BERGISCHE
UNIVERSITÄT
WUPPERTAL

Fakultät IV - Mathematik und Informatik

Institute for Technologies and Management of Digital Transformation

Decision Transformers for solving Production Scheduling Problems via Reinforcement Learning

**Masterarbeit zur Erlangung
des akademischen Grades
Master of Science (M.Sc.)**

Gutachter:
Prof. Dr.-Ing. Tobias Meisen
Prof. Dr.-Ing. Dietmar Tutsch

Betreuer:
Constantin Waubert de Puiseau, M.Sc.

Vorgelegt am 18.12.2023 von:

Fabian Wolz
Bilker Allee 83
40219 Düsseldorf
fabianwolz@gmx.de
Matr.-Nr. 2113700

Abstract

The Job Shop Scheduling Problem (JSSP) is an NP-hard combinatorial optimization problem that deals with finding time-efficient schedules for the allocation of a set of tasks to a set of machines. Finding faster schedules for it can potentially lower expenses and decrease production times for industrial companies. This thesis investigates the usefulness of the Decision Transformer (DT), a novel offline Reinforcement Learning (RL) approach based on the transformer architecture, for solving the JSSP. The DT interprets the sequence of state-action-reward tuples in RL as a sequence modeling problem. By conditioning the model on so called, returns-to-go (rtg) the DT can autoregressively generate future actions that achieve the desired return. In this study, two DT models are trained on data created from two distinct origin-agents in order to solve the JSSP. One agent, the dispatching Decision Transformer (D-DT), creates schedules by predicting the next task that is to be dispatched. The second DT, named NeuroLS Decision Transformer (NLS-DT), predicts actions to control a local search for finding schedules. The research aims to analyze if the DT models are able to outperform their respective origin-agents and whether they learn their own strategy or just clone the behavior of their origin-agent. Additionally, the responsiveness of the DT models to the rtg conditioning is investigated. The models are evaluated by comparing their performance on a created test dataset and for the NLS-DT on five problem sizes of the established Taillard benchmark. The D-DT shows similar performance than its origin-agent but demonstrates that it uses a different strategy to achieve this. The NLS-DT shows notable improvement to its origin-agent for the 15x15 problem size, reducing its optimality gap by 2.79%. For the other problem sizes, the NLS-DT is at least competitive to the origin-agent or slightly outperforms it. Its behavior significantly deviates from that of its origin-agent. While the D-DT shows low responsiveness to the rtg in some scenarios, for the NLS-DT no significant influence of the rtg can be determined.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AI Artificial Intelligence

CEI Critical End Insert

CET Critical End Transpose

CI Confidence Intervall

CT Critical Transpose

D-DT Dispatching Decision Transformer

DT Decision Transformer

DRL Deep Reinforcement Learning

ECET Extended Critical End Transpose

GNN Graph Neural Network

GPT Generative Pre-Trained Transformer

JSSP Job Shop Scheduling Problem

LS Local Search

MDP Markov Decision Process

MLP Multi Layer Perceptron

NLS NeuroLS

NLS-DT NeuroLS Decision Transformer

PDR Priority Dispatching Rules

PPO Proximal Policy Optimization

PSP Production Scheduling Problem

RASCL Reinforced Adaptive Staircase Curriculum Learning

RL Reinforcement Learning

rtg returns-to-go

SPT Shortest Processing Time

TT Trajectory Transformer

1. Introduction

Effective production scheduling in manufacturing systems is an essential activity for industrial companies, playing a critical role in enabling them to remain competitive in consumer markets [1]. The Job Shop Scheduling Problem (JSSP), as one of the most prevalent Production Scheduling Problems (PSPs) [2], is an NP-hard combinatorial optimization problem that deals with finding optimal and time-efficient schedules for the allocation of a set of tasks to a set of machines. Faster production schedules lower expenses and decrease production times for industrial companies, as well as enabling them to respond more quickly to market demands [1]. Therefore, developing effective methods for solving JSSPs is of critical importance.

Traditional approaches for solving JSSPs in practice mainly focus on meta-heuristics or local search strategies [3][4]. Although these methods effectively produce schedules with reduced production times (makespans), they still exhibit large optimality gaps. This motivates research to find new and better approaches. In recent years Artificial Intelligence (AI) and especially Reinforcement Learning (RL) have been investigated in the scientific literature, as novel solution approaches for optimization problems and the JSSP specifically. Simultaneously the transformer architecture proposed by Vaswani et al. [5] has revolutionized various domains such as natural language processing [6] and image generation [7], leading to several state-of-the-art AI models. This motivates to investigate whether the transformer architecture is also able to similarly enhance performance in the field of RL and its application to the JSSP.

The Decision Transformer (DT) is a novel offline RL method based on the transformer architecture which abstracts the sequence of state-action-reward tuples, which are common for reinforcement learning, as a sequence modeling problem. This sequence of tuples can be compared to a sequence of words in language modelling with transformers. Instead of forecasting the next word, the DT uses the transformers ability to capture long-term dependencies within sequences in order to predict the next action of the RL agent. An important difference of the DT to classical reinforcement learning is the inclusion of a desired target reward denoted as returns-to-go (rtg) as an additional input to the model. The rtgs supplement the past states, actions and rewards. Thereby, as opposed to traditional RL agents, the DT does not try to choose an action that maximizes the cumulative reward, but an action that will most likely evoke the targeted reward. The method has demonstrated high success in offline reinforcement learning settings for Atari games and the OpenAi gym where it is able to significantly outperform the strategy that created the trajectories on which the DT has been trained [8]. The notable success of the DT architecture promises great potential for the JSSP if trained on datasets created from currently best performing RL agents.

The goal of this thesis is to investigate and demonstrate if and how well the DT architecture is able to solve the JSSP. In order to do that, the DT architecture is applied on two distinct RL-Agents. One utilizes a dispatching approach for schedule creation and a second named NeuroLS in [9] finds schedules by employing a local search approach. This aims to analyze if the DT is able to learn their behavior and potentially outperform these in creating schedules with lower makespan by conditioning them on the rtgs.

The primary objective is to investigate the effectiveness of the DT in comparison to origin-agents and analyze its responsiveness to the rtg, the following research questions shall be answered in the scope of this thesis:

1. Are the DT models superior to their respective origin-agents in terms of the makespan and if so, in which specific scenarios or instances do they exhibit improved performance?
2. Do the DT models learn their own strategy, or do they just clone the behavior of their respective origin-agent?
3. How does the learned strategy respond to differing rtg values during inference?

By addressing these research questions, this thesis contributes insights into the applicability of DT models in the domain of JSSP.

The structure of this thesis is as follows: First, the relevant foundations of the JSSP are explained, followed by an introduction to reinforcement learning and transformer architectures. Furthermore, the decision transformer model and the NeuroLS origin-agent is presented. The third chapter provides insight in current RL and transformer models for the JSSP, and introduces other applications and variations of the DT. Chapter four describes the method used to answer the research questions and provides technical details of the created models. The fifth chapter explains the conducted experiments and answers the research questions by presenting the experiment results. Finally, a conclusion of the thesis and recommendations for future research directions in the field of DTs in combination with the JSSP are given.

2. Foundations

2.1 Fundamentals of Production Scheduling Problems

2.1.1 Definition

In general, a scheduling problem can be defined as a problem of allocating limited resources to a set of tasks over time. The resources are necessary to accomplish the task and the tasks individually compete for the resources [10, p. 1][4, p. 3-9]. Since the number of possible solutions grows with the number of tasks and resources, scheduling problems belong to the class of combinatorial optimization problems.

Production scheduling problems are a subclass of the aforementioned problems that occur in industrial productions processes. Here the scarce resources are machines (or tools) designated for the execution of production tasks. Finding an optimal schedule has been proven to be NP-Hard which implies that the time for finding an optimal schedule grows at least polynomial with the size of the problem, rendering the computation time infeasible for larger problem sizes [11].

Formally, a Production Scheduling Problem can be defined by the following entities:

- A finite set of *machines*: $M = \{m_0, m_0..., m_m\}$ is
- A finite set of *jobs*: $J = \{j_0, j_0..., j_n\}$
- A finite set of *tasks*: $T = \{t_0, t_0..., t_k\}$

Furthermore, the following mappings are necessary to formally define a Production Scheduling problem:

- $f : T \to J$
- $g : T \to M$

f and g assign each task a job and a machine, respectively. Hereby, f is necessarily a bijective mapping, whereas the properties of g depend on further specifications of the specific problem.

2.1.2 The Job Shop Scheduling Problem

The Job Shop Scheduling Problem, in literature also referred to as $J||C_{max}$-Problem [4, p.28], is a distinct variant of the PSP considering so-called *Job Shops*. In a Job Shop environment of size $n \times m$, a total of n jobs needs to be sequenced across m machines. Each job has at maximum one task for each machine and each task needs to be processed on one specific machine. The sequence of the single tasks of a job is fixed in the Job Shop setting. This is also called routing or precedence constraint. However, the routing does not necessarily need to be identical for each job [12, p. 18][4, p. 23]. Each task has a positive integer runtime. Once a machine has started to process a task, it cannot be interrupted.

In this specific scheduling problem, the objective is to minimize the target criterion C_{max} referencing the overall makespan of a schedule. The makespan denotes the entire time required to complete a given set of jobs. Other objectives can be to minimize the flowtime or tardiness of the schedule. This work focuses on minimizing the makespan.

2.1.3 Representation Types of the JSSP

Two common representation types exist for visualizing a Job Shop Schedule [4]: Gantt charts and the disjunctive graph representation as explained in the following.

Disjunctive graph

For graphic visualization of the problem structure, the disjunctive graph G is commonly used [4, p. 36-42]. G is a triplet (V, C, D) with V being the set of nodes where each node represents one task of a job. C and D both are sets of arcs between nodes. C is the set of so-called conjunctive arcs defined as $C \subseteq \{(v_i, v_j) \in V \times V | v_i \neq v_j\}$ where each arc is directed and indicates the asymmetric precedence constraint between two jobs of a task. This means a node v_i must be processed before any node to which it has an outgoing conjunctive arc. The set of nodes V and the set of conjunctive arcs C together form the undirected conjunctive graph $G = (V, C)$ [4, p. 36-42].

D is the set disjunctive arcs defined as the initially symmetric relation $D \subseteq \{(v_i, v_j) \in (V \times V) \setminus \{v_{start}, v_{end}\} | v_i \neq v_j\}$. A disjunctive arc between two tasks, is an undirected arc between two tasks, where both tasks must be processed on the same machine. Since a finished schedule needs a unique processing sequence, each arc in D needs to be transformed to a directed arc to create a schedule. A *selection* $D' \subset D$ is a subset of D with the constraint that each arc in D is directed [4, p. 36-42].

The conjunctive graph G in combination with the *selection* D builds the directed Graph $G' = (V, D' \cup C)$, representing a complete operation sequence of the machines and therefore a complete schedule for the JSSP instance.The starting time of each task is now given by the length of the longest path from v_start to the corresponding task node. Therefore, the makespan of the schedule represented by the disjunctive graph is equivalent to the longest path from v_{start} to v_{end} [13]. According to Nowicki and Smutnicki [14], the longest path of a disjunctive graph can be broken down into critical arcs and critical blocks. A critical arc is an arc in the longest path that connects two jobs that are consecutively processed on the same machine. A critical block is characterized as a series of operations connected by the longest possible

chain of critical arcs within a single longest path [15]. Figure 2.1 shows an example of a disjunctive graph before the schedule is finished (undirected disjunctive arcs). Figure 2.2 shows the graph with the finished schedule (directed arcs).

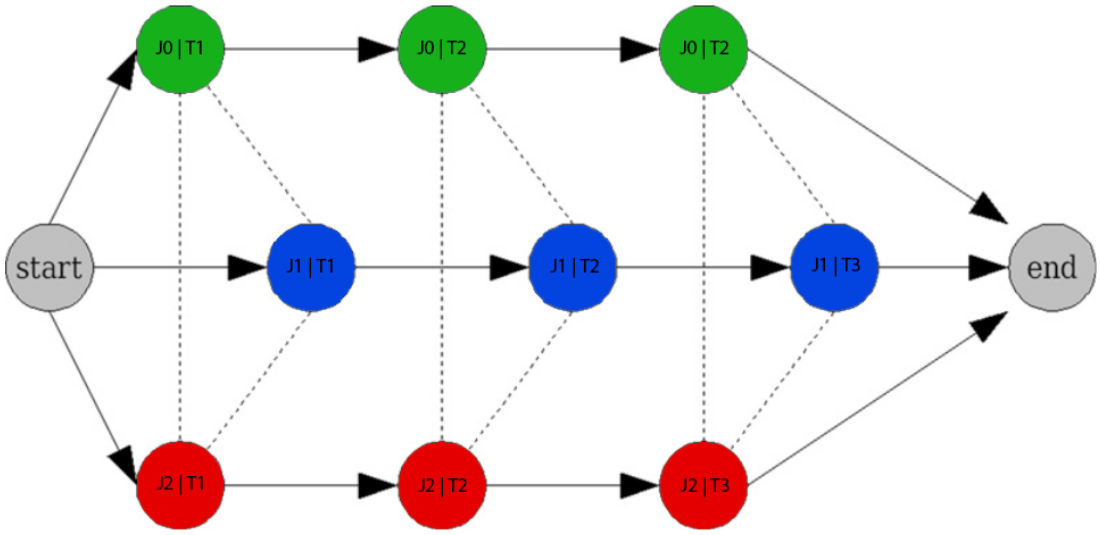

Figure 2.1: Disjunctive Graph

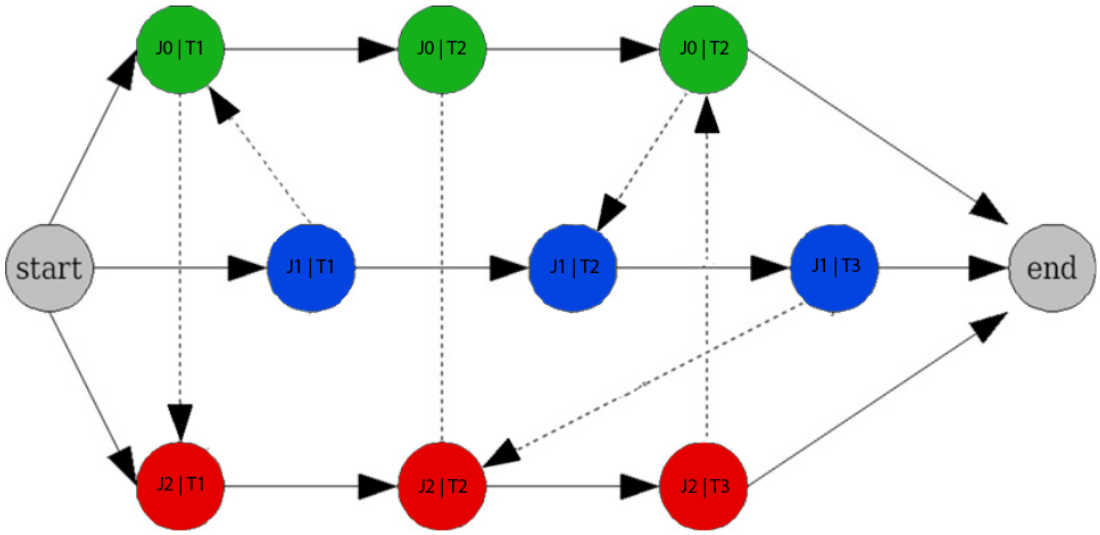

Figure 2.2: Disjunctive graph with finished machine sequence

Gantt chart

Another possible representation of a JSSP is a Gantt chart [4, p.44 - 46]. A Gantt chart represents the execution of tasks as bars on a time axis, whereby the length of the bars represents the execution time of the task. Time is displayed on the

horizontal axis, while the vertical axis is divided into rows, with each row assigned to a machine. A task set in a row means that it must be processed on that machine. For visualizing the relation of a task to its job, each job has a color or pattern that is applied to the blocks of the respective jobs. One advantage of the Gantt chart over the disjunctive graph representation is that the time intervals between the execution of the tasks can be read directly and do not have to be calculated based on the length of the weighted paths [4]. They also make it possible to directly see when which task starts, when it is ready and what tasks are executed simultaneously. Figure 2.3 shows the Gantt chart for the same JSSP instance as in the disjunctive graph in Figure 2.2.

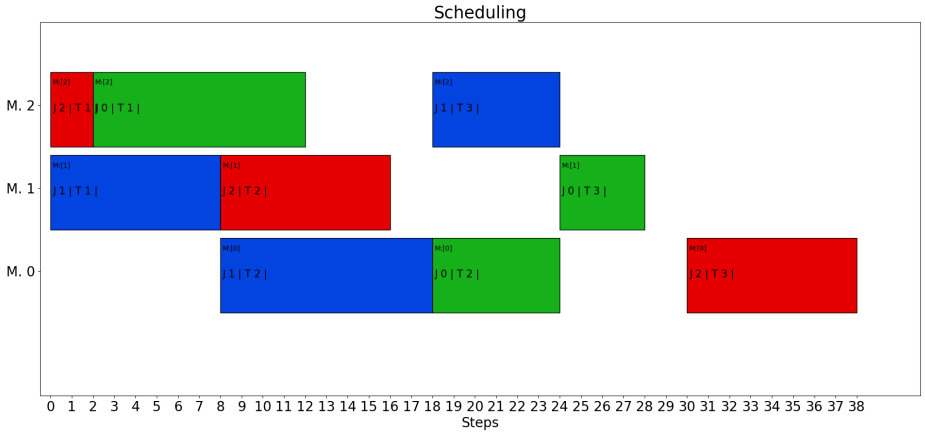

Figure 2.3: Gannt Chart for JSSP

2.1.4 Solution Approaches

Priority Dispatching Rules

Due to the computational complexity associated with finding a schedule with minimal makespan, particularly for larger problem sizes, it is common to use heuristics in real-world-scenarios. Heuristics are commonly much less computational complex because they do not perform extensive search in the search space. One kind of heuristic methods are *Priority Dispatching Rules (PDR)*. They determine the schedule by rules that assign priorities to jobs based on simple calculations of various criteria, such as processing times, due dates or remaining work in a job. Illustrative examples of such PDR include *Shortest Processing Time (SPT)* where the task with the lowest processing time is selected at each step, and the *Earliest Due Date (EDD)* method, which prioritizes tasks based on their closest due date [16].

Local Search

Another heuristic method which is not only used for JSSPs but combinatorial optimization problems in general, is the Local Search (LS)[17]. The fundamental idea of LS is to iteratively explore potential solutions $s \in S$ in the neighborhood of the current solution. The neighborhood $\mathcal{N}(s) \subseteq S$ is a set of solutions that slightly differ from the current solution. For a JSSP, for example, a neighbor $s' \in \mathcal{N}(s)$

of a schedule could be another schedule where one task is scheduled at a different time. Whether a solution is part of the neighborhood is defined by a neighborhood function $\mathcal{N}^{\varphi}$ where $\varphi \in \Phi$ is an operator like for example to change one task in the schedule s. Kuhpfahl et al. [15] analyze different neighborhood operators that consider JSSPs represented as disjunctive graph of which the following are of relevance for this work:

- **Critical Transpose (CT)** [18]: Creates the neighborhood by reversing one critical arc in the longest path of the disjunctive graph. However, this will only reduce the length of the longest path if the critical arc is positioned at the beginning or the end of a critical block. If the inverted arc is within the critical block, the starting times of the subsequent jobs stays unaltered. Therefore this would not reduce the length of the path [15].
- **Critical End Transpose (CET)** [14]: This operator adapts the CT operator by restricting the inversion to critical arcs positioned at the end or the beginning of a critical block [15].
- **Extended Critical End Transpose (ECET)** [15]: Is an extension of CET in which the neighborhood is created by inverting either the first or last arc or both simultaneously, instead of only the first or last arc like in CET.
- **Critical End Insert (CEI)** [19]: The CEI operator creates neighborhoods by shifting jobs within in a critical block, whereas the new position of the job is the most distanced position in the block, which doesn't change the feasibility of the schedule [15].

Figures 2.1.4 to 2.6 illustrate the CET, ECET, and CEI move. Here, the grey arrows illustrate a possible move to create a neighbor of the shown schedule.

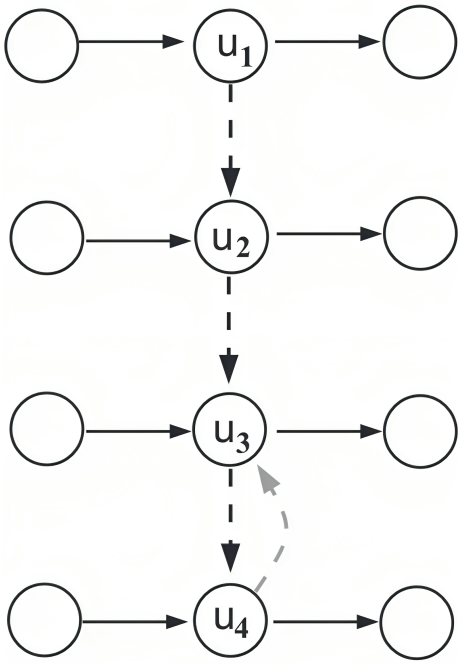

Figure 2.4: CET move. Source: [15]

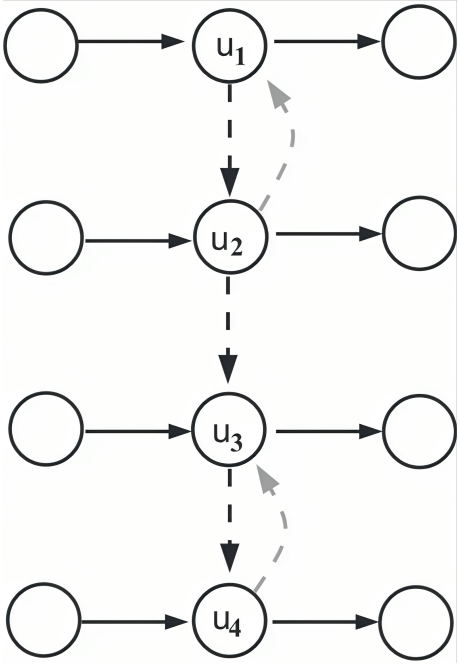

Figure 2.5: ECET move. Source: [15]

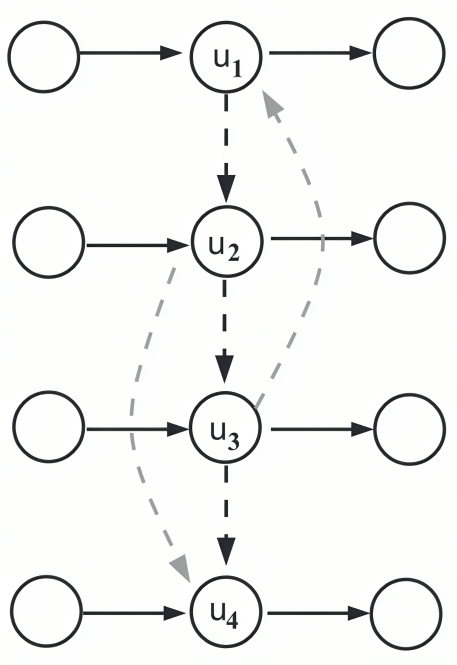

Figure 2.6: CEI move. Source: [15]

Each neighborhood has a local optimum $\hat{s}$ which satisfies $C_{max}(\hat{s}) \leq C_{max}(s) \forall s \in \mathcal{N}(\hat{s})$. Listing 2.1 illustrates a general LS procedure.

Listing 2.1 Local Search as in[9]

Input: cost function $\overline{C}_{max}$, solution s, neighborhood function $\mathcal{N}$,
acceptance rule **accept**, stopping rule **stop**
while stop(s) is False **do**
Find $s' \in \mathcal{N}(s)$ such that accept(s, s')
$s \leftarrow s'$
end while
return s

One of the basic LS techniques is a hill climbing approach which greedily accepts local changes to the current solution. The hill climbing approach goes along with the problem that the LS can easily get trapped in an unfavorable local optima. To circumvent this problem, LS is usually employed in combination with so called meta-heuristics, which allow escaping from the local optima by balancing exploration and exploitation of the search space. Different meta-heuristics have different intervention points at which it can be decided how the local search can escape local optima and be guided towards good solutions. Falkner et al. [9] defines the following three intervention points for meta-heuristics that have considerable influence on the local search.

1. **Acceptance:** Acceptance is about whether a candidate solution s' in a local search step will be accepted as the new current solution or not. The greedy hill climbing approach for example only accepts improving solutions, while other heuristics may also accept non-improving solutions with a given probability.

2. **Neighborhood:** The second possible choice a meta-heuristic can make in order to influence the local search is the selection of the neighborhood operator φ which makes it possible to define the neighborhood in different ways at each search step.

3. **Perturbation:** The third intervention strategy is to employ a perturbation to the current solution. Perturbations escape local optima by simply moving to completely different regions in the search space. The perturbation itself can take various forms, such as a fresh start from a new stochastically generated initial solution, a random reshuffling of parts of the current solution, or a randomized sequence of node exchanges. Deciding when to employ such a perturbation typically relies on a predefined number of steps without observing any improvement.

Listing 2.2 shows the LS-approach when using meta-heuristics.

Listing 2.2 Local Search with meta-heuristic

Input: Solution space S, cost function f, stopping criterion

1: Construct initial solution s
2: **while** not stopping criterion **do**
3: $s \leftarrow$ perturb(S,s) ▷ Decide if to perturb
4: $\mathcal{N} \leftarrow$ GetNeighborhood(S,s) ▷ Define search Neighborhood
5: $s \leftarrow$ LocalSearch($C_{max}, s, \mathcal{N}, accept, stop$) ▷ Execute local search
6: **end while**
7: **return** s

Another example of a heuristic method, which is specifically tailored for the JSSP is the *Shifting Bottleneck heuristic* [20, p. 193].

Optimal solutions

Even though it is computationally expensive, there exist exact methods to solve a JSSP. With constraint programming, for example, an optimal schedule can be found for a JSSP by representing it as a constraint optimization problem [3]. Two state-of-the art constraint solvers are the open source solution OR-Tools from Google [21] and the proprietary IBM solver CP-Optimizer [22].

(Deep) Reinforcement learning

In addition the named approaches, RL and especially Deep-RL have become promising solution approaches for the JSSP in recent years [23]. For example, RL techniques, including Deep-RL, offer solutions to the JSSP by letting a RL-agent dispatch tasks in a step-by-step manner providing end-to-end solutions. This can be accomplished by either letting the agent select a common PDR or by letting it learn its own PDRs. Other RL-approches aim to optimize local search strategies for finding better schedules.

2.1.5 Taillard Benchmark

The Taillard benchmark is a frequently used set of benchmark instances for the JSSP [24]. In total, the benchmark consists of 80 instances, with 10 instances for each problem size, where the problem sizes are 15x15, 20x15, 20x20, 30x15, 30x20, 50x15 50x20, 100x20. The benchmark is widely used to assess the performance of different JSSP solution approaches. Thereby it is common to compare the achieved mean makespan per problem size by the current best known upper bounds of the mean makespan per problem size. Beside the absolute value of the mean makespan sometimes an optimality gap is provided which states how far away the achieved makespan is from the optimal value in percent. The optimality gap is calculated by 2.1. The current upper bound of the mean makespans are available in

$$\text{Optimality Gap (\%)} = \left(\frac{\text{achieved makespan} - \text{upper bound}}{\text{upper bound}}\right) * 100 \tag{2.1}$$

2.2 Reinforcement Learning

2.2.1 Core Concept

The following paragraph is mainly based on [25]. The fundamental idea behind RL is to learn what to do in specific situations, or more generally, how to map situations to actions that maximize a numerical reward [25, p. 2]. The core elements of RL are an *agent* and an *environment*. The agent interacts with the environment in form of *actions* executed in discrete time steps. It perceives the environment, or a partial observation of it and based on this observation, chooses what action is to take. Whenever the agent performs an action, the environment is influenced and may change. Based on the changed environment, the agent receives a *reward* in form of a numerical value $r_t \in \mathbb{R}$ indicating how good or bad the state of the environment becomes after the respective action was taken (see Figure 2.7). Positive rewards indicate good states, encouraging the agent to repeat similar actions in similar situations. Negative rewards, conversely, signal bad states, guiding the agent to avoid such actions in the future. The objective of the agent is to maximize the cumulative reward in the long run.

Further entities of a RL system are a state, an action space, a policy, a reward and a trajectory.

A state s_t represents the state of the environment at time t. An observation o_t is a partial description of s_t that may omit some information. o_t is what the agent perceives at time t and s_t is the complete description of the world. States and observations are usually represented by real-valued vectors or matrices [26]. The state of an Atari game can, for example, be represented by a matrix of the pixels of one or more frames of a situation in a game.

The action space is the set of actions that an agent can perform in the environment. In general, a distinction is made between discrete and continuous action spaces. In discrete action spaces, the agent's choice is limited to a finite set of actions. In continuous action spaces, the actions are represented by real-valued vectors [26].

The policy is the core of the agent and defines its behavior at a given time. It is a function, mapping from an observation to an action the agent should execute, and

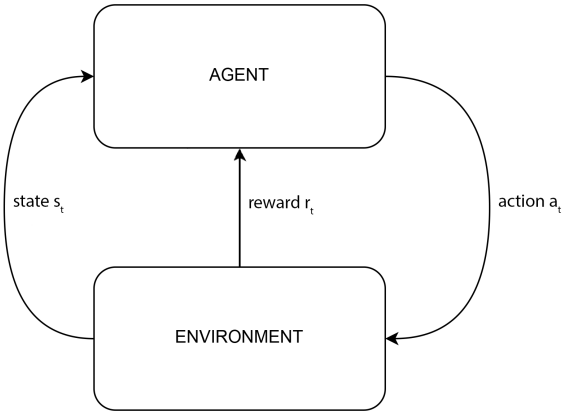

Figure 2.7: Agent interaction with environment

thus completely describes the behavior of the agent. The policy can be deterministic or stochastic and can range from a simple lookup table to extensive computations. The goal of a RL system is thereby to learn a policy which maximizes the total return. During the learning process of the agent the policy is adjusted based on the received reward If the action selected by the policy yields a poor return, the policy may be adapted to select a different action for similar observations in the future.

A trajectory is a sequence of states, actions and rewards which provides an overview of the behavior of the agent and the development of the environment. It is typically is denoted as:

$$\tau = (s_0, a_0, r_1, s_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T) \tag{2.2}$$

Furthermore, RL is fundamentally built upon the concepts and framework of a Markov Decision Process (MDP) as the previously introduced notions generally hold for MDPs. MDPs represent an idealized, mathematical formulation of the RL problem, for which clear theoretical statements can be made [25]. One key characteristic of a MDP, that by the intrinsic connection between MDPs and RL also holds for RL, is the Markov property. The Markov property fundamentally states that the future state of the environment only depends on the current state and the action taken, not at all on previous states and actions [25].

2.2.2 Online and Offline Reinforcement Learning

The previously described RL approach has as its main characteristic that the agent is directly interacting with the environment in the learning phase. This implies that the agent either needs to be trained in a real-world environment or in a simulation. Both have their own complications. A simulation can be hard to build properly and training in real-world scenarios can be quite expensive and insecure. Offline RL solves these problems as it can be seen as a data-driven formulation of the RL problem [27]. Here, no direct interaction between the agent and the environment is happening during the learning phase. In contrast, the RL system learns from a static dataset of transitions $D = \{(s_t, a_t, s_{t+1}, r_t)\}$ collected from other agents or human interactions. Conclusively, in offline RL the goal is to derive a sufficient

understanding of the environment from a fixed dataset and from that learn a policy which maximizes the return when interacting with the real world scenario. By this notion, offline RL resembles supervised learning. In consequence of using a fixed dataset, the agent is not able to explore the environment on its own, which is a disadvantage to classic RL [27].

2.3 Transformer Architecture

2.3.1 Transformer

The Transformer architecture proposed by Vaswani et al. [5] is a neural network architecture aimed at learning context in sequences by identifying relations in sequential data [28]. They were initially introduced for translation tasks but have been successfully applied to other domains like for example image modelling [7].

The original Transformer architecture follows an encoder-decoder structure as shown in figure 2.8. By example of an English-German translator, this architecture can be explained as follows: The encoder processes an input sequence, such as an English sentence consisting of tokens $(x_1, ...x_n)$ into a continuous representation $z = (z_1, ...z_n)$ and the decoder processes z into an output sequence $(y_1, ...y_n)$ which can be the German translation of the given sentence. The encoder part of the transformer consist of multiple (six in the original version) encoder blocks. Each of the encoder blocks consists of two sublayers. First a multi-headed self-attention mechanism and second a fully connected feed-forward network. Self-attention is a mechanism that allows each token in the input sequence to attend each other token in the sequence to a different degree. This helps the model to identify relations in the sequence. To illustrate the self-attention mechanism with an example from the translation domain, the following sentence can be considered: *"The animal didn't cross the street because it was too tired"*. Considering the word *"it"* in the sentence, the self-attention enables the model to learn that *"it"* refers to *"animal"* by assigning appropriate attention scores to each word in relation to *"it"*. Around each of both sub-modules a residual connection is employed and finally completed by a layer normalization.

A decoder block is build slightly different, since it adds a third sub-module called *masked-multihead-attention*. The masked-multihead-attention ensures that the decoder works autoregressively. Autoregressive in this context means that the prediction of a next word or token is based on a sequence of previously from the model itself predicted words or tokens. So the masked multih-ead attention attends to all already predicted tokens. This is followed by a *encoder-decoder-attention* which allows each already predicted token to apply attention to each token in the input sequence. The output of this is then, as in the encoder, separately for each position processed by a feed forward network. The output of the decoder is a floats vector. The output is then put through a linear transformation resulting in a logits vector of the vocabulary size (the number of all different tokens). Finally, a softmax layer is applied to the logits vector to get a probability for each token in the vocabulary. The token with the highest probability will then be chosen as the next prediction.

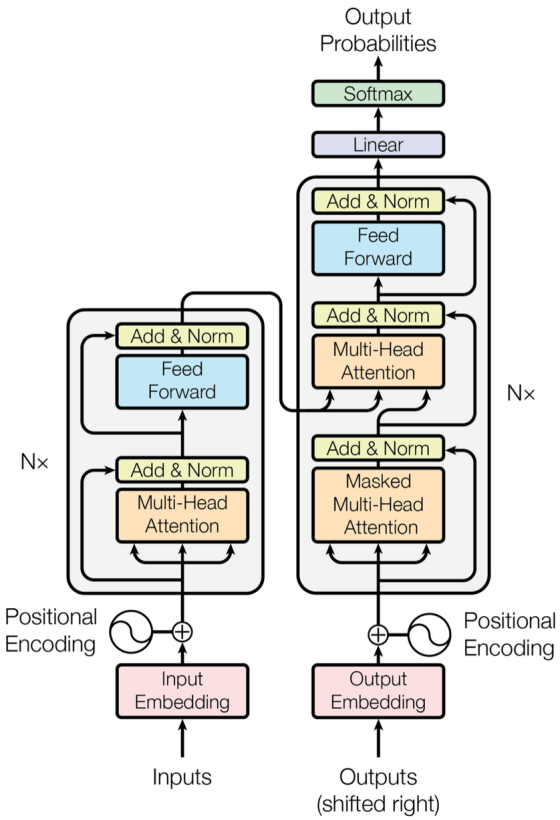

Figure 2.8: Transformer architecture. Source: [5]

2.3.2 Generative Pre-Trained Transformer

Generative Pre-Trained Transformer (GPT) is an autoregressive language model which is based on the decoder architecture of the transformer [29][30]. Thus, a sequence of tokens is not generated all at once but token by token where each generated token is based on the tokens predicted before.

GPT is learned in a self-supervised manner with an unsupervised corpus of tokens $\mathcal{U}$ to maximize the following likelihood:

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-\mathcal{K}}, ..., u_{i-1}; \theta) \tag{2.3}$$

Here $\mathcal{K}$ represents the context length, meaning the number of previous tokens taken into account for predicting the next token. The conditional probability P is modelled using a neural network with parameters θ, which is trained with stochastic gradient descent. The learning is self-supervised in that way, that the unlabeled corpus of token have labels inherent in their sequence through the natural ordering of language. This means, for a sequence of tokens $(u_i, u_{i+1}...u_n)$, u_{i+1} can be taken as a label for u_i and u_n as a label for the whole previous sequence $(u_i, u_{i+1}...u_{n-1})$.

GPT has been proposed in four diff rent versions so far. While GPT-1 was fine-tuned in a supervised manner on specific tasks like question answering or text similarity [29], the later versions do not need supervised fine-tuning but are able to learn the task in an unsupervised manner as well by adjusting the conditional probability to contain the task: $P(u_i | u_{i-\mathcal{K}}, ..., u_{i-1}, task)$ [6][30].

2.4 Decision Transformer

The Decision Transformer is an architecture for offline-RL which abstracts RL as a sequence-modelling problem [31]. It is an autoregressive model which uses the GPT architecture for generative trajectory modeling in order to solve RL-problems.

The fundamental idea behind DT is, to generate actions a_t of an RL-agent based on the sequence of the last $\mathcal{K}$ *states, actions and returns-to-go* where $\mathcal{K}$ is denoted as the context-length. The DT is trained on sequences of length $\mathcal{K}$ extracted from a dataset of offline trajectories collected from an arbitrary policy. The reward in the DT is modeled with the so called *return-to-go* formalized as, $\hat{R} = \sum_{t'=t}^{T} r_{t'}$ which represents the sum of the (desired) future rewards. This enables the DT to generate actions during inference based on a *future* desired return instead of past rewards. The model learns to reach the specified *return-to-go* and thus can outperform the underlying policy of the offline-RL dataset, by passing an optimal *return-to-go*. A trajectory is thus represented in the following way:

$$\tau = (\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T, a_T) \tag{2.4}$$

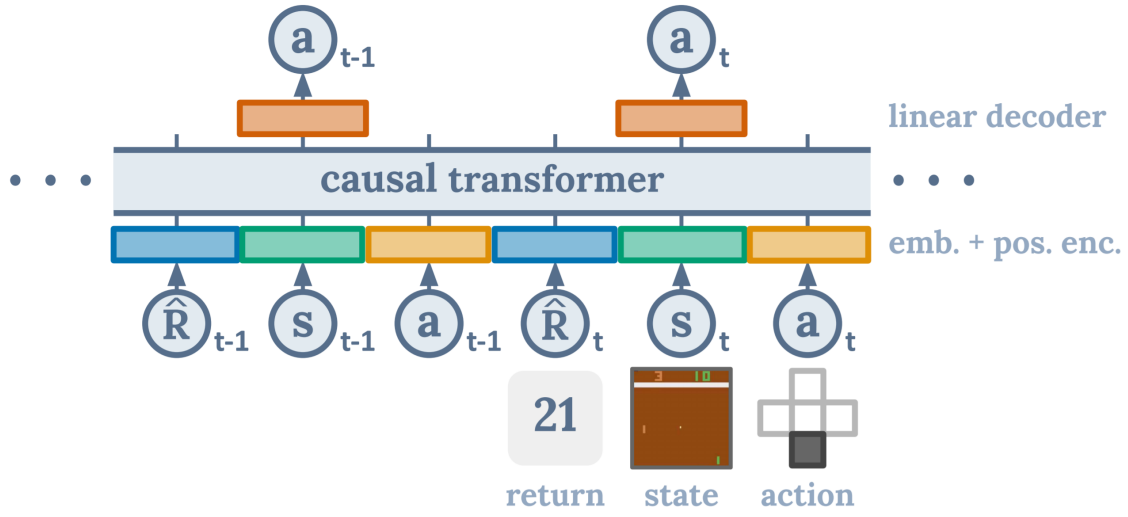

Figure 2.9: Decision Transformer Architecture. Source: [8]

The architecture of the Decision Transformer is shown in figure 2.9 In order to generate actions, the DT receives the last $\mathcal{K}$ time steps. In total the generation is based on $3\mathcal{K}$ tokens, meaning the last $\mathcal{K}$ *actions*, *states* and *rewards*. An embedding is created for the three tokens by learning a separate linear layer for each of the respective tokens. Additionally, a positional embedding based on the current time step is added to each of the token embeddings. Then the three resulting embeddings are concatenated. The concatenated embeddings is ultimately processed by a GPT model, to generate the next action. Assuming we are at state s_t and want to predict action a_t during inference to transfer to state s_{t+1}. Then the prediction of a_t will be based on the following sequence: $(\hat{R}_{t-k}, s_{t-k}, a_{t-k}, ..., \hat{R}_t, s_t)$. After transitioning to s_t+1 $\hat{R}_{t+1}$ is determined by: $\hat{R}_{t+1} = \hat{R}_t - r_t$ with r_t being the reward of transitioning from s_t to s_{t+1}. During training a prediction is made for each of the embeddings in sequence so the loss is always calculated for $\mathcal{K}$ predictions per sequence.

2.5 NeuroLS

2.5.1 Concept

NeuroLS is a new approach to perform a local search with meta-heuristics introduced by Falkner et. al [9] which defines meta-heuristics as a MDPin order to control the heuristic decisions with a RL agent. This means that the decision about acceptance of a new solution, the neighborhood operator and perturbation are no longer made by meta-heuristics, but by an RL-agent. The original paper proposes NeuroLS for the JSSP and the Capacitated Vehicle Routing Problem. Since this thesis doesn't consider the latter, the following explanations only provide a summary on the method tailored to the JSSP.

In order to control the heuristic decisions with an RL agent, it is necessary to define the RL-entities *States*, *actions*, *rewards* and the *policy* for the problem. The authors of NeuroLS defines these in the following manner [9]:

- **States:** In NeuroLS the state s_t of a JSSP at time step t is represented as a combination of the solution s at time step t along with 1) its cost $C_{max}(s)$, 2) the cost $C_{max}(\hat{s})$ of the best solution discovered up to that point, 3) the last acceptance decision, 4) the last operator applied, 5) the current time step t, 6) the count of LS steps without any enhancements, and 7) the number of perturbations or restarts so far.

- **Actions:** For the actions, they propose three different action sets with which they learn different policies.

  1. *Acceptance* decisions: a boolean decision whether the last LS steps is accepted or not
  $$A_A := 0, 1, \tag{2.5}$$

  2. *Acceptance-Neighborhood* decisions: The tuple of both the decision to accept the last move and the set of possible operators $\varphi \in \Phi$ to be utilized in the subsequent step, where the operator can be one of the four described in 2.1.4
  $$A_{AN} := 0, 1 \times \Phi \tag{2.6}$$

  3. *Acceptance-Neighborhood-Pertubation* decisions: The tuple of acceptance and the intersection of the operators $\varphi \in \Phi$ and perturbations $\psi \in \Psi$
  $$A_{ANP} := 0, 1 \times \{\Phi \cup \Psi\} \tag{2.7}$$

Rewards

The reward r from step s_t in s_{t+1} in the NeuroLS environment is defined as the relative improvement of the last LS step compared to the best found solution at timestep t. To prevent negative rewards, the received reward is constrained to be no less than zero.

$$r_t = max(C_{max}(\hat{s}_t) - C_{max}(s_{t+1}), 0) \tag{2.8}$$

Policy

Deep Q-Learning is an RL-algorithm which uses deep neural networks to learn and approximate the so called Q-function which is sometimes also referred to as state-action value function. The Q-function yields an estimate of the expected cumulative reward that the agent would receive when it performs an action in a given state and follows the current policy thereafter [32]. The highest Q-Value of each state-action-pair for a given state and all possible actions in that state, is thus the best possible action for the current policy. Falkner et al. [9] parameterize the policy π_θ by a softmax over the corresponding Q-function $Q_\theta(s_t, a_t)$. Q is represented by a Graph Neural Network (GNN) based encoder-decoder model with trainable parameters θ:

$$\pi_\theta(a_t \mid s_t) = \frac{\exp(Q_e(s_t, a_t))}{\sum_{\mathcal{A}} \exp(Q_e(s_t, a))} \tag{2.9}$$

According to Sanchez et al. [33] a GNN is a special kind of neural network, specifically designed to work with data structures that can be represented as graphs. The key concept of GNNs is the use of so-called message propagation, where nodes iteratively, per layer of the network, update their respective feature vector representation, by aggregating information from its neighbors feature vectors. The way of aggregating information of the feature vectors can differ depending on the kind of GNN used.

2.5.2 Model Architecture

The NeuroLS architecture is composed of an encoder, an aggregator and a decoder part. The full architecture is show in Fig. 2.10 and explained in the following subsection.

Encoder

The encoder makes use of the disjunctive graph representation (see:2.1.3) of a respective JSSP-instance and leverages it by using a so called *1-GNN* architecture proposed by [34]. In the first layer the initial feature vector $h_i^{(0)}$ is created by feeding the node features x_i into an Multi Layer Perceptron (MLP): $\mathbb{R}^{in} \rightarrow \mathbb{R}^{emb}$

$$h_i^{(0)} = MLP^{(0)}(x_i). \tag{2.10}$$

Then a GNN of the conjunctive graph with L^{conj} layers is created (called static GNN in the original paper). This is followed by a GNN of the graph with disjunctive arcs, with L^{disj} layers (called dynamic GNN) representing the machine groupings. After this, a last GNN layer, which also uses the conjunctive arcs, is applied to merge the conjunctive graph with the disjunctive arcs. At the end of the GNNs another $MLP^{(L)}$ is placed which finishes the encoder by creating the node embeddings ω_i^{node}. One GNN-Layer is defined as:

$$h_i^{(l)} = GNN^{(l)}(h_i^{(l-1)}) = \sigma\left(MLP_1^{(l)}(h_i^{(l-1)}) + MLP_2^{(l)}\left(\sum_{j \in \mathcal{H}(i)} e_{j,i} \cdot h_j^{(l-1)}\right)\right) \tag{2.11}$$

here $h_i^{(l-1)} \in \mathbb{R}^{(1 \times d_{emb})}$ is the latent feature embedding of node i at the previous layer $l-1$. $\mathcal{H}(i)$ represents the 1-hop graph neighborhood and $e_{i,j} \in \mathbb{R}$ is the edge feature for the edge between nodes i and j. For the multi layer perceptrons MLP_1 and MLP_2 the following projection applies $R^{d_{emb}} \rightarrow R^{d_{emb}}$. σ is a GELU nonlinearity.

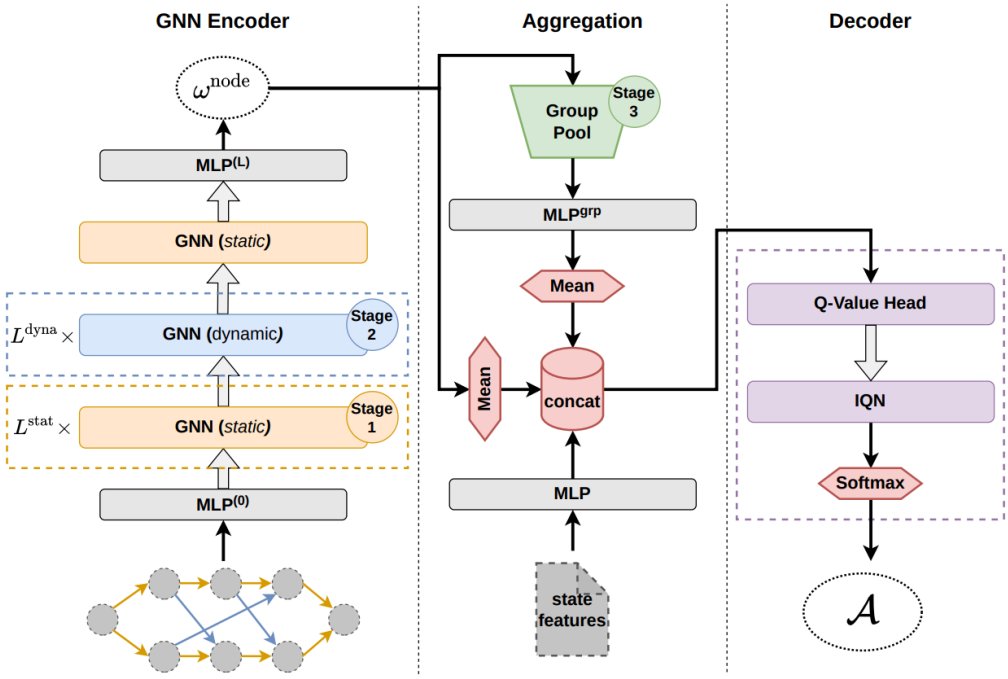

Figure 2.10: NeuroLS architecture. Source: [9]

Aggregation

The aggregation step takes the ω_i^{node} embeddings, resulting from the encoder and groups them according to the machine the respective node needs to be processed on. This creates the group embedding ω_{k}^{grp}. The grouping is done by feeding ω_{i}^{node} through another MLP:

$$\omega_{\text{k}}^{grp} = \text{MLP}\left(\left[\text{MAX}(\omega_i^{node}|i \in \mathcal{M}); \text{MEAN}(\omega_i^{node}|i \in \mathcal{M}_k)\right]\right) \tag{2.12}$$

The max and mean pooling are achieving a dimensionality reduction $\mathbb{R}^{N\times d_{emb}} \rightarrow \mathbb{R}^{K\times d_{emb}}$ of the embeddings. In equation 2.12 $\mathcal{M}_k$ refers to the K-th machine and ";" represents a concatenation operation in the embedding dimension d_{emb}. Finally, the aggregator takes the output of the encoder and first combines it with the seven state features and then projects it by a linear layer to create a final feature vector $\omega^{feat} \in \mathbb{R}^{emb}$.

Decoder

The decoder aggregates the node embeddings $\omega_{\text{node}} \in \mathbb{R}^{N\times d^{emb}}$ and group embeddings $\omega_{\text{k}}^{grp} \in \mathbb{R}^{K\times d^{emb}}$ by computing the mean over the node and group dimensions. These are concatenated with the feature embedding $\omega_{\text{feat}} \in \mathbb{R}^{emb}$. This is finally processed by final two-layer MLP regression head $\mathbb{R}^{3\times\text{demb}} \rightarrow \mathbb{R}^{|\mathcal{A}|}$, which produces the value predictions for the Q-function.

3. State of the art

The following chapter will first give an overview of the currently best performing reinforcement learning based models for solving the JSSP. This is followed by a presentation of models that explicitly use the transformer architecture to solve the JSSP. Finally, an overview of the current literature on the DT architecture is given by presenting use cases of different domains and variations of the decision transformer.

3.1 Reinforcement Learning Models for Solving the JSSP

A literature review of current models for the JSSP identified the following three as the currently best performing RL-based approaches, beside the NeuroLS model.

Zhang et. al [35] propose an end-to-end Deep Reinforcement Learning (DRL) agent that utilizes Proximal Policy Optimization (PPO) to learn Priority Dispatching Rules. The authors leverage the disjunctive graph representation of a JSSP to parameterize the policy with a size-agnostic *Graph Isomorphism Network*, a type of GNN. This structure enables the model to generalize to larger problem sizes when trained on smaller problems. They apply the model to different benchmarks where it outperforms common PDRs like Shortest Processing Time (SPT).

ScheduleNet [36] is another approach for solving scheduling problems with generalization ability between various kinds and sizes of scheduling problems. It is a multi-agent approach that considers each machine as a separate agent, with a decentralized policy that is shared among the agents. To solve JSSPs they propose a graph representation that differs from the disjunctive graph representation. The so called *agent-task-graph* has different kinds of nodes so that each task and each machine is represented by a node. The nodes have features indicating their current state like a machine being idle or a job being processable. Edges between nodes indicate whether the destination node is processable by the source node. The graph is embedded at each step with a so called type-aware-graph-attention mechanism. The policy is learned with a variant of PPO. Their approach is trained for the JSSP on random instances of size $(n \times m)$ where n and m are sampled from uniform distributions $U(7, 14)$ and $U(2, 5)$ respectively. The model trained on the random instances is able to outperform the model of Zhang et. al [35] on all taillard benchmark instances with regard to the average optimality gap per problem size.

Iklassov et. al [37] currently hold the best performing, reinforcement learning based model for solving the JSSP that was found in the literature research. They propose a new *Reinforced Adaptive Staircase Curriculum Learning (RASCL)* strategy which employ, as the name suggests, curriculum learning to better generalize dispatching rules on the JSSP. Curriculum Learning is a technique where the model is trained on

increasingly complex instances [37]. RASCL dynamically adapts the complexity level throughout the learning process to revisit instances with the poorest performance. Table 3.1 compares the performance of the three introduced models on the Taillard benchmark.

Instances		ScheduleNet	Zhang et. al[35]	RASCL
15X15	$\overline{C}_{max}$	1417	1547	1339
	Opt. gap	(15.31%)	(25.96%)	(9.02%)
20X15	$\overline{C}_{max}$	1630	1774	1509
	Opt. gap	(19.43%)	(30.03%)	(10.58%)
20X20	$\overline{C}_{max}$	1896	2128	1793
	Opt. gap	(17.23%)	(31.61%)	(10.87%)
30X15	$\overline{C}_{max}$	2129	2378	2038
	Opt. gap	(18.95%)	(33.0%)	(13.98%)
30X20	$\overline{C}_{max}$	2411	2603	2261
	Opt. gap	(23.75%)	(33.62%)	(16.09%)
50X15	$\overline{C}_{max}$	3104	3393	3030
	Opt. gap	(11.91%)	(22.38%)	(9.32%)
50X20	$\overline{C}_{max}$	3229	3593	3125
	Opt. gap	(13.55%)	(26.51%)	(9.89%)
100X20	$\overline{C}_{max}$	5723	6097	5578
	Opt. gap	(6.67%)	(13.61%)	(3.96%)

Table 3.1: Performance of current RL models on the Taillard benchmark.

3.2 Transformers for the JSSP

The proposed thesis is not the first work that tries to leverage the transformer architecture for solving the JSSP. Bonetta et. al. [38] treats the JSSP as a sequence-to-sequence process inspired by natural language models. They propose a DRL approach using an encoder-decoder architecture. The encoder receives, as input, a sequence of all tasks with their respective machines and processing times and creates an embedding of these using multi-headed-self-attention. The Decoder receives a masked representation of the embedding and generates a schedule in form of a sequence of jobs. Here, the masking ensures that only valid jobs can be scheduled. This is achieved by two separate maskings, where one masks out all jobs which were already processed, and the second masking ensures that no precedence constraint is violated, or no machine is multiply occupied. Their proposed model greatly outperforms traditional dispatching rules and is superior in performance in 6x6 and 10x10 problems, compared to the model by Zhang et. al [35].

Linlin et. al [39] propose an end-to-end DRL method using PPO. A model based on an encoder only transformer architecture, incorporating three identical attention layers, is created to train the actor. The trained model generates sequential

decision actions as the scheduling solution. The actor network receives a sequence of all unprocessed jobs and leverages the attention mechanism to assign attention to the input jobs. The job with the most attention will be processed next. Their architecture is superior to three common PDRs.

3.3 Decision Transformers Applications

The decision transformer architecture has been applied to many different domains, after the initial paper used it for Atari and musjocoGym.

Zhang et al. [40] explore the application of the DT in medical contexts with continuous time decision-making. Trained on historical and simulated patient data, the DT suggests optimal timings for visits and treatment plans, demonstrating potential to enhance patient health and prolong life. ChipFormer [41] is a DT model trained for chip placement which is a constraint optimization problem with the goal of placing circuit modules on a physical chip to minimize wirelength. It is able to outperform other state-of-the art model in that field and therefore underlines the potential of the DT architecture for optimization problems. Other domains the DT has been successfully applied to are active object detection for robots [42], recommender systems [43].

Due to its success, many variations of the decision transformer architecture have been developed. These include the *Graph Decision Transformer*, the *Elastic Decision Transformer* [44] and the *Hierarchical Decision Transformer*.

Graph Decision Transformer [45], is an adaption of the Decision Transformer architecture which models the input sequence into a causal graph to capture potential dependencies between different concepts and simplify temporal and causal relationship learning.

The Elastic Decision Transformer improves the DT by enabling it with so-called *trajectory stitching*. Stitching is a model-based data augmentation method for offline-RL that enables to "stitch"-together high value regions of different suboptimal trajectories, to improve the quality of the historical trajectories [46]. The Elastic Decision Transformer uses this in the action inference step, in combination with estimating the optimal context length at each step, allowing it to switch to a better trajectory by forgetting unsuccessful steps [44].

Hierarchical Decision Transformer is an approach consisting of two individual DT-models, named *High-Level-Mechanism* and *Low-Level-Controller*. The Low-Level-Controller is a *goal conditioned* descision transformer. This means that instead of the regular token sequence of states, actions and rtgs, which is used in the original DT, the rtgs are replaced by *sub-goals*. Sub-goals are defined, in relation to the current state, as later states in the trajectory that are highly valuable for the agent to reach. These sub-goals mark milestones in the trajectory, and reaching them sequentially indicates that it is highly probable that the agent successfully performs the task. The sub-goals have the same purpose as the rtg namely to guide the agent through the task. Therefore, it is possible to completely replace the rtgs. This has two advantages. On the one hand it is not necessary to know the best possible achievable reward in advance, and on the other it can replace *null* rewards in sparse reward tasks, to improve the learning process. Since the sub-goals are not explicitly

present in the data, they need to be inferred from it. The purpose of the High-Level Mechanism is to predict the next sub-goal based on a sequence of previous states and sub-goals, and then provide the predicted sub-goals to the Low-Level Controller. Thereby, the High-Level Mechanism is guiding the Low-Level controller through the task.

Concurrently to the development of the decision transformer, the Trajectory Transformer (TT) has been developed [47]. Trajectory Transformer is another architecture which abstracts RL as a sequence modelling problem. Like DT it uses a GPT model, but instead of conditioning on specified returns-to-go, TT is able to condition on a specific goal state. TT uses beam search as a planning algorithm. Beam search is commonly used in language modeling, to find the most likely sequence of words or tokens given a model and input. By using beam search, TT is able to model distributions over trajectories and find action sequences that maximize the reward. They apply the TT only to continuous action spaces in the mujoco environment. TT achieves higher returns in the Hopper and Walker2d environments on expert datasets and similar returns to the DT in medium datasets.

The emergence of the variation of the DT underlines its potential and its relevance in the field of offline RL. Particularly, the promising performance in the chip placement optimization task further motivates the application of the DT architecture to the JSSP and combinatorial optimization problems in general. Although the other variation are promising approaches as well, this work focuses on the application of the foundational DT architecture to the JSSP.

4. Methodology

4.1 Research Design

To investigate whether the Decision Transformer architecture is able to outperform current approaches, two applications of the DT are implemented and evaluated through several experiments. The first application, called the Dispatching Decision Transformer, aims to use the DT for dispatching JSSP schedules from one task to the next. The second application, referred to as NeuroLS Decision Transformer, aims to learn the DT to guide the heuristic decisions of a local search approach, and thus does not generate schedules one task at a time, but rather searches for optimal schedules. The general approach for both applications, is to train the DT on trajectories created from already trained *origin-agents* in a supervised manner, in order to decrease the makespans achieved by the origin-agents (see Figure 4.1). This potential makespan decrease is hypothesized to emerge by providing the learned DT a lower, near optimal, return-to-go. Various experiments are conducted with each application, to investigate how the DT-Models create the schedules and what makespans they are able to achieve.

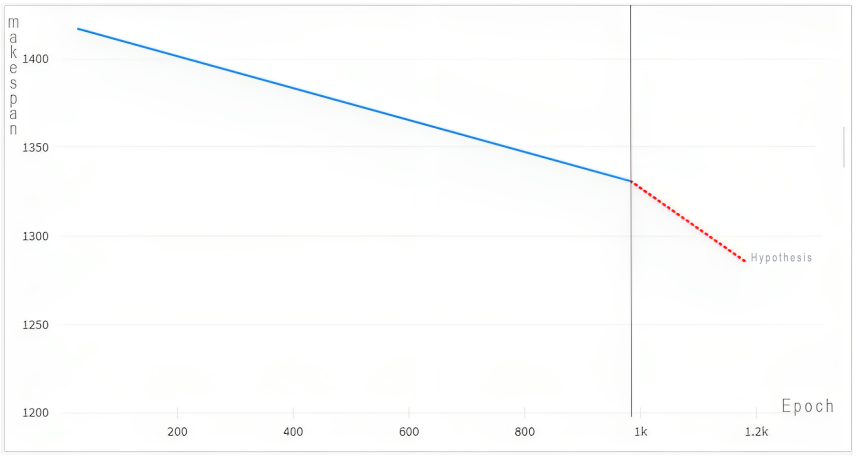

Figure 4.1: Expected makespan decrease after additionally training a DT. **Blue**: orgin-agent training **Red**: Additional DT training

4.2 Common Decision Transformer Core and Training Process

Both, the Dispatching Decision Transformer (D-DT) and the NeuroLS Decision Transformer (NLS-DT) build upon a common core of the DT implementation, which is explained in this section.

The original Decision Transformer paper proposes two implementations, one for problems with continuous action space and a second one for problems with discrete action space. Since both the D-DT and the NLS-DT have a discrete action space (see: 4.3 and 4.4) their implementations are based on the discrete action space implementation of the DT, which was originally applied to atari games. The discrete action space DT implementation is based on the *minGPT* model by Kaparthy [48]. minGPT is a minimal pytorch implementation of the GPT model which provides a boilerplate training loop and a GPT implementation that both were originally designed for generating sentences from character strings, but can be adapted for other types of problems, making it possible to adjust it for the proposed DT models. The GPT implementation employs an implementation of the transformer block similar to the explanations in 2.3.1 containing a multi-head self-attention implementation. For training, an ADAM optimizer in combination with cosine learning-rate-decay is used. Adam is an optimization algorithm which calculates how the weights of neural network should be updatet. It is able to adjust the learning rates individually for each parameter. This adaptability often results in faster convergence and better performance compared to traditional optimization algorithms like stochastic gradient descent [49]. Cosine learning rate decay adapts the learning rate by calculating the decay factor with a cosine function. This smoothly reduces the learning rate from its initial value to near-zero during the training and adjusts it back to the initial value periodically.

The authors of DT adjusted the minGPT model by replacing its token embedding by three separate embeddings for each of the three modalities (*states, actions and rtgs*). In the DT states are encoded with multiple convolutional layers because the input of Atari games are images. As activation function the hyperbolic tangent function is used which projects the input to the interval [-1,1]. In contrast to the original minGPT model, the positional embedding is not in relation to the block size but considers the maximum trajectory length of the problem.

Thanks to the masking of the transformer decoder architecture (see: section 2.3.1), during training, the loss is calculated based on the whole sequence of length $\mathcal{K}$. This means for each token t_i a prediction is made based on its previous tokens. For t_0 no previous token is available. For t_K the prediction is based on all the previous $\mathcal{K}$ tokens. Therefore, $\mathcal{K}$ *predictions* are made, for each sequence iterated in the training. This means the loss is calculated on $\mathcal{K}$ action predictions per sequence. As loss function, the *cross entropy loss*, which is given in equation 4.1 is used.

$$\text{Categorical Cross-Entropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} \cdot \log(p_{ij}) \tag{4.1}$$

Furthermore, the DT is trained with the hyperparameters shown in table 4.2.

Hyperparameter	Value
Number of layers	6
Number of attention heads	8
Embedding dimension	128
Batch size	512
Context length $\mathcal{K}$	50
Nonlinearity	GeLU
Dropout	0.1
Adam betas	(0.9, 0.95)
Grad norm clip	1.0
Weight decay	0.1
Learning rate decay	cosine decay

Table 4.1: Further hyperparameters adopted from original DT

4.3 Dispatching Decision Transformer

4.3.1 Environment

For the Dispatching Decision Transformer the *schlably tetris_scheduling* environment is used. schlably is a framework developed by [50] that simplifies the set up of a reinforcement learning architecture for the JSSP. Among other things, it offers functions for creating JSSP problem instances and environments for interacting with them. The *tetris_scheduling* environment within schlably facilitates interactions with JSSP instances of varying complexities, accommodating any number of machines, jobs, and even the selection of tools. For learning D-DT a 6x6 environment without tools is chosen. The environment implements the basic *gym*-function *step*, *reset*, *_state_obs* and *render*. States actions and returns in the environment are defined in the following way:

States

Schlably offers different observation strategies that represent the state for the agent. The following observation strategies are used in various experiments:

- ***sensible_obs:*** The *sensible_obs* observation contains the following information per task of a JSSP
  - The time required for processing the task.
  - The remaining processing time for the entire job.
  - The remaining processing steps within the job.
  - The remaining processing time necessary for the next required machine.
  - The remaining processing steps for the next required machine.
  - The earliest starting point for the task.
  - The earliest finishing time for the task.
- ***mtr_comparision:*** The "MTR comparison observation" is a method that involves comparing different jobs based on their progress in processing. In this comparison:

  - A job is assigned a value of 1 if it has progressed further than another job.
  - It is assigned a value of -1 if it has progressed less than the other job.
  - If two jobs are at the same processing stage, they are assigned a value of 0.

  This approach allows for a straightforward evaluation of how far jobs have advanced in their processing, providing a clear indication of their relative progress.

- ***full_raw:*** The *full_raw* observation consists of:
  - remaining processing times of all tasks on each machine
  - remaining processing times of tasks on each job
  - processing time of the next task per job
  - which machine is used the for next task per job

Actions

The action space of the environment is given by the number of jobs, so in the case of the D-DT the action space is defined as $A = \{a|a \in [0, 5]\}$ where each possible action refers to one job. The *step* method receives a selected action and based on that, schedules the next task from the job given by the received action.

Reward

As reward strategy the *dense-reward-strategy* proposed by Zhang et. al [35] is used. Using this strategy, the reward for transitioning from s_t to s_{t+1} is calculated as the difference of the makespans of the states:

$$r_t = R(a_t, s_t) = C_{max}(s_t) - C_{max}(s_t + 1) \tag{4.2}$$

Since at each time step t a new task is dispatched it holds that $C_{max}(s_t + 1) \geq C_{max}(s_t)$ at each time step, because the schedule can never become shorter than before and can only be of the same length, when a task is scheduled on a machine that is not the machine with the current highest makespan. Therefore, r_t is either zero or negative and can therefore be considered as a punishment the longer the schedule becomes. The cumulative reward $\sum_t^{|T|} r_t = C_{max}(s_0) - C_{max}(s_{|T|})$ during inference, for an instance to be solved, the return-to-go is calculated as the lower bound m_{lb} of the JSSP instance with the lower bound being equivalent to equation 4.4. Furthermore the environment employs a job mask, which ensures that only jobs which have open tasks can be scheduled.

4.3.2 Model Description

The D-DT autoregressively creates a schedule for a JSSP instance by dispatching one task at time in $|T|$ steps. In each step, the D-DT predicts the job whose next task is to be scheduled next, setting its start time to the end time of the task that is previously scheduled on the machine where the task needs to be processed on. At each timestep the last $\mathcal{K}$ scheduled jobs, as well as the rtgs and last states will be considered as the input. The schedule is completed as soon as all tasks have been dispatched.

4.3.3 Training

Dataset creation

To create the training datasets for the D-DT, an already trained proximal policy optimization reinforcement learning agent is used. The agent is able to solve JSSPs of size 6x6, by scheduling one task at a time, achieving a mean makespan $\overline{C_{max}} = 65.2$. In the data creation process a predefined, for some experiments varying number of JSSP instances, is solved by the agent and for each scheduled task a new datapoint is created. Each datapoint contains the following information which is then used to learn the D-DT:

- Step number of the current instance
- The observation s_t taken in that step, defers depending on which observation strategy is used
- Action a_t performed in that state
- Reward received after performing a_t in s_t
- *action_mask* restricting which actions are possible in the current step
- *returns-to-go*

The returns-to-go are calculated after a full JSSP instance is done, by iterating over the solved instance from the last to the first step. Listing 4.1 shows the code for calculating the returns to go on the data points of a solved instance.

Listing 4.1: Calculation of rtg for data points

```
def calculate_returns_to_go(solved_instance):
    return_to_go = 0
    for datapoint in reversed(solved_instance):
            return_to_go = return_to_go + datapoint.get("reward")
            datapoint['returns_to_go'] = return_to_go
```

Training process

The training was performed on 2 NVIDIA Tesla V100 32GB GPUs. To perform parallel training on multiple GPUs the training loop of minGPT was adjusted by using pytorchs *DistributedDataParallel* [51].

To properly load the data in the training process, a new pytorch *Dataset*-subclass called *StateActionReturnDataset.py* is created. This is also done in the original DT-work but needs to be completely replaced due to a fundamentally different structure of the Atari-dataset compared to the NLS-DT datasets. The *StateActionReturn-Dataset.py* contains the whole dataset for the current training and has the context length and the problem size (number of iterations for NLS-DT and number of tasks to schedule for D-DT) as class variables. Furthermore, it overrides the *__getitem__*-method in order to return a sequence of $\mathcal{K}$ data points. Given an index i of a data point, the method returns the sequence of datapoints $dp_i, dp_i + 1 ... dp_i + k - 1$. Each

data point dp_i contains the *observation*, the *action*, the *rtg*, the *action_mask* and the *timestep* of the data point. If the passed index i is less than $\mathcal{K}$ steps smaller than the maximum problem size, the last data point of the instance for which a complete sequence of length $\mathcal{K}$ is possible is returned. Listing 4.2 shows a simplified version of the method.

During training, the data points are shuffled, meaning random permutations of indexes are drawn from the dataset and used for training on the different GPUs. As mentioned in section 4.2, during training, a cosine learning rate decay is used.

Listing 4.2: Method to retrieve sequence of $\mathcal{K}$ data points

```
def __getitem__(self, index):
    instance_step_number = data[index].get("step_number")
    if (instance_step_number + context_length) > problem_size):
        index -=
            (instance_step_number + context_length  - problem_size)

    sequence = [self.data[index:index+context_length]]
    return sequence
```

4.3.4 Implementation

The implementation mostly follows the common core description in 4.2. One adjustment for the D-DT is made in the minGPT model. In order to properly predict the next action it is necessary to implement the action masking in the model. Since the minGPT model predicts actions based on a softmax of logits coming from the minGPT decoder head, the masking of unavailable jobs is achieved by setting the logits of the masked actions to an extremely low value ($-1e8$). This way the result of the applied softmax will assign the masked actions the lowest value and thus prevent them from beeing prediccted as the next action to choose. In order to implement this, the mask needs to be passed as a parameter to the model at each prediction step.

4.3.5 Evaluation Metrics

As for the NLS-DT, an online evaluation is performed for the trained D-DT models. Each trained model is applied to the *tetris_scheduling* environment for the respective problem size. Since the D-DT is only applied to problem sizes of 6x6 and the Taillard benchmark only provides instances for problem sizes starting at 15x15, a test dataset created with the *schlably* instance generator is used to evaluate the results. Each model result is compared relative to the original model to see if the achieved mean makespan is smaller.

4.4 NeuroLS Decision Transformer

4.4.1 Environment

The environment completely retains the original Neuro-LS environment. A state is equivalent to the state description in section 2.5.1. The action space depends on what kind of different action set described in section 2.5.1 is used. Since the A_A action policy, i.e. only deciding whether to accept the next solution or not, for smaller problems mostly decides to accept the next solution (so basically a regular local search is performed) and for larger problem sizes it is always worse than the A_{AN} and A_{ANP} approach, NLS-DT was only trained on the A_{AN} and A_{ANP} agents. Thus, the number of possible actions depends on the number of different neighborhood operators and whether a perturbation should be performed. Therefore the action space is either of size 8 for A_{AN}, because 4 different neighborhood operators are possible, or of size 10 for A_{ANP}, because perturbation counts as an additional operator and each operator can be used in combination with acceptance or non-acceptance.

4.4.2 Model Description

The architecture of the NeuroLS Decision Transformer consists of the encoder component of the original NeuroLS model as described in section 2.5 and a separate DT implementation, which replaces the decoder part of the original NeuroLS model. Thus the input or observation of the DT-Component is the output of the NeuroLS aggregator. Figure 4.2 visualizes this idea.

Given a JSSP, NLS-DT will encode its graph representation by using the original NeuroLS encoder and aggregator. The resulting aggregated observation, which is a linear projection of the solution s at time step t, concatenated with 1) its cost $f(s)$, 2) the cost $f(\hat{s}_t)$ of the best solution found so far, 3) the last acceptance decision, 4) the last operator used, 5) current time step t, 6) number of LS steps without improvement and 7) the number of perturbations or restarts, will then be passed to the DT.

The DT will generate an action based on the last $\mathcal{K}$ aggregated observations, returns-to-go and actions. The generated action will be executed in the NeuroLS environment and defines if the current schedule will be accepted, how the neighborhood of the current accepted schedule will be built, or if a perturbation will take place.

During inference the returns-to-go cannot be simply calculated by the lower bound makespan, as it is the case for the D-DT. Since the NeuroLS environment calculates the reward as the relative improvement from one step to another, starting from an initially created schedule (see: section 2.5.1) the returns-to-go need to be calculated relative to the initial schedule that the local search is starting from. Therefore, during inference the return-to-go for an instance is computed as:

$$\hat{R}_0 = m_{init} - m_{lb} \tag{4.3}$$

With m_{init} being the initial makespan of the schedule that the local-search starts from (this is created by using the FDD/MWKR PDR) and m_{lb} is the lower-bound of makespan for the current instance. The lower-bound is given by equation 4.4

$$l_{mb} = \max_{m \in M} \sum_{t \in T_m} d(t) \tag{4.4}$$

With T_m representing the set of tasks assigned to machine m and $d(t)$ being the duration task t. For m_o being the optimal makespan of a JSSP it may be noted, that $m_{lb} \leq m_o$ holds in all cases, since $m_{lb} = m_o$ is only possible if the optimal schedule has no gaps on the machine with the longest duration [52].

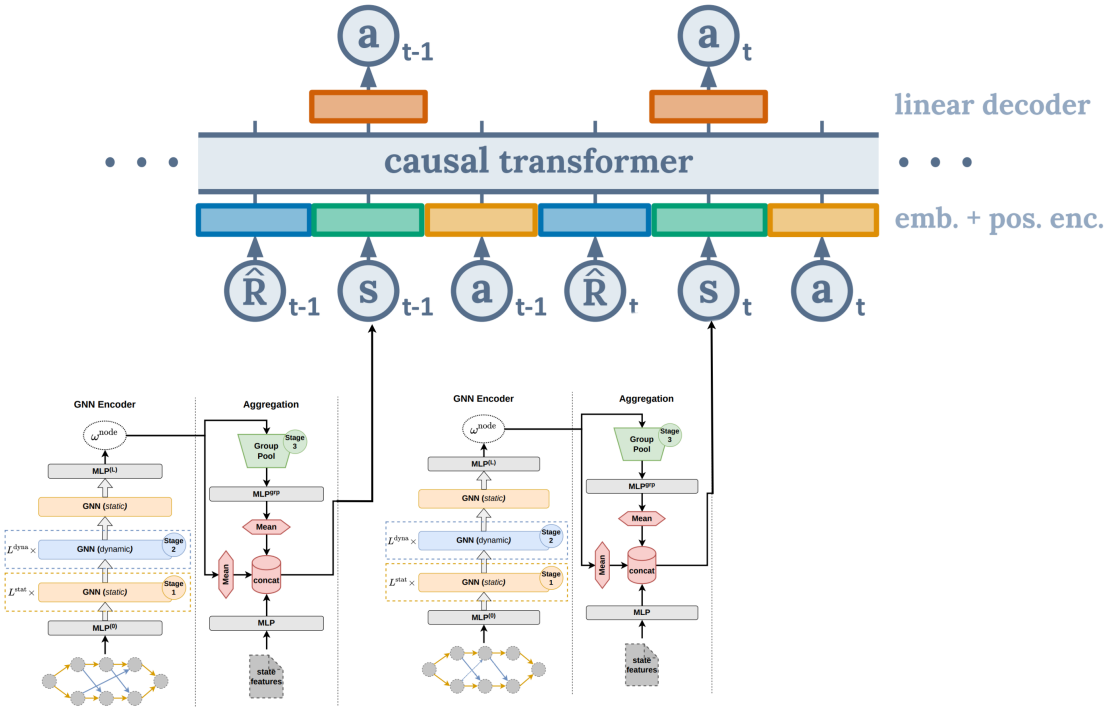

Figure 4.2: NeuroLS Decision Transformer Architecture

4.4.3 Training

Dataset creation

To train the NLS-DT, different datasets for each problem size were created. Each training dataset contained the local search operations conducted by the NeuroLS (NLS) model for 1,000 instances. Furthermore, for each problem size, one dataset was created from trajectories that included 100 LS iterations per instance, while a second dataset was derived from 200 LS iterations for each instance. Each instance has the number of LS iterations as the amount of data points, resulting in datasets of 200,000 and 100,000 data points. To construct a dataset, the NLS model is applied to a specified number of instances with the specified number of iterations. For each JSSP instance to which the model is applied, the following data is extracted during the dataset creation process to generate a training data point:

- Step number (current iteration)
- Current aggregated observation o_t
- Action a_t performed in s_t
- Reward received after performing a_t in s_t

- Makespan of the current schedule
- return-to-go

The rtgs are calculated after completing all search iterations, by iterating over each instance from the last to the first step, following the code in listing 4.1 Each data point within the generated dataset represents a single iteration step of the local search.

Training process

The training process of the NLS-DT is mostly equivalent to that of the D-DT. The *StateActionReturnDataset.py* is only adapted for the the problem size, and the *action_mask* is removed from the dataset.

Test runs during training

To assess the performance of the model during training, it is regularly tested on a validation dataset in addition to monitoring the training loss. Based on how well the current model under training performs on this validation dataset, the current version is saved. If the model achieves a smaller mean makespan on the validation dataset compared to previous test runs, the version is saved. Otherwise, the training continues without saving the current version. Since the NeuroLS environment is computationally intensive to run and solving one instance takes 9 to 15 seconds to be solved, performing a test run after every epoch is not feasible without having to accept extremely high computation times. Here a tradeoff between an appropriate number of validation instances, that should be solved in the test run and the amount of test runs performed during the training process is made. On the one hand, choosing the amount of validation instances too low leads to a mean makespan that is too far away of the true mean value. On the other hand, choosing too many validation instances leads to high testing times, during which the training is not continued and therefore makes the whole training process much slower. During experimentation, a value of 30 validation instances and running a test run every 33 steps was found to be a good tradeoff between test runs and a expressive mean makespan.

4.4.4 Implementation

In order to replace the decoder of NeuroLS with a DT implementation, a new class *DecisionTransformer* was implemented in the *tianshou_utils.py* of the NeuroLS code. As shown in Figure 4.3 the created class provides four methods. *calc_neuroLs_rtg()* manages the calculation of the initial rtgs based the lower-bound makespan of the current instance and the makespan of the initially created schedule. *update_rewards()* calculates the rtg of the current step and appends it to the list of previously received rtgs. *update_states* method manages to append the current aggregated observation to the list of previous states. The *get_sampled_action()* processes the sequence lists by using the utils.sample() method from the minGPT implementation to crop them to the context length, pass them through the minGPT model and receive the predicted action. Finally, the *reset()* methods resets all sequences and calculates the initial rtg for the upcoming JSSP instance. Since the lower bound and the aggregated

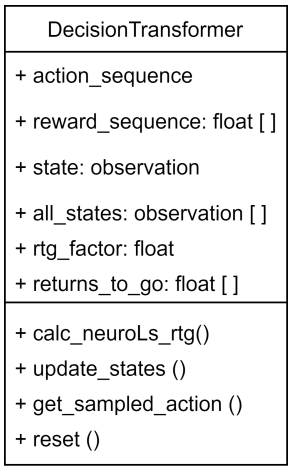

Figure 4.3: UML for DecisionTransformer

state are not accessible in the original NeuroLS implementation, small adjustments have been made to it to access these. The *lib/networks/model.py* of NeuroLS gets an additional public class variable where in each iteration the aggregated state is saved. In the *lib/env/jssp_env.py* an additional function *calc_lower_bound* is added which calculates the lower bound makespan of the current instance and adds it to the observation.

4.4.5 Evaluation Metrics

To evaluate the performance of the different models, online evaluations are used. This means that the trained model is applied to the simulated environment [27]. For the JSSP, the evaluation is usually done by testing the model, in the respective environment, on different benchmark instances [53]. As evaluation metric the achieved mean makespan on each problem size is used.

The trained models are applied to the Taillard benchmark, and to newly created test dataset for each problem size, which consist of 100 instances each.

4.4.6 Hyperparameter Tuning

First, initial manually performed experiments for 15x15 problems showed that for $\mathcal{K} = 50$ and 200 iterations the trained model performed best. Furhtermore a hyperparameter tuning with fixed $\mathcal{K}$ and a fixed number of iterations is performed. For the hyperparameter tuning, a manual grid search is performed. In this process, a fixed subset $h_i \subset H_i$ of the parameter space is formed. Then for each hyperparameter combination in $h_i \times h_j \times ... \times h_k$ a model is trained in order to evaluate which performs best.

For the NLS-DT the Hyperparameters, batch size and learning rate are tuned. To keep computation times low, three different values are selected for each hyperparameter. This leads to nine models that are trained to find the best parameter combination. Table 4.2 shows the results of the different hyperparameter combinations. Each was evaluated on 100 JSSP instances. Other hyperparameters are chosen equivalently as in the original DT paper, shown in table 4.2. Figure 4.4 and figure 4.5 show the mean makespan on the validation dataset and the development of the train loss, respectively.

Additional manualy performed experiments revealed, that the model performs better when its trained on a dataset which consits of NLS trajectories of that performed 100 LS iterations per instace instances but the NLS-DT is applied for 100 instances.

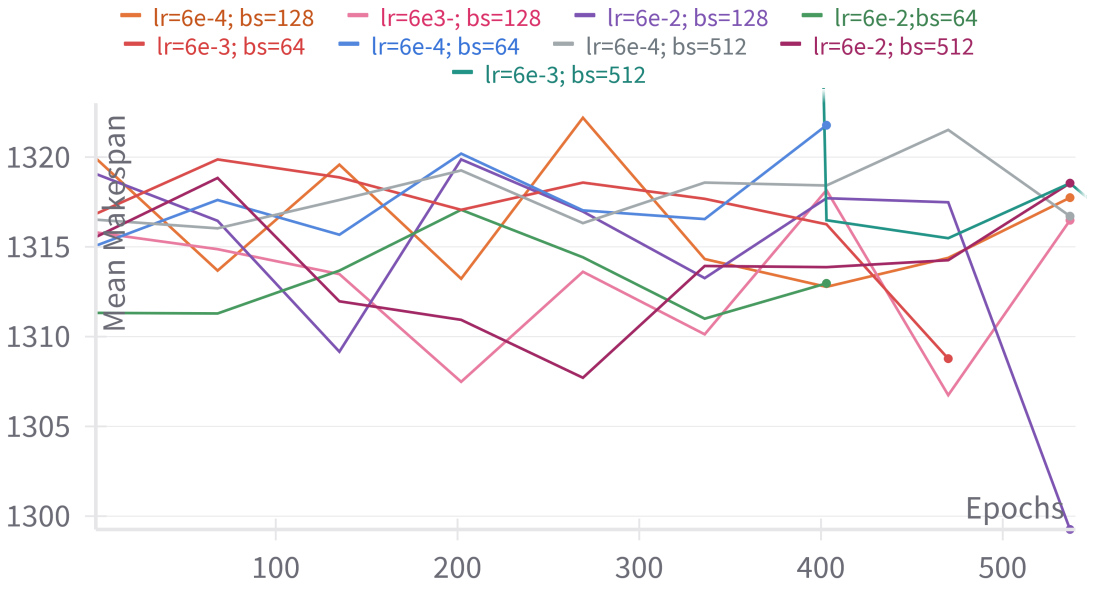

Figure 4.4: Mean makespan curves during hyper parameter tuning.

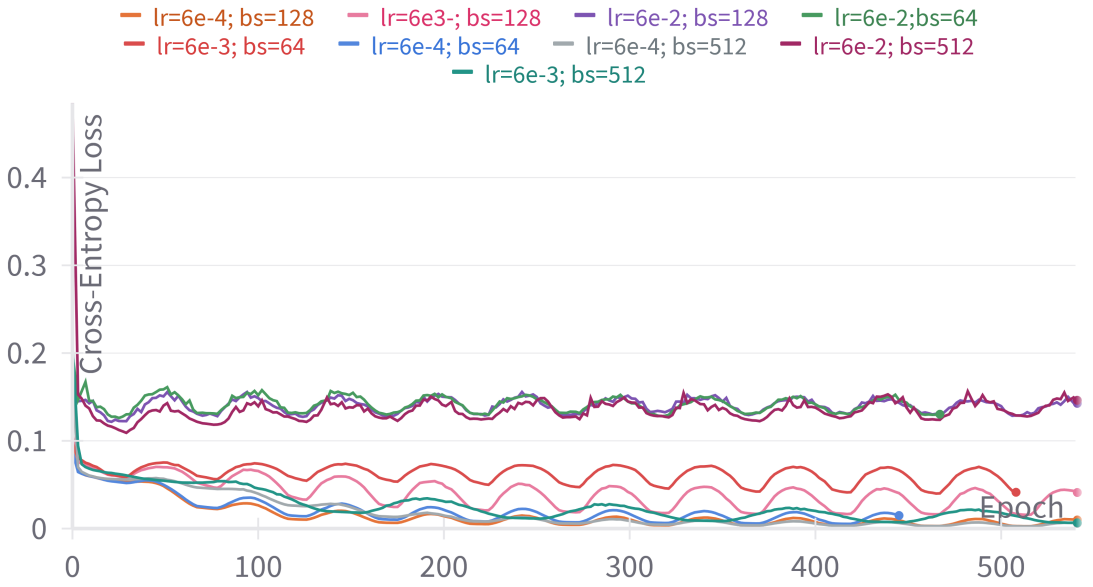

Figure 4.5: Loss curves of the hyper parameter tuning

Learning Rate	Batch Size	$\overline{C}_{max}$
6e-4	64	1325
6e-4	128	1315
6e-4	512	1315
6e-3	64	1311
6e-3	128	1323
6e-3	512	1327
6e-2	64	1311
6e-2	**128**	**1310**
6e-2	512	1319

Table 4.2: Hyperparameter tuning results for NLS-DT

5. Experiments and Results

The following chapter presents the conducted experiments and their results in order to answer the three research questions. First the Experiments of the D-DT and its results are presented, followed by those of the NLS-DT.

5.1 Dispatching Decision Transformer Experiments

With the D-DT various experiments are conducted with the main goal to evaluate if the D-DT is able to outperform the PPO based origin-agent. The broader objective is to investigate the effects of different context lengths, the action selection of the origin-agent and the observation strategies on the performance of the model. In this context, targeted experiments are conducted to gain a deeper understanding of the influence of these variables. This is followed by analyzing the distance of the taken actions and the created schedules to those of the origin-agent. Furthermore, the influence of the rtg is analyzed by applying the model to the test-dataset with various rtgs.

5.1.1 Evaluation on mean Makespan

To answer the first research question and analyze whether the D-DT is able to outperform the orgin-agent in terms of the achieved makespan, various models are trained. The trained models vary in the context length how the origin agent chooses actions (deterministic or stochastic) and in the observation strategy used. The achieved makespan of the origin agent on the created test dataset that is used for comparison is 65.2

Context length

The goal of the variations in the context length $\mathcal{K}$ is to illuminate if and how strong varying values of $\mathcal{K}$ influence the performance of the D-DT. For $\mathcal{K}$ the three values, 2, 6 and 30 are used to train individual models. The initial experiments were conducted with $\mathcal{K} = 6$, this aimed to align the context length with the number of available tasks, allowing to look back over the immediate task history. The extreme values of 2 and 30 were employed to explore if very low and very high context lengths significantly influence the behavior of the model. Figure 5.1 shows the learning progress of the achieved makespan on the validation dataset of the models trained with the three different context lengths. It shows that the overall performance during training with the three context lengths does not differ. Table 5.1 shows the achieved makespans on the test dataset. As shown in figure 5.1, neither the learning progress nor the performance of the trained model is affected by changes in context length. Although

the performance of the model with $\mathcal{K} = 2$ ()is slightly better than that of the origin-agent, the improvement is negligibly small. This indicates that the context length does not have a significant effect on the performance of the model.

$\mathcal{K}$	$\overline{C}_{max}$
2	65.42
6	65.68
30	66.7

Table 5.1: Results on the test dataset of D-DT models with difrent context length

The results imply that there is no meaningful information in the sequence that the model can capture. Reasons for this possibly are, that much of the information which is contained in the sequences, except for the returns-to-go, is already contained in the observation of a single state. The *sensible_obs|mtr_comparision*, for instance, implicitly provides information on the actions performed in earlier steps, since this is evident from the start and end times of the already scheduled tasks. Other information contained in the observation, like the remaining processing steps within a job, do also not provide additional useful information when considering their past values. Therefore, for the idea that similar past trajectories should yield similar future actions, the actual explicit indication of the last $\mathcal{K}$ observartions and actions is not necessary.

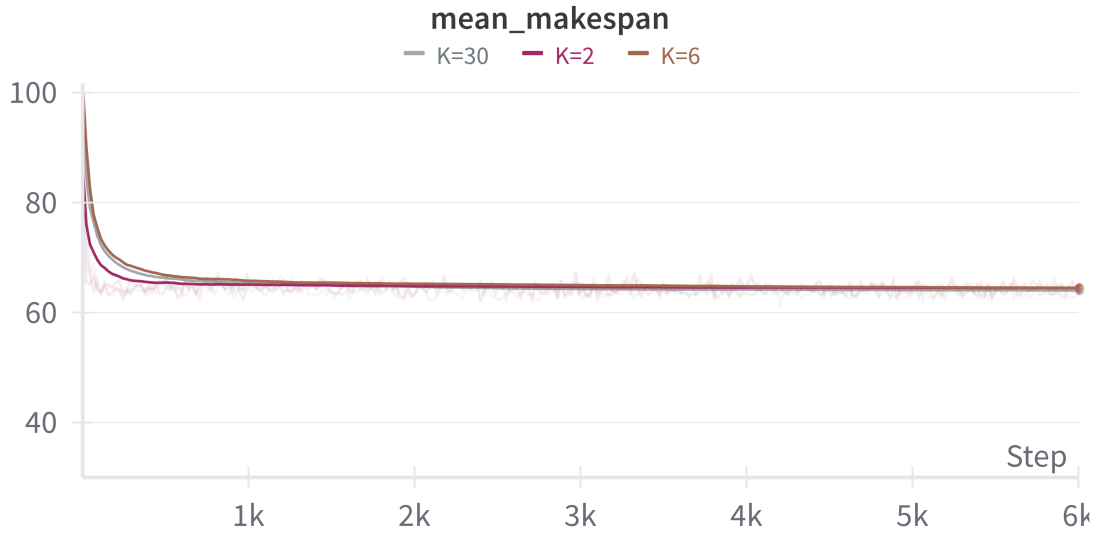

Figure 5.1: Learning Progress of models with diferent context lengths

Origin-Agent action selection

Another factor that affects the D-DT, is the method by which the origin-agent picks its actions. The PPO origin agent can choose its actions either stochastically or deterministically. The deterministic approach always selects the action with the highest probability returned by the softmax value. Conversely, the stochastic approach considers the softmax values as a probability distribution across the action

space and samples one action accordingly. This approach is supposed to add randomness, enabling the agent to explore, while deterministic selection can enhance exploitation by consistently selecting the optimal action based on the current policy. Since exploration during offline reinforcement learning is not feasible in the learning process, but must be applied in the data collection phase, the idea is that the stochastic action selection will create a more explorative dataset. Therefore, one additional training dataset is created, where the origin-agent always picks it actions stochastically. Figure 5.2 illustrates the learning progress in terms of achieved makespan on the validation set. It compares the performance of an agent trained on data generated by an origin-agent using stochastic action selection, with that of an agent trained on data from an origin-agent employing deterministic action selection.

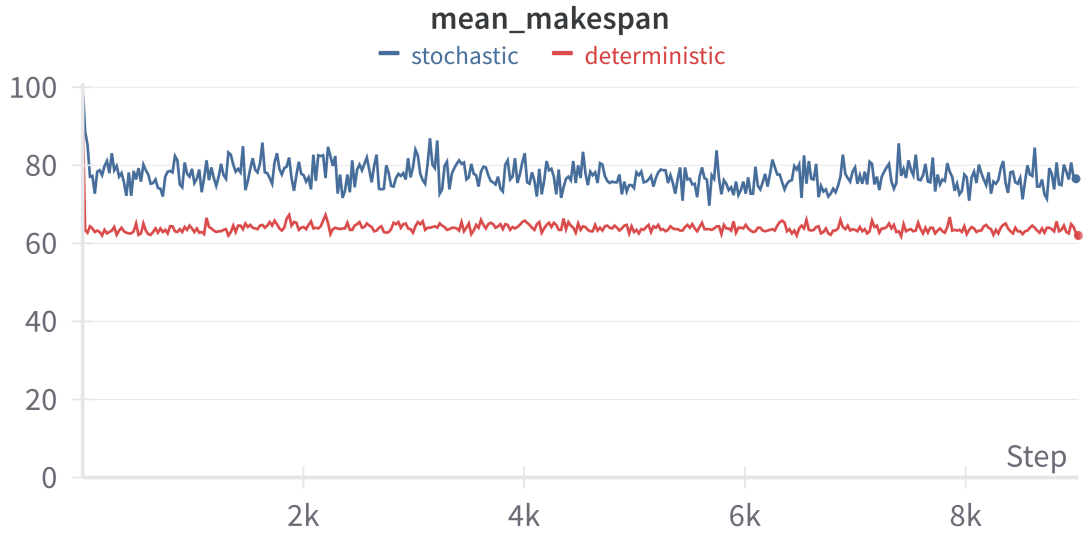

Figure 5.2: Stochastic vs Deterministic origin-agent

The stocastic origin-agent is performing significantly worse on creating the schedule with achieving in a mean makespan on the test dataset of 77.05. The origin-agent employing stocastic action selection exhibits a mean makespan of 85.375. Consequently, in this particular case the D-DT is able to outperform its origin agent, albeit falling short of the performance achieved by the deterministically learned D-DT and its deterministic origin-agent.

These results underscore that the D-DT is indeed able to find better schedules than its origin agents at least in some cases.

Observation strategy

Further experiments with the model are conducted by choosing different observation strategies. Additionally, to the observation strategie combination (sensible_obs|mtr_comparison) the observartion strategie full_raw alone and in combination with mtr_comparison are employed. It should be noted, that the origin-agent is still working with the (sensible_obs|mtr_comparison) observation, but during the data creation process, the other observations are extracted from the environment in order to train the D-DT on the other observations. In table 5.2 the results of

observation-strategy	$\overline{C}_{max}$
full_raw	76.96
full_raw|mtr_comparison	82.92
sensible_obs|mtr_comparison	67.02

Table 5.2: Results on tests dataset of different observation strategies with $\mathcal{K} = 6$

the different observation strategies are recorded. The results show, that the (sensible_obs|mtr_comparison) observation performs significantly better, and thus represents the state of the environmet better for predicting the actions.

5.1.2 Behavior Similarity

In order to get a deeper understanding of the behavior of the D-DT, it is of interest to see, if the D-DT creates different trajectories than the origin-agent, or mainly tries to clone the behavior of it. Furthermore the distance of the final created schedules is calculated. In order to analyze this, both the origin-agent and the D-DT are applied to 100 instances. For each of the instance, the taken actions and the schedules created by both models are compared.

The distance for the taken actions is calculated by the *Hamming distance* [54]. The Hamming distance quantifies the difference between two lits of equal length by counting the number of positions at which the corresponding list entries are different. Thereby it counts how many substitutions are needed to change one list into the other. For the D-DT the maximum possible Hamming distance between two schedules is 36, which would be achieved when at each step a different task is scheduled. Conversely, the lowest value is zero when both perform the same action at each step. It should be noted that two schedules can be identical even though the Hamming distance is > 0. This is the case when two tasks of different machines are dispatched in a different order. This will give a different action sequence but the same start-time for the tasks. In order to compensate this the second distance measure is aimed at comparing the schedules.

The second distance measure used for comparing the final schedules is the *start time distance* originally proposed as *cumulative absolute task deviation* in [52], which calculates the sum of absolute differences of the task starting times for each task, divided by the total number of tasks. The start time distance is calculated by the code shown in listing 5.1.

Listing 5.1: Calculation of start time distance

```
def start_time_distance(nlsdt, nls):
    sum = 0
    for i in range(len(nls)):
        sum += abs(nlsdt[i] - nls[i])
    return sum/len(nls)
```

Here $nlsdt[i]$ represents the start time of task i when the instance is solved by the respective NLS-DT-model and $nls[i]$ when scheduled by the original NLS model.

In Figure 5.3, the Hamming distances, and in Figure 5.4, the start time distances of the 100 runs are shown in box plots. The results are separated in three boxplots

according to whether the D-DT is better, worse or as good as the origin-agent. This is intended to illustrate the relationship between the distance measure and the performance of the model. Of the 100 runs 25 have a lower makespan than the origin-agent, 44 achieve the exact same makespan and 31 have a schedule with a longer makespan than the origin-agent. Both plots show the obvious tendency that the D-DT performs equally well to the origin-agent, when having a low Hamming and low start time distance, and thus in these instances the D-DT creates the same schedule as the origin-agent. The tendency that, about half of the schedules that are equally good, are distributed around a higher Hamming distance between 10 and 20, but anyway have a low start time distance is explainable by the fact, that the same schedule can be created even though the jobs are dispatched in a different order. Nevertheless, the high Hamming distance for equal schedules highlights that the D-DT learns its own strategy to solve the schedules On average, the better schedules have a Hamming distance of 20.25. For worse schedules, the average distance is slightly smaller with a value of 17.27. Similar holds for the start time distance. The mean start time distance of instances solved worse and instances solved better are close to each other, with a mean start time distance of 3.92 for instances performing better and 3.21 for instances performing worse.

In summary, the values of the Hamming distances in combination with the start time distances indicate, that the D-DT is able to learn an own strategy. On average, the D-DT chooses 16.74 actions out of the 36 possible actions different than the origin-agent. This creates schedules where the start times of tasks scheduled by the D-DT on average deviate by 2.32 from the start time assigned by the origin-agent.

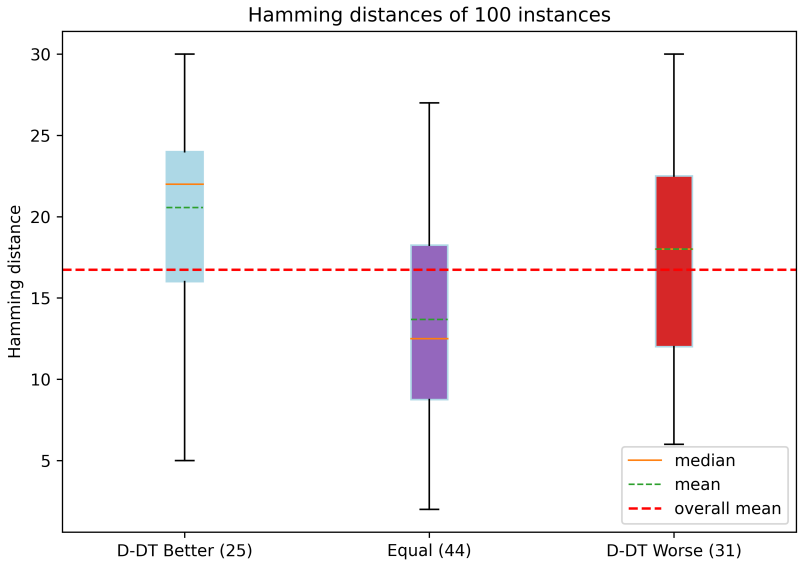

Figure 5.3: Hamming distance of 100 instances (Runs ordered by successfulness of the D-DT).

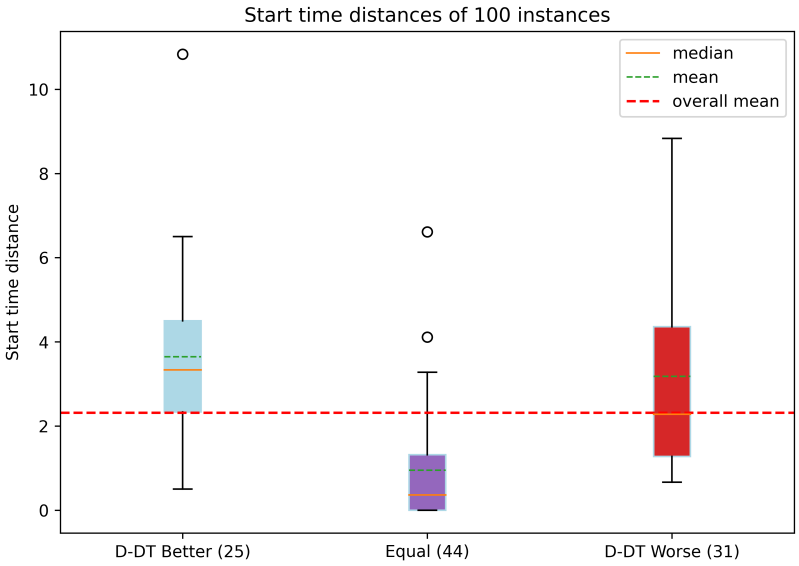

Figure 5.4: Start time distance of 100 instances (Runs ordered by successfulness of the D-DT.

5.1.3 Influence of return-to-go Conditioning

Since the rtg conditioning of the models is the key parameter allowing the model to possibly achieve better results than its origin model, it is of interest how different rtg affect the performance of the trained model. Therefore, in addition to experiments that only evaluate the mean makespan of the models, additional experiments are performed that change the rtg that the agent shall achieve, allowing to analyze its influence on the model behavior.

The authors of the orginal DT show that with a rtg value that is much higher than the possible achievable reward, the performance of the DT drops compared to using an optimal lower rtg value [8]. As mentioned before, for the previous explained experiments, the rtg value was calculated as the lower bound of the current JSSP instance. As explained in section 4.3, the lower bound represents the theoretically minimum makespan achievable for an instance. This can only be attained if a schedule exists where there are no gaps between tasks on the machine that has the longest makespan in the schedule. This means that the calculated lower bound might be lower than the actual possible makespan and thus the rtg might be higher than the achievable cumulative reward. This motivates to analyze if the lower bound is an appropriate value for the rtg.

To address the third research question regarding the responsiveness of the D-DT to different rtg values and to determine if the lower bound serves as an effective rtg benchmark, the model is tested across 100 instances. This is done for models trained with context lengths of 2, 6, and 30, applying varying rtg values to each. The rtg is varied by multiplying the calculated lower bound per instance with the factors between 0.3 and 1.7 in steps of size 0.05. It should be noted that the rtg value in case of the D-DT is always negative, since the rewards are designed as punishment for a makespan increase (see: 4.3). Therefore, a factor larger than 1, does lower the rtg, meaning the goal is set to a worse makespan and a factor lower than 1 will decrease the rtg and should condition to a better makespan.

Figure 5.5 illustrates the relationship between varying values and the mean makespan across the one hundred runs for the three different context lengths. The experiments show that with $\mathcal{K} > 2$, conditioning the D-DT to target lower makespans the scheduling performance improves. However, the observed absolute decrease of the achieved makespan is of small magnitude. For instance, in a context length of 30, the mean makespan improved marginally from a high of 68.5 to a low of 66.25, achieved with a factor of 1. In the case of context length 6, the variation in mean makespan was even more negligible, where the mean makespan varies aroung 66.5. For context length 2 no clear tendency to improved schedules by lower rtg values is visible as the outcomes fluctuate significantly. The overall best makespan is obtained with a factor of 1.5 for both context length 2 and 6. Here, achieving a mean makespan of 65.12, which is slightly lower than the mean makespan of the origin-agent.

In addition to evaluating the impact of rtg on makespan, its influence on the Hamming distance and the start time distance is analyzed. Figure 5.6 and Figure 5.7 show the mean Hamming distance and the mean start time distance respectively. As visualized in Figure 5.6 there is a noticeable trend where smaller rtg values correspond to reduced mean Hamming distances. This effect is strongest for the highest context length $\mathcal{K} = 30$. For the start time distance the effect for $\mathcal{K} = 30$ similar. $\mathcal{K} = 2$ exhibits the overall lowest start time distance across all rtg values, while $\mathcal{K} = 6$, although similar to $\mathcal{K} = 30$, shows slightly lower values. The same holds for the Hamming distance.

Overall, the conducted rtg experiments show, that rtg conditioning influences the strategy of the D-DT. In particular the experiment results demonstrate that the lower bound as the default value is not optimal and conditioning the D-DT to even lower rtgs slightly improves its performance. Beyond the influence of the rtgs, it is highlighted, that the D-DT performance is better in those scenarios where both distance measures are lower and therefore the strategy of D-DT is closer to that of the origin-agent.

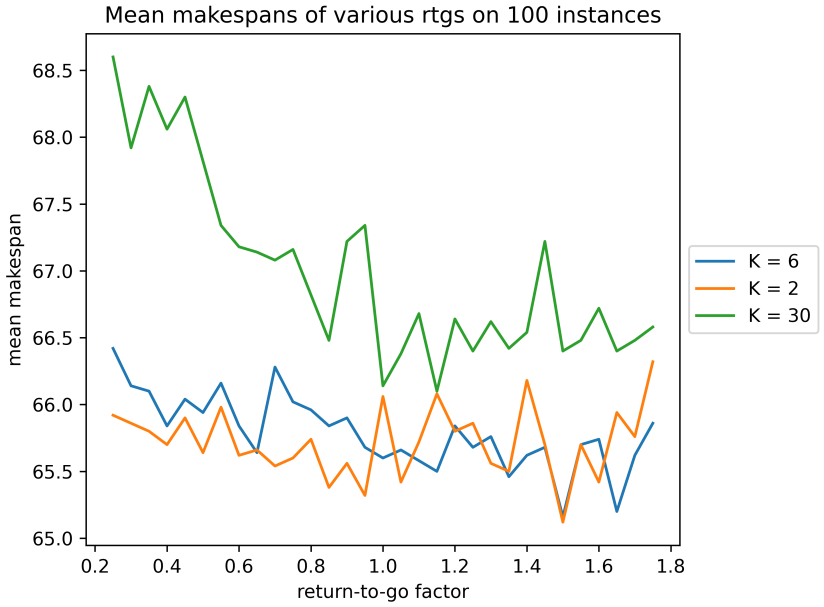

Figure 5.5: Mean makespan of 100 iterations per return-to-go

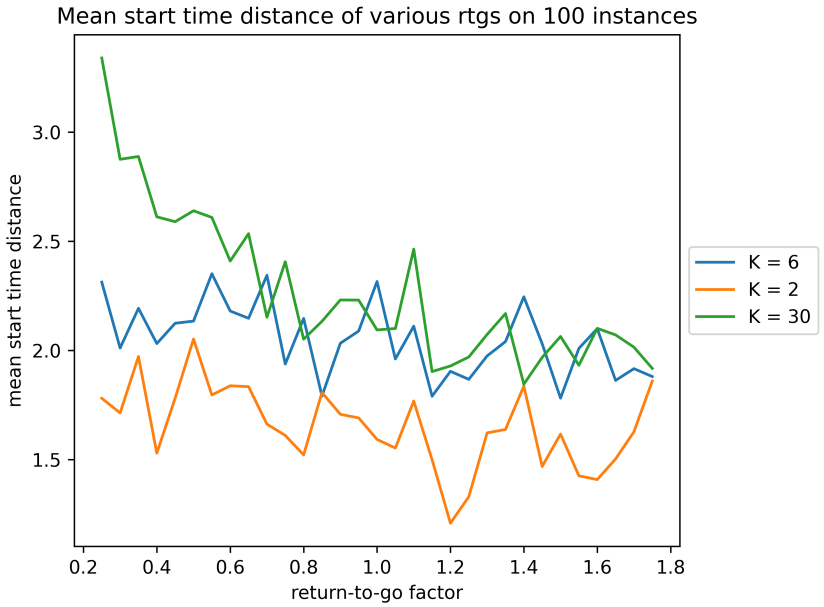

Figure 5.6: Mean start time distance of 100 iterations per return-to-go

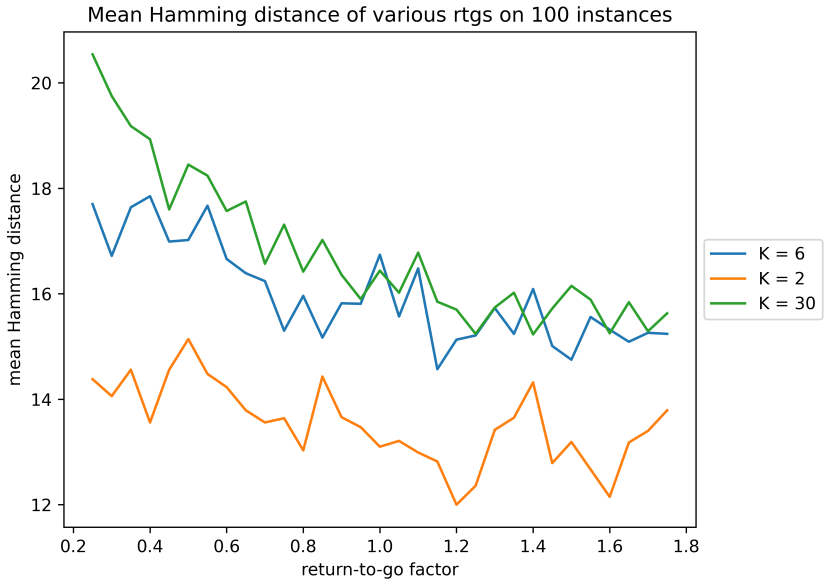

Figure 5.7: Mean Hamming distance of 100 iterations per return-to-go

5.2 NeuroLS Decision Transformer Experiments

The experiments with the NLS-DT follow a similar structure to those for the D-DT. First, the performance is evaluated by examining the achieved mean makespan on the test dataset and additionally on the Taillard benchmark. This is followed by an analysis of the behavior differences between the models. Furthermore, the action frequency is examined as an additional measure to determine if the model learns its own behavior. Lastly, it is analyzed how NLS-DT responds to variations in the rtg.

5.2.1 Evaluation on mean Makespan

In order to answer if the NLS-DT is superior to its origin agent in terms of the makespan, this section presents the results of experiments in which the NLS-DT is applied to JSSPs of various size. The achieved makespans are compared to those of the respective NLS model. NLS-DT models are trained for the problem sizes (15x15, 20x20, 30x20, 50x20, 100x20). Analyzing remaining problem sizes is beyond the scope of this thesis.

For each of the problem sizes two models are trained and evaluated on the mean makespan. One is trained on NLS trajectories that performed 100 LS iterations per instance and is further referred to as NLS-DT_{100}. The other is trained on NLS trajectories that performed 200 LS iterations per instance. However, both of the resulting models are applied for 200 LS iterations. Except for the 20x20 problem size, all models are trained on the NLS_{ANP}, because for the 15x15 problem NLS_A and NLS_{AN} show low variation in the action and therefore are not interesting for training the NLS-DT. The same issue holds for the NLS_A for the 20x20 problem. Therefore here the NLS_{AN} is chosen. For the remaining problem sizes the respective NLS_{ANP} is chosen because it exhibits the best performance.

The models are tested on a separate, newly created test data set and on the established Taillard benchmark. For the results on the Taillard benchmark, the respective optimality gaps are calculated as well. All results presented are achieved with models trained with the hyperparameters presented in section 4.4.6. Table 5.3 shows the results for the Taillard Benchmark and table 5.4 shows the results on the 100 created test instances. The achieved makespans of the NLS-DT models on each Taillard instance and the upper bounds for the calculation of the optimality gaps are given in Appendix A.

Model		Instance Size				
		15x15	20x20	30x20	50x20	100x20
		NeuroLS				
NLS_A	gap	7.74%	11.54%	16.35%	11.25%	5.90%
	$\overline{C}_{max}$	1324	1804	2267	3163	5682
NLS_{AN}	gap	8.74%	11.64%	16.53%	11.18%	5.84%
	$\overline{C}_{max}$	1336	1806	2270	3161	5679
NLS_{ANP}	gap	10.44%	13.38%	**16.06**%	10.88%	5.73%
	$\overline{C}_{max}$	1357	1841	**2261**	3152	5673
		NeuroLS Decision Transformer				
NLS-DT	gap	8.71%	11.75%	16.17%	**10.62**%	5.89%
	$\overline{C}_{max}$	1335	1807	2263	**3145**	5681
$NLS\text{-}DT_{100}$	gap	**7.65**%	**11.44**%	16.22%	10.69%	**5.59**%
	$\overline{C}_{max}$	**1323**	**1802**	2264	3152	**5664**

Table 5.3: Performance of NLS and NLS-DT for 200 iterations on the Taillard instances

The results shown in table 5.3 show that the NLS-DT models are able to outperform their respective origin agents on the Taillard benchmark in four out of five cases. Particularly the $NLS\text{-}DT_{100}$ shows competitive performance demonstrating the lowest optimality gap compared to the origin-agents in three of the four cases. For the 30x20 problem the NLS-DT is not able to outperform the origin-agent on the Taillard benchmark. The $NLS\text{-}DT_{100}$ for the 15x15 instances achieves the highest performance increase compared to its origin-agent NLS_{ANP} by lowering the optimality-gap by 2.79%. Although this is significantly better compared the origin-agent it is only 0.09% better compared to the NLS_A model. In the other three cases the improvement of the optimality-gap is of lower magnitude, ranging between 0.14% for the 100x20 problem and 0.2% for 20x20 problem. The only problem size for which the NLS-DT is better than the $NLS\text{-}DT_{100}$ is 50x20.

On the 100 instances of the created test datasets, the NLS-DT outperforms its origin-agents in three out of five cases. The models for problem size 20x20 and 50x20 do not outperform their origin-agent, but achieve the same mean makespan. For the 30x20 problem size the NLS-DT achieves a better mean makespan than the origin-agent, contrasting with its underperformance on the Taillard benchmark for the same size. For problem size 100x20 $NLS\text{-}DT_{100}$ is able to lower the achieved mean makespan by 7. The $NLS\text{-}DT_{100}$ for the 15x15 problem again shows the largest performance increase, lowering the achieved mean makespan by 19.

Model		Instance Size				
		15x15	20x20	30x20	50x20	100x20
		NeuroLS				
$NLS_{AN^*/ANP}$	$\overline{C}_{max}$	1320	1737*	2184	3137	5689
		NeuroLS Decision Transformer				
NLS-DT	$\overline{C}_{max}$	1310	1737	2181	3138	5689
$NLS\text{-}DT_{100}$	$\overline{C}_{max}$	1301	1737	2181	3137	5682

Table 5.4: Performance of NLS and NLS-DT for 200 iterations on 100 created test instances

5.2.2 Behavior Similarity

As for the D-DT, in order to answer the second research question about analyzing whether NLS-DT learns its own strategy, the distance between the action sequence and the start time of the final created schedules is calculated. Additionally, the Hamming distance of the machine sequence is calculated, since this is for the NLS-DT, as opposed to the D-DT, not equivalent to the action sequence. The machine sequence is defined as a list containing the tasks processed by each machine, arranged according to their execution order in the schedule. To compute the Hamming distance, the sequences from each machine are concatenated into a single list per instance, and the resulting lists from the NLS and the NLS-DT are compared. The machine sequence can be extracted from the disjunctive graph representation of the current schedule after the last local search iteration. A method to receive the machine sequence is already implemented in *jsspGraph* class of the NLS code. In order to retrieve the start times for calculating the start time distances, a new method is implemented in the *jsspGraph* class which determines the start times of each scheduled task from the disjunctive graph representation. This is achieved by applying a topological sort according to the precedence constraints of the tasks. This returns a sequence of nodes, in which no node comes before its predecessor. So for a given directed edge (u, v) the returned sequence contains u before v [55, p. 192]. Using the topological order of the nodes, the starting time for each job can be calculated, by summing up all processing times (represented by node weights) of the predecessors of each node. The python code for this is shown in listing 5.2

Listing 5.2: Calculation of start times from disjunctive graph

```
def get_task_starting_times(self):
    topological_order = list(networkx.topological_sort(self.graph))
    start_times = np.zeros(self.num_machines*self.num_jobs + 2)

    for node in topological_order:
        predecessors = list(self.graph.predecessors((node)))
        if len(predecessors) == 0:
            start_time = 0
        else:
            max_start_time = max([start_times[predecessor] +
            self.graph[predecessor][node]['weight']
            for predecessor in predecessors])
```

```
            start_time = max_start_time
        start_times[node] = start_time
    return start_times
```

The results of the three distance measures are shown in Figure 5.8 to 5.10. They are visualized in boxplots, separated for the instances in which the NLS-DT performs better, worse, or equally good as the original agent.

For all three measures, there is no notable difference when the model performs better, worse, or equal. The Hamming distance between the action sequence shows that of the 200 actions taken during the LS, on average 117.10 actions are chosen differently. In terms of the machine sequences, out of the 225 scheduled tasks in the 15x15 scenario, 108 tasks are scheduled differently. The overall mean start time distance is 77.40, which means that on average the starting time of the same task deviates by 77.40 between the schedules created by the two agents. Conclusively, these results clearly demonstrate that the NLS-DT has a different strategy in the action selection than the NLS that also leads to the creation of different schedules.

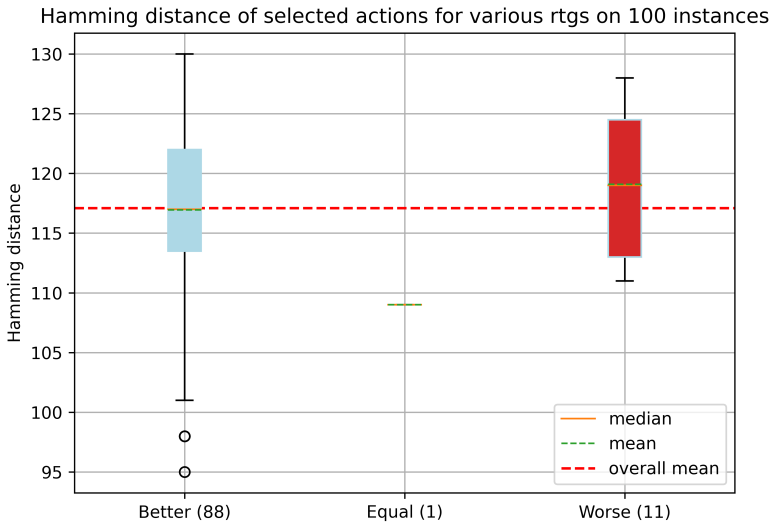

Figure 5.8: Hamming distance between action of 100 instances solved by NLS and NLS-DT (Numbers in brackets equals how often NLS-DT was better worse or equally good as NLS).

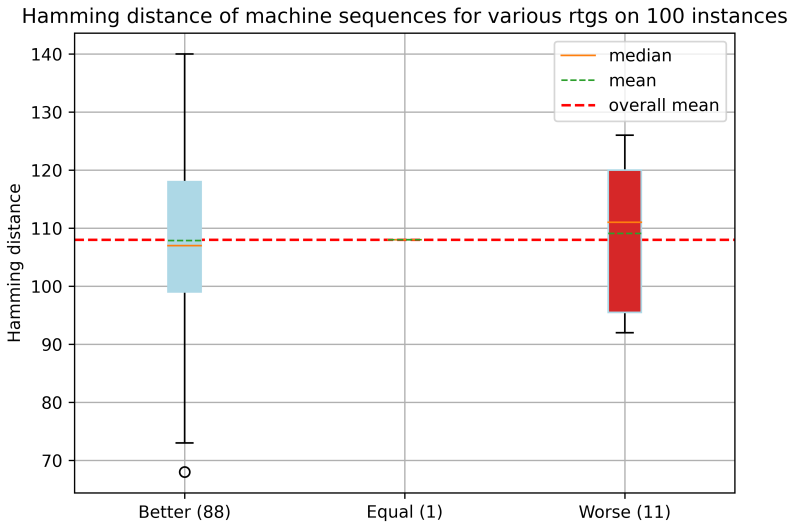

Figure 5.9: Hamming distance between machine sequences of 100 instances solved by NLS and NLS-DT (Numbers in brackets equals how often NLS-DT was better worse or equally good as NLS).

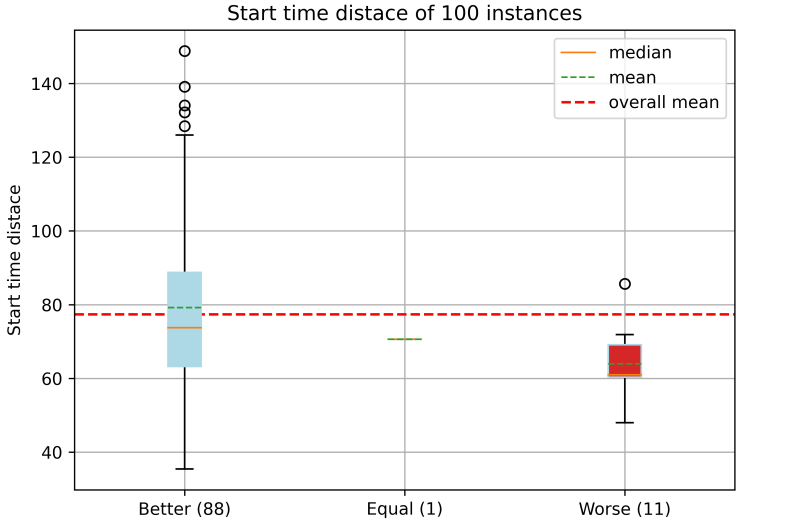

Figure 5.10: Start time distance between 100 instances solved by NLS and NLS-DT (Numbers in brackets equals how often NLS-DT was better worse or equally good as NLS).

5.2.3 Action Frequency

Another property that analyzes how the behavior of the NLS-DT differs from the behavior of the original NLS model is how often each action of the action space is performed, and thus how often each neighborhood operator is used. Figure 5.2.3 to Figure 5.2.3 show the frequency of each action for the NLS and NLS-DT or

NLS-DT$_{100}$ depending on which performed better for the different problem sizes. Here, each even action represents an acceptance of the solution in the previous LS step in combination with the neighborhood operator used for the next LS step.

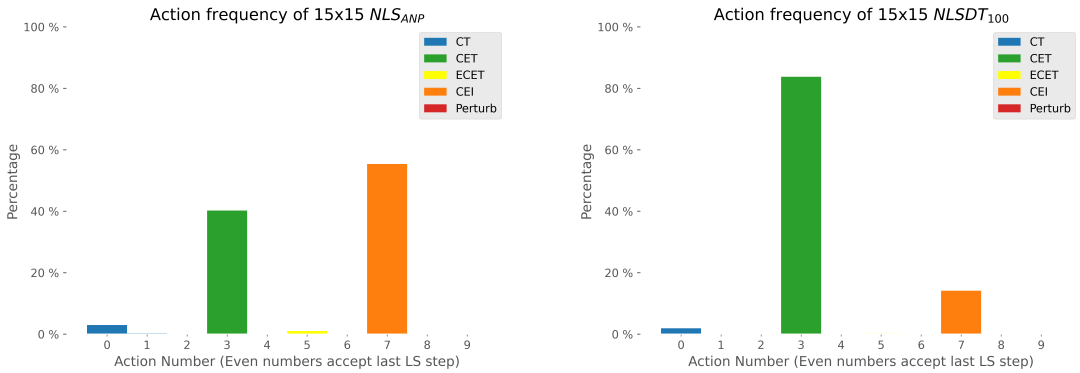

Figure 5.11: Action frequency for NLS$_{ANP}$ and NLS-DT$_{100}$ on 15x15

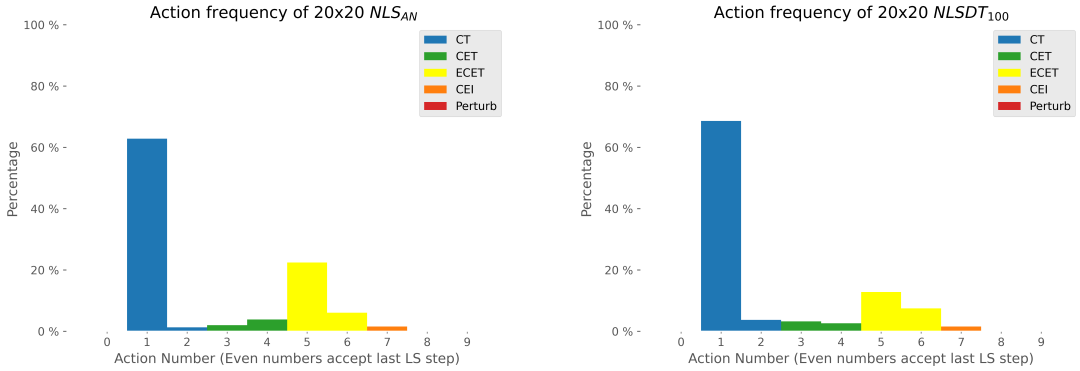

Figure 5.12: Action frequency for NLS$_{ANP}$ and NLS-DT$_{100}$ on 20x20

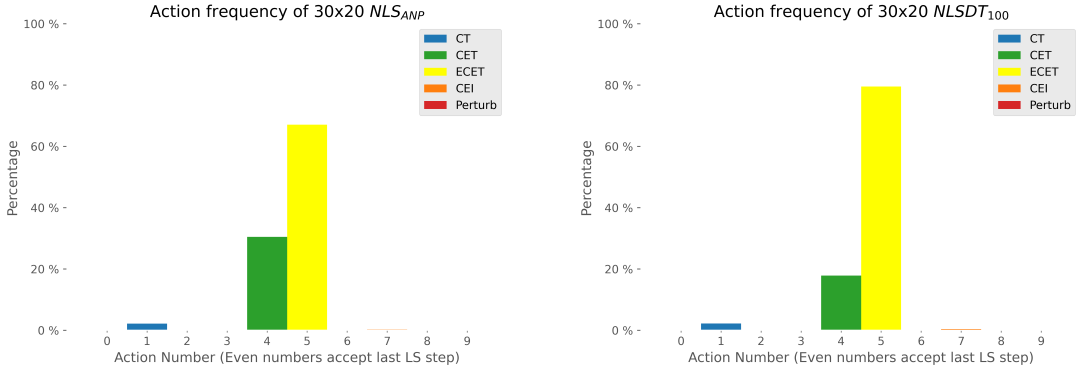

Figure 5.13: Action frequency for NLS$_{ANP}$ and NLS-DT$_{100}$ on 30x20

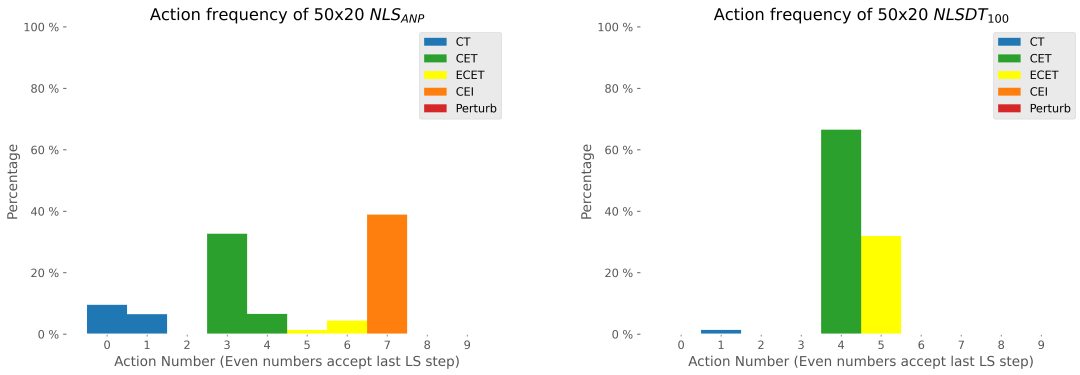

Figure 5.14: Action frequency for NLS_{ANP} and NLS-DT_{100} on 50x20

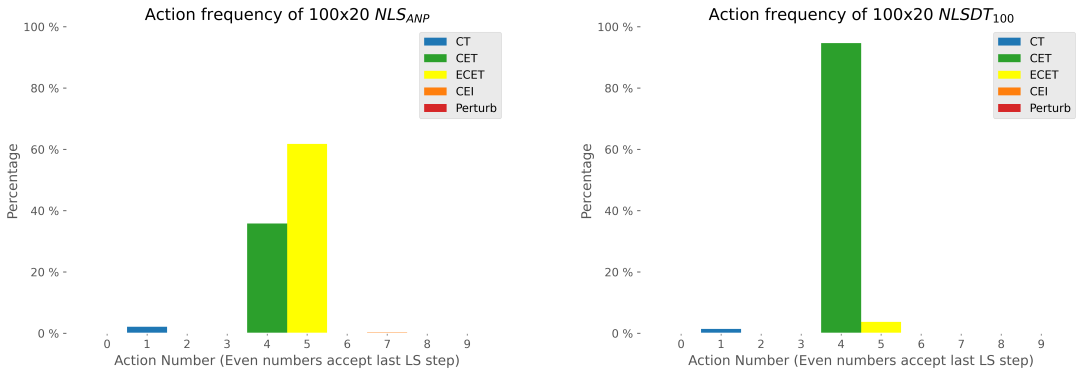

Figure 5.15: Action frequency for NLS_{ANP} and NLS-DT_{100} on 100x20

The figures show, that for each problem size the NLS-DT learns its own strategy as the frequency of the used neighborhood operators differs from that of the NLS model. The lowest difference between the action frequency is visable for the 30x20 problem, which aligns with the finding that the NLS-DT does not outperform the origin-agent. The highest difference is visable for the 15x15 and the 50x20 problem size. Here the NLS-DT learns a significantly different strategy than its origin-agent, notably these are the models that outperform their origin-agent by the highest margin. Overall, the comparison between the action frequency provides deeper insight into the difference of the strategies and aligns with the findings of the behavior similarity. Consequently, this analysis contributes to answer the second research question by underlining that the NLS-DT does learn its own strategy and does not clone the behavior of the origin agent.

5.2.4 Influence of the returns-to-go

The following experiments analyze how the rtg influences the achieved mean makespan, the start time distance, and the Hamming distance of the performed actions and the order of the jobs in the schedule. Thereby, they answer the second research question with respect to the NLS-DT.

The experiments are conducted in the following way: For each problem size, the best model with the best achieved makespan is selected to solve the test dataset for the respective environment for different initial rtg values. The rtgs are varied in the range of 0.25 times the initially calculated lower bound of the respective instance to 1.75 times the calculated lower bound. Each rtg value is applied to solve 100 instances of size 15x15 in 200 search iterations. Figure 5.16 displays the average makespan and the corresponding 95% Confidence Intervall (CI) for the 100 instances across various rtg values.

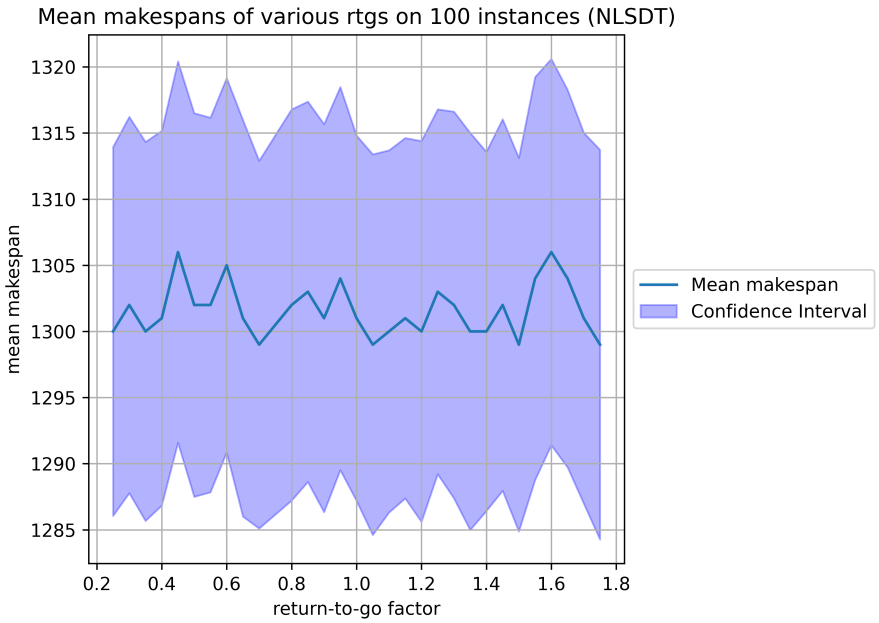

Figure 5.16: Mean makespans with 95% CI for various rtgs on 100 instances

The plot illustrates, that although some rtg values achieve a better makespan on the instance sample than others, they also exhibit large confidence intervals, indicating that the actual true mean value of the achieved mean makespan is somewhere inside that CI and thus the rtg might not actually differ.

In order to evaluate if there is a statistically significant difference between the various rtgs, additionally the CI of the difference between the mean makespans of the best (0.7) and the worst performing (1.6) rtg is calculated. If the calculated CI contains 0, the difference of the means is not statistically significant [56]. Equation 5.2 shows the calculation of the CI of the difference of the means based on [57]. The equation holds under the assumption of homogenous variances between both samples, which can be assumed for the sample variances 70.07^2 and 73.64^2.

Given the pooled variance $S2$

$$\begin{aligned} S^2 &= \frac{(n_1 - 1) \cdot S_1^2 + (n_2 - 1) \cdot S_2^2}{n_1 + n_2 - 2} \\ &= \frac{99 \cdot 70.07^2 + 99 \cdot 73.64^2}{198} = 5166.37 \end{aligned} \tag{5.1}$$

the CI of the difference between the means is calculated by:

$$\begin{aligned} \text{Confidence Interval} &= (\overline{X}_1 - \overline{X}_2) \pm t \cdot \sqrt{S^2 \cdot \left(\frac{1}{n_1} + \frac{1}{n_2}\right)} \\ &= (1299 - 1306) \pm 1.97 \cdot \sqrt{5166.37 \cdot \left(\frac{1}{100} + \frac{1}{100}\right)} \\ &= [-27.03, 13.03] \end{aligned} \tag{5.2}$$

Since the resulting confidence interval encloses the 0 no statistical significance in the difference of means is evident and therefore the rtg factor does not have a statistical significant influence on the resulting mean makespan of the model.

Furthermore, the influence of the different rtg values on the three distance measures is analyzed. Figure 5.18 visualizes the Hamming distances of the machine sequences between NLS and NLS-DT of the 100 instances in a boxplot for each rtg factor. Analogical figure 5.17 visualizes this for the action sequences. The start time distance between NLS and NLS-DT for the 100 instance is visualized in figure 5.19. All three distance measures show a low responsiveness