0, 2, 0, 0]   # the result
]
```

In this scenario, the task with the due date *2* has been moved to the second position in the result array, and its original position in the tasks array has been marked with a zero.

The process continues iteratively until all tasks from the tasks array have been moved to the result array, at which point the result array represents the sorted sequence of tasks according to their due dates. The state representation, therefore, reflects the transition

of tasks from being unsorted to a state of being sorted, effectively capturing the evolving nature of the task management scenario. The agent can therefor learn well, as all the information he needs for a decision is in the state.

7.1.1 Possible Actions

The main change in this environment is the way possible actions where calculated. The previous calculations where quite resource intensive, having a time complexity of $O(n^2)$. To remove this complexity the approach of getting the possible actions where changed. The method sill filters out methods that are impossible, but with a lighter touch. Some of the possible actions are actually impossible. In other words the possible action function only removes some illegal actions. Instead the reward function now punishes illegal actions.

```
def get_possible_actions(self, state, index=True):
    # Function to determine possible actions in the current state
    possible_actions = []
    impossible_actions = []

    i = 0
    for action in self.actions:
        if state[1][action[1]] == 0 or action[1] == -1:
            if index:
                possible_actions.append(i)
            else:
                possible_actions.append(action)
        else:
            if index:
                impossible_actions.append(i)
            else:
                impossible_actions.append(action)
        i += 1

    return np.array(possible_actions), np.array(impossible_actions)
```

Aside from the removed complexity, an index mode was also added. Which returns the index of the action, as that is often more prevalent.

7.1.2 Reward

The reward function is the hardest part to design of the environment. Here several changes where made to it. First of the reward function heavily punishes illegal actions (with negative 10). If the action taken was illegal it also stop the current episode (to prevent endless loops). If the action was not illegal, the algorithm is rewarded for previous progress made. If all tasks before it had a lower number in the result array it is rewarded (with plus 2) and it is also rewarded if all the numbers following it are higher (also with plus 2). In case the correct final state was reached that algorithm is also rewarded with plus 10.

```
def get_reward(self, state, action, next_state):
    current_f = state[0][action[0]]

    if sum(sum(s) for s in state)!= sum((sum(ns) for ns in next_state))\
            or np.count_nonzero(state[1] == 0) != \
                            (np.count_nonzero(next_state[1] == 0)+1)\
```

```
            or current_f == 0:
        self.done_flag = True
        return -10

    before_position = state[1][0:action[1]]
    after_position = state[1][action[1]+1:len(state[1])]
    reward = np.count_nonzero(next_state[1] != 0)

    if len(before_position) > 0:
        if max(before_position) <= current_f:
            reward += 2
    if len(after_position) > 0:
        if min(after_position) >= current_f or max(after_position) == 0:
            reward += 2

    if sum(next_state[0]) == 0:
        reward += 10

    return reward
```

7.1.3 Results

For this environment a supervised method was designed, so that a training set could be set up. This means that the environments could be evaluated with supervised methods such as calculating the accuracy. Though this approach is a bit problematic, as the test dataset has a one hot result approach. Meaning that the correct actions are flagged with a 1 for being correct, while the false actions are marked with a 0. But the algorithms are not trained using such an approach. instead the action is picked with the highest Q-value irrespective of it's size. This is why the results are quite poor. These evaluation tools where not adapted to the A2C models, because their special approach with an actor and a critic can not be measured with the current implementation. What is possible with a test set though is running a supervised algorithm.

	$[J, m = 1 \vert nowait, f_j, g_j = 1 \vert T]$
DQN	2.20%
DDQN	0.60%
Prioritized DDQN	1.70%
A2C	-
Supervised	0.25%

Table 1: Accuracy Results Between algorithms

	$[J, m = 1 \vert nowait, f_j, g_j = 1 \vert T]$
DQN	7.19
DDQN	17.10
Prioritized DDQN	7.77
A2C	-
Supervised	2.36

Table 2: Mean Squared Error Results Between algorithms

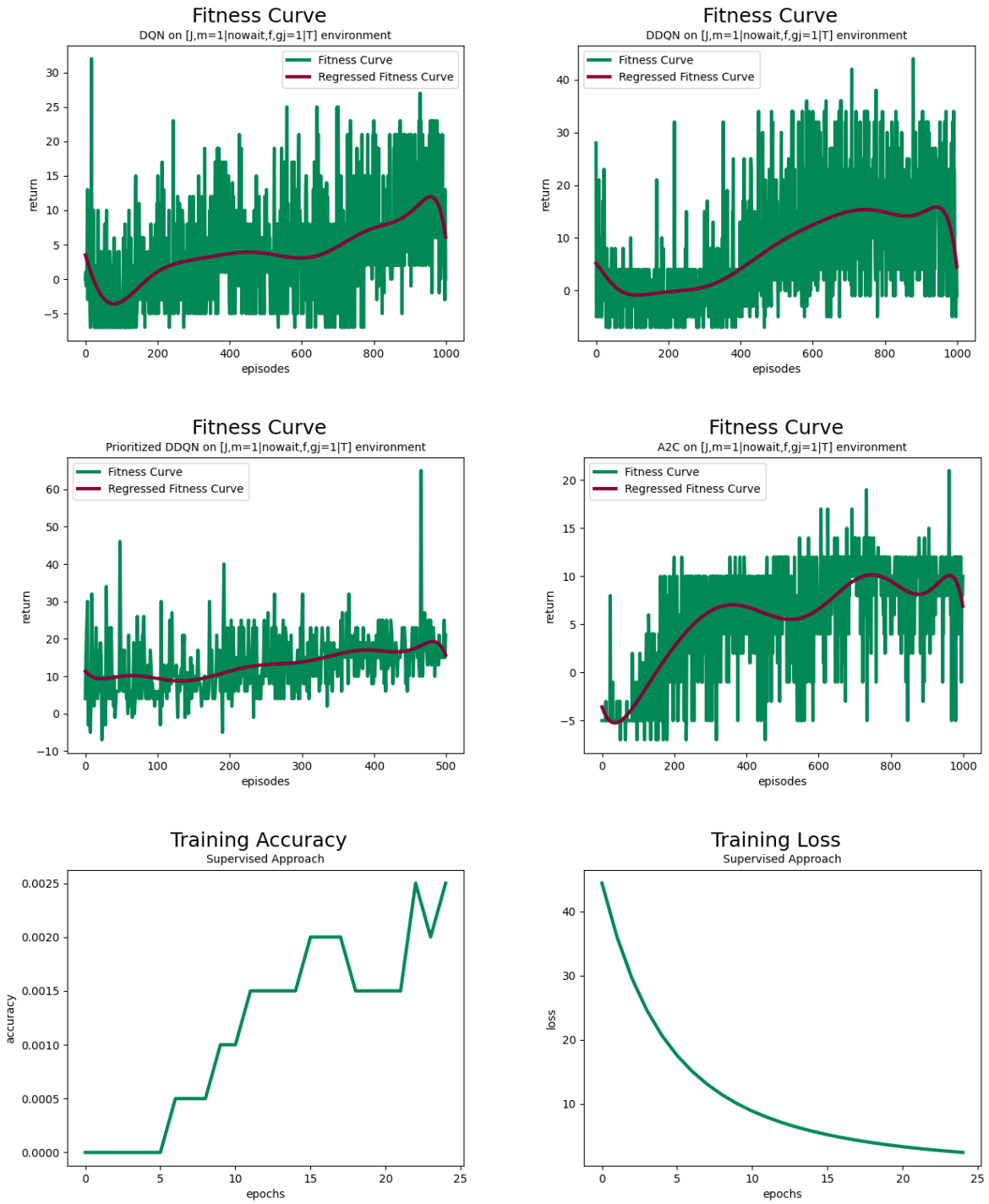

Figure 8: Showing the Fitness Curves and other evaluations from the $[J, m = 1|nowait, f_j, g_j = 1|T]$ environment

7.2 [J|nowait,t,g$_j$=1|D]

Previously: ResourceManagement

This environment was also already present in the previous work and the main features were explained there. As well as in the previous environment, the naming of the environment changed to the notations explained in chapter 2. The goal of this environment is to optimize (minimize) the overall execution time (throughput) for the given jobs. The different jobs do not have to be executed in a particular order, but the associated duration

of each job is of interest.

A state in the [J|nowait,t,g_j=1|D] environment is defined as follows:

```
[
[4,3,2,1,0,0,0,0,0,0], # the jobs
[0,0,0,0,0,0,0,0,0,0], # machine zero
[0,0,0,0,0,0,0,0,0,0], # machine one
[0,0,0,0,0,0,0,0,0,0]  # the step / the index that
                                    the actions apply to
]
```

Here, the jobs list represents the set of jobs yet to be assigned, with the number indicating the duration of each job. The subsequent lists represent the machines, with each machine being assigned a certain job (or multiple jobs) represented by non-zero entries in the list. The last list represents the current time step for each machine.

An action is a list of two integers. For example the action *[1,0]* represents assigning the first job (indexing starts at 1 for jobs to distinguish them from unassigned slots represented by zeros) to machine zero. Upon executing this action, the next state would look like:

```
[
[0,3,2,1,0,0,0,0,0,0], # the jobs
[1,1,1,1,0,0,0,0,0,0], # machine zero
[0,0,0,0,0,0,0,0,0,0], # machine one
[0,0,0,0,0,0,0,0,0,0]  # the step / the index that
                                    the actions apply to
]
```

The special action *[-1,-1]* represents advancing the time step. After executing this action, the next state would look like:

```
[
[0,3,2,1,0,0,0,0,0,0], # the jobs
[1,1,1,1,0,0,0,0,0,0], # machine zero
[0,0,0,0,0,0,0,0,0,0], # machine one
[1,1,1,1,1,1,1,1,1,1]  # the step / the index that
                                    the actions apply to
]
```

Now if the action *[2,1]* is chosen, it assigns the second job to machine one at the next available time step (after advancing the step with the previous action). This results in the following state:

```
[
[0,0,2,1,0,0,0,0,0,0], # the jobs
[1,1,1,1,0,0,0,0,0,0], # machine zero
[0,2,2,2,0,0,0,0,0,0], # machine one
[1,1,1,1,1,1,1,1,1,1]  # the step / the index that
                                    the actions apply to
]
```

The state representation combined with the action space allows for complex interactions and job scheduling sequences.

7.2.1 Reward

The calculation of the reward also proved to be a challenge in this environment and represents the point at which the environment was revised once again. The idea behind the rewards in this environment is to strictly punish illegal actions. Therefore the legality of the action is determined by the following considerations:

- If the sum of tasks remains the same and the step hasn't advanced, the action is considered illegal.
- If the action tries to assign a task that doesn't exist (tasks[action[0]-1] == 0) and the action is not a special 'no-operation' action (action[0] != -1), it's illegal.
- It checks if the change in the number of zeros (idle slots) in machines from the current to the next state matches the expected change based on the task's requirements. If it doesn't match and the action isn't a 'no-op', the action is illegal.
- If there's an attempt to perform a 'no-op' action (action[0] == -1) when there are no tasks that can be scheduled in the current step, it's considered illegal.

If the action is determined to be illegal, the function sets a *done_flag* to True, indicating the end of the episode, and returns a reward of -100.

If the action legal, the reward is calculated as follows:

- The initial reward is the count of tasks that have been completed up to a certain number (max_numb_of_tasks).
- A penalty is applied based on the difference between the worst distribution of tasks among machines (util.current_worst) and an assumed optimal distribution (util.assumed_optimal), to encourage even distribution.
- An additional reward of 100 is given if all tasks have been completed (sum(next_tasks) == 0).

```
def get_reward(self, state, action, next_state):

    machines, tasks, step = self.extract_info_from_state(state)
    next_machines, next_tasks, next_step = self.extract_info_from_state(
                                        next_state)

    legal = True

    if sum(tasks) == sum(next_tasks) and step == next_step:
        legal = False

    s_zeros = sum([np.count_nonzero(m == 0) for m in machines])
    ns_zeros = sum([np.count_nonzero(m == 0) for m in next_machines])

    if tasks[action[0]-1] == 0 and action[0] != -1:
        legal = False

    if s_zeros != ns_zeros+tasks[action[0]-1] and action[0] != -1:
        legal = False

    if sum([m[step] for m in machines]) == 0 and action[0] == -1:
```

```
            legal = False

        if legal:
            reward = np.count_nonzero(
                                next_tasks[0:self.max_numb_of_tasks] == 0)

            # penalty for uneven distribution of tasks
            reward -= (util.current_worst(self.current_cumulative_machines)
                                        - util.assumed_optimal(
                                        self.current_cumulative_machines))

            if sum(next_tasks) == 0:
                reward += 100

            return reward
        else:
            self.done_flag = True
            return -100
```

7.2.2 Results

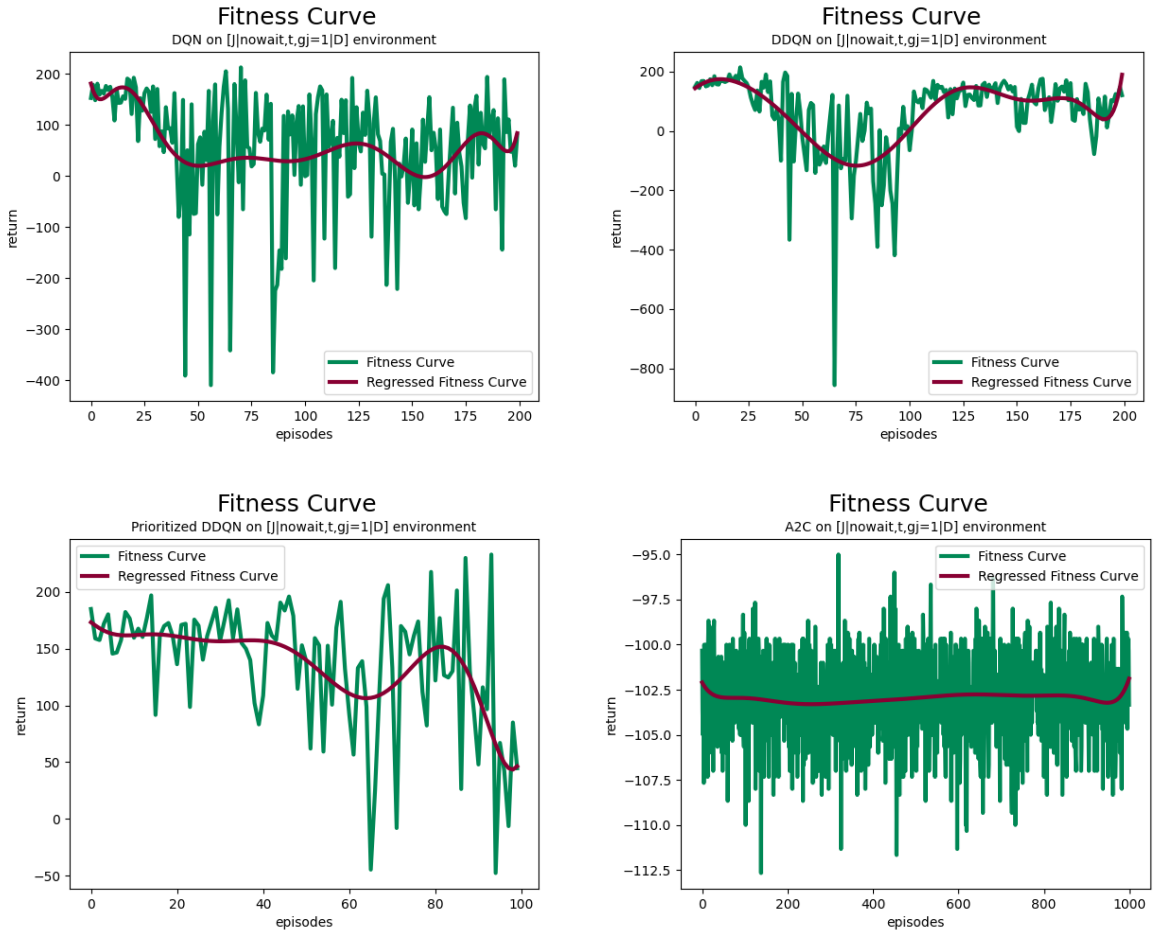

Figure 9: Showing the Fitness Curves from the [J|nowait,t,g_j=1|D] environment

As shown by the fitness curves in Figure 9, the environment does not produce satisfactory results. The visualizations clearly show that none of the algorithms were able to achieve substantial learning progress. A possible reason for this could be that training should have been conducted for a much longer period to see such progress. However, a

problem was that as training progressed, the calculations per episode took increasingly more time. Figure 10 shows an example of this problem by needing over 600 seconds (10 minutes) to calculate one episode. Also the training process is killed and exits with code 137 indicating a memory issue where the algorithm apparently required too much memory and got killed. At this point, it should be pointed out once again that the implementation of the algorithms could probably be made even more efficient and thus possibly achieve better performance of the computing power. It would also be an option to train the algorithms on a more powerful hardware. Interestingly, as can be seen in the Figure 9, it was possible to train the A2C algorithm over a number of 1000 episodes. Nevertheless, it did not achieve satisfactory results either.

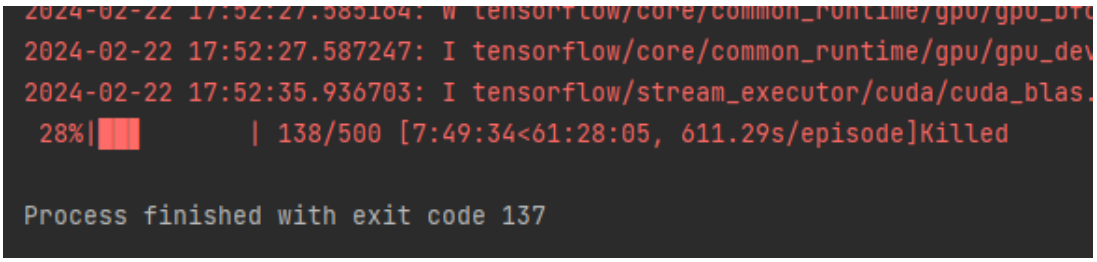

Figure 10: Execution time problem for [J|nowait,t,g_j=1|D] environment

7.3 [J,m=1|pmtn,nowait,tree,n_j,t_j,f_j,g_j=1|T]

The $[J, m = 1|pmtn, nowait, tree, n_j, t_j, f_j, g_j = 1|T]$ environment was created with the intention of creating a more complex one machine ($m = 1$) environment. Some tasks are preemptive, some are not ($pmtn$). Some tasks need to be completed first, before others can be touched ($tree$). Though tasks can not have more than one parent tasks, that needs to be completed before it ($!prec$). Some tasks have lead times (n_j). En example for a lead time would be drying paint. All tasks have both a processing time (t_j) and a due date (f_j). The algorithm should attempt to optimise the tardiness. In other words all tasks should be completed before their Due Date (T)

Since it is quite complicated to track all these elements, the state is also quite large. A state is built up of 10 arrays. the first array is a list of ids.

```
[0, 1, 2, 3, 4, 5, 6, 7, 8] # IDs
```

The second array consist of 'child_foreign_keys'. This is the functionality, that enables the ($tree$) functionality. Entries that have a parent (i.e. a child foreieign key entry) may not be executed, before their parent. In this case the job with the ID 0 may only be executed after job 6 has been executed.

```
[-1, -1, -1, -1, -1, -1, 0, -1, -1] # child foreign keys
```

The -1 value in the list symbolises that the job has no child. The third array tells the algorithm whether the job is preemptive. If the flag is set to 1 then the task is not preemptive, i. e. it can not be paused. If it is set to 0 the task can be paused.

```
[0, 0, 0, 1, 0, 0, 1, 1, 0] # non-preemtive flag
```

The fourth and fifth array deal with the lead time. The third array says how much lead time the task had at the beginning of the episode, the fourth array says what the current lead time is.

```
[0, 2, 4, 3, 0, 0, 0, 4, 3] # total lead time
[0, 2, 4, 3, 0, 0, 0, 4, 3] # current lead time
                            # / lead time in todo
```

The sixth and seventh array deal with the processing time. Since tasks can be preemptive they can be worked on for some time, and then be left on pause for some time. This is why there is also a current processing time and a total processing time. The smallest time interval to which a task is divisible is the smallest possible whole number.

```
[9, 3, 7, 4, 6, 9, 7, 8, 3] # total processing time
[9, 3, 7, 4, 6, 9, 7, 8, 3] # current processing time
                            # / processing time in todo
```

The eighth array is the due date or deadline.

```
[34, 2, 6, 34, 34, 2, 43, 35, 32] # deadline
```

The ninth array symbolises the done flag. which is 1 in case a task is finished.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0] # done flag
```

The last array symbolises whether a task can be started or not. Tasks may not be started if if they are not ready or if they have been processed but still need some lead time.

```
[0, 1, 1, 1, 1, 1, 1, 1, 1] # is ready flag
```

Put together a state looks as follows:

```
[
[0, 1, 2, 3, 4, 5, 6, 7, 8], # IDs
[-1, -1, -1, -1, -1, -1, 0, -1, -1], # child foreign keys
[0, 0, 0, 1, 0, 0, 1, 1, 0], # non-preemtive flag
[0, 2, 4, 3, 0, 0, 0, 4, 3], # total lead time
[0, 2, 4, 3, 0, 0, 0, 4, 3], # lead time in todo
[9, 3, 7, 4, 6, 9, 7, 8, 3], # total processing time
[9, 3, 7, 4, 6, 9, 7, 8, 3], # processing time in todo
[34, 2, 6, 34, 34, 2, 43, 35, 32], # deadline
[0, 0, 0, 0, 0, 0, 0, 0, 0], # done flag
[0, 1, 1, 1, 1, 1, 1, 1, 1], # is ready flag
]
```

Actions are in comparison relatively simple in this environment as they only consist of one value. say the action 7 where to be selected, then the job with the id 7 would be processed. Since the state is not preemptive it is processed all at once as it can not be stopped. This is how the next state would look like.

```
[
[0, 1, 2, 3, 4, 5, 6, 7, 8], # IDs
[-1, -1, -1, -1, -1, -1, 0, -1, -1], # child foreign keys
[0, 0, 0, 1, 0, 0, 1, 1, 0], # non-preemtive flag
[0, 2, 4, 3, 0, 0, 0, 4, 3], # total lead time
[0, 2, 4, 3, 0, 0, 0, 4, 3], # lead time in todo
[9, 3, 7, 4, 6, 9, 7, 8, 3], # total processing time
```

```
[9, 3, 7, 4, 6, 9, 7, 0, 3], # processing time in todo
[26,  6 ,  2 , 26, 26,  6 , 35, 27, 24], # deadline
[0, 0, 0, 0, 0, 0, 0, 0, 0], # done flag
[0, 1, 1, 1, 1, 1, 1, 0, 1], # is ready flag
]
```

The task has now been completed, but it is still not done, as the this task has lead time. But now an other job can start, since this job no longer needs active work. Something that changed in the state (outside of the changes of the id 8 task) is that the deadline has been reduces/ moved forward. It took 8 time intervals to complete this task, so the deadline for the other tasks has approached. Let us say that action 6 is now selected.

```
[
[0, 1, 2, 3, 4, 5, 6, 7, 8], # IDs
[-1, -1, -1, -1, -1, -1, 0, -1, -1], # child foreign keys
[0, 0, 0, 1, 0, 0, 1, 1, 0], # non-preemtive flag
[0, 2, 4, 3, 0, 0, 0, 4, 3], # total lead time
[0, 2, 4, 3, 0, 0, 0, 0, 3], # lead time in todo
[9, 3, 7, 4, 6, 9, 7, 8, 3], # total processing time
[9, 3, 7, 4, 6, 9, 0, 0, 3], # processing time in todo
[19,  13 ,  9 , 19, 19,  13 , 28, 23, 17], # deadline
[0, 0, 0, 0, 0, 0, 1, 1, 0], # done flag
[1, 1, 1, 1, 1, 1, 1, 1, 1], # is ready flag
]
```

Since the lead time of the task with the id 7 has now been completed, it's done flag is set to 1. Also while all other deadlines moved forward by 7 but its deadline only moved forward by 4. The point when it was done. Since task 6 is now also done task 0 may now be selected. Since it is now possible, task 0 is selected.

```
[
[0, 1, 2, 3, 4, 5, 6, 7, 8], # IDs
[-1, -1, -1, -1, -1, -1, 0, -1, -1], # child foreign keys
[0, 0, 0, 1, 0, 0, 1, 1, 0], # non-preemtive flag
[0, 2, 4, 3, 0, 0, 0, 4, 3], # total lead time
[0, 2, 4, 3, 0, 0, 0, 0, 3], # lead time in todo
[9, 3, 7, 4, 6, 9, 7, 8, 3], # total processing time
[8, 3, 7, 4, 6, 9, 0, 0, 3], # processing time in todo
[18,  14 ,  10 , 18, 18,  14 , 27, 22, 16], # deadline
[0, 0, 0, 0, 0, 0, 1, 1, 0], # done flag
[1, 1, 1, 1, 1, 1, 1, 1, 1], # is ready flag
]
```

Since task 0 is preemptive, the environment only moved forward by one time step, since the algorithm could now consider to chose a different action. This process continues iterative until all tasks are done and the done flag everywhere is one. This environment symbolises quite well why job sop management is relevant, since at a certain complexity level it would also become hard for humans to solve this problem.

7.3.1 Reward

At the start, the reward function distinguishes between possible and impossible actions from the current state by calling *self.get_possible_actions(state, index=False)*. This differentiation is crucial for evaluating whether the chosen action is feasible within the given state's context.

It then checks if the action taken is one of the impossible actions. If the action is impossible (i.e., not allowed in the current state), it sets a flag *legal* to False. Actions deemed impossible result in a significant penalty to discourage the selection of such actions. The reward calculation for legal actions is as follows:

- If the action is legal, it iterates through a specific subset of the state (referred to by self.processing_time_todo, self.is_task_ready, and self.done_flag) to find an action that meets certain criteria: the task is ready (is_task_ready is 1), not yet completed (done_flag is 0), and has the minimum processing time among those that meet the first two criteria.
- If the chosen action (based on the minimum processing time criterion) does not match the action taken, a reward of -1 is returned, indicating that the action taken was not the most efficient or correct choice given the current state.
- If the chosen action matches the action taken, it returns a reward of 1, indicating a correct or optimal action under the circumstances.

If the action is deemed illegal (i.e., it was among the impossible actions for the current state), the function sets a flag *self.is_done* to True, which indicated that the episode or task is terminated due to an illegal action. It then returns a reward of -100, serving as a substantial penalty to strongly discourage choosing actions that are not allowed.

```
def get_reward(self, state, action, next_state):
    # Function to calculate the reward based on the state, action, and
                                        next state

    possible_actions, impossible_actions = self.get_possible_actions(
                                        state, index=False)

    legal = True

    if action in impossible_actions:
        legal = False

    if legal:
        min_time = float('inf')
        chosen_action = None
        for i, processing_time in enumerate(state[
                                        self.processing_time_todo]):
            if state[self.is_task_ready][i] == 1 and
                            state[self.done_flag][i] == 0 and
                                        processing_time < min_time:
                min_time = processing_time
                chosen_action = i

        if chosen_action != action:
            return -1
        else:
```

```
            return 1
    else:
        self.is_done = True
        return -100
```

7.3.2 Result

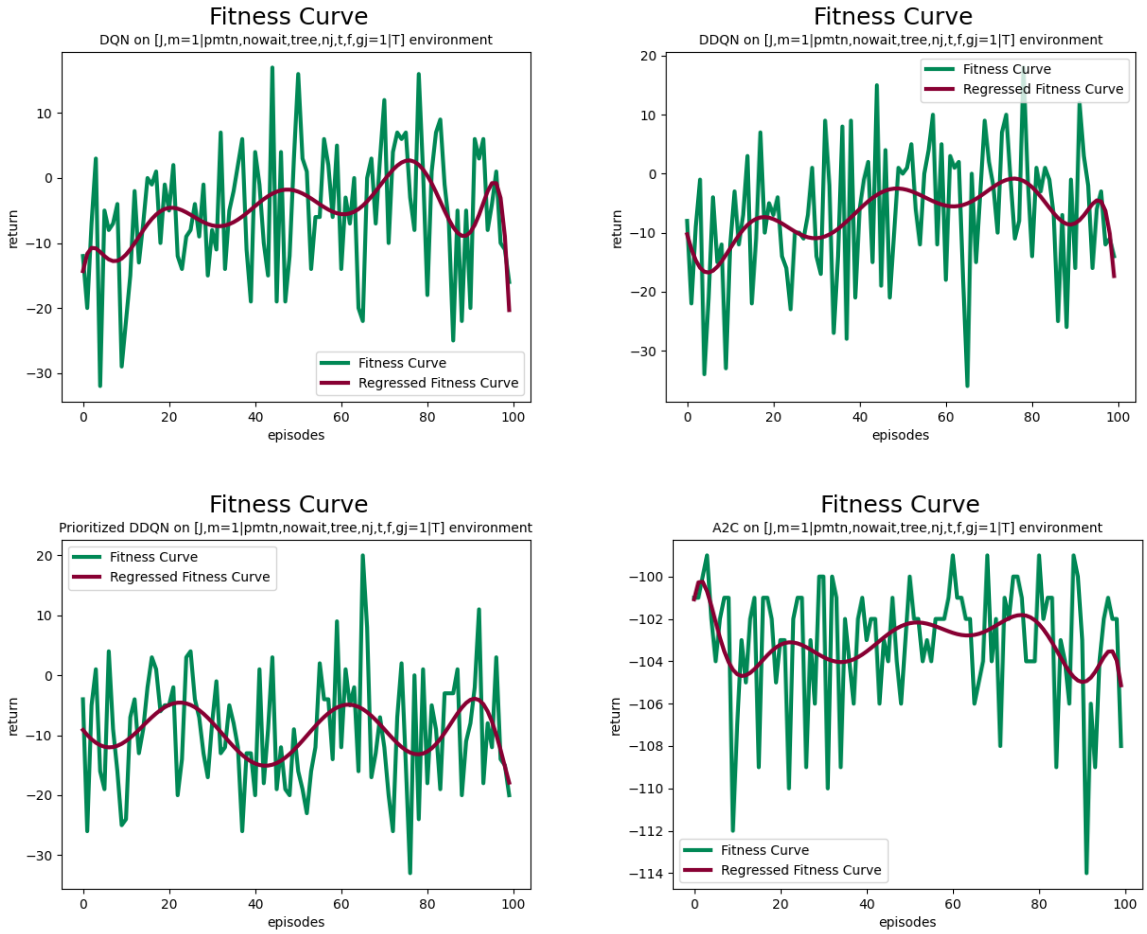

Figure 11: Showing the Fitness Curves from the [J,m=1|pmtn,nowait,tree,n_j,t_j,f_j,g_j=1|T] environment

Figure 11 shows that the algorithms do not produce satisfactory results in this environment either. The visualizations clearly show that no substantial learning progress could be achieved with any of the algorithms. in the 100 episodes that they could be trained for.

8 Improvment Consepts

This Project presented us with many challenges. One the Cynefin Framework it would have been characterised as a chaotic project. Every exploration of every possible problem lead to further problems and opened up a new can of worms. In the quest of optimising JSS problems not all our improvement ideas could be realised. Since the execution time increased with increasing episodes, it became nearly impossible to run some environment for more then 100 episodes. To develop this project further it is absolutely crucial to decrease the execution time. Some changes have yielded very promising results but could sadly not be implemented across all environments and all algorithms. One of those is the vectorization of the predictions and other calculations, as this greatly reduced the processing time when implemented. Some environments are still cluttered with unnecessary time and resource complex operations. They should be streamlined and reprogrammed with a focus on efficiency. A very powerful brute force method of renting a server and letting the algorithm run for several weeks should also be attempted. Further a visual debug tool for evaluating the actions of the algorithm would also be very nice. With it mistakes could be spotted easier. A stronger algorithm such as the Rainbow algorithm is a quest we actively perused but could not complete.

9 Conclusion

In reflecting upon this investigation into the use of Deep Reinforcement Learning for Job Shop Scheduling, it becomes evident that the field has considerable hurdles to overcome when implementing DRL approaches for JSS problems. The development of the RL_Resource_Manager application and the exploration of various JSS environments have shed light on the complexities and limitations inherent in applying DRL algorithms to scheduling problems. This research navigates through these challenges, offering insights into the nuanced performance of DRL models such as DQN, DDQN, Prioritized DDQN, and A2C across different scenarios, yet it also underscores the gaps between theoretical potential and practical application.

Our findings highlight the nuanced performance of DRL models, such as DQN, DDQN, Prioritized DDQN, and A2C, across different scheduling scenarios. Each model's effectiveness is closely tied to the specific characteristics of the JSS environment, underscoring the importance of tailored algorithmic approaches and environment-specific configurations. The introduction of supervised methods for environment evaluation, despite facing challenges, marks an innovative step towards validating the effectiveness of DRL models in scheduling contexts.

However, the research also uncovers significant challenges related to computational efficiency, memory limitations, and the necessity for prolonged training durations to achieve meaningful learning outcomes. These challenges indicate the need for further optimization of algorithmic implementations and training processes to enhance the practical applicability of DRL solutions in job scheduling.

As mentioned in the previous section, there are plenty possibilities for future research directions and improvements of the application. One possibility is to focus on addressing the computational challenges, exploring alternative reward structures, and investigating the scalability of DRL models to accommodate larger and more complex scheduling environments.

References

Domschke, W. (1997). *Produktionsplanung: Ablauforganisatorische Aspekte*. Berlin: Springer, 2 ed. überarbeitete und erweiterte Auflage.

Framinan, J. M., Leisten, R., & Ruiz García, R. (2014). *Manufacturing Scheduling Systems*. London: Springer London.

Gabel, T., & Riedmiller, M. (2008). Adaptive reactive job-shop scheduling with reinforcement learning agents. *International Journal of Information Technology and Intelligent Computing*.
URL https://ml.informatik.uni-freiburg.de/former/_media/publications/gr07.pdf

Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., & Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18)*, (pp. 3215–3222). New Orleans, LA, USA: AAAI.

Lawler, E. L., Lenstra, J. K., Rinnooy Kan, A. H., & Shmoys, D. B. (1993). Chapter 9 sequencing and scheduling: Algorithms and complexity. In *Logistics of Production and Inventory*, vol. 4 of *Handbooks in Operations Research and Management Science*, (pp. 445–522). Elsevier.
URL https://www.sciencedirect.com/science/article/pii/S0927050705801896

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Harley, T., Lillicrap, T. P., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, *v2*.

Riedmiller, S., & Riedmiller, M. (1999). A neural reinforcement learning approach to learn local dispatching policies in production scheduling. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, (p. 764–769). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Rottler, L., & Mahn, N. (2023). Forschungsprojekt a.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2016). Prioritized experience replay.

Simonini, T. (????). Advantage actor critic (a2c). https://huggingface.co/learn/deep-rl-course/unit6/advantage-actor-critic. Accessed: 2024-02-22.

Tassel, P., Gebser, M., & Schekotihin, K. (2021). A reinforcement learning environment for job-shop scheduling.

van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30.
URL https://doi.org/10.1609/aaai.v30i1.10295

Yoon, C. (2019). Understanding actor critic methods and a2c. *Towards Data Science*. Accessed: 2024-02-22.

Zhang, C., Song, W., Cao, Z., Zhang, J., Tan, P. S., & Chi, X. (2020). Learning to dispatch for job shop scheduling via deep reinforcement learning. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.) *Advances in Neural Information Processing Systems*, vol. 33, (pp. 1621–1632). Curran Associates, Inc.
URL https://proceedings.neurips.cc/paper/2020/file/11958dfee29b6709f48a9ba0387a2431-Paper.pdf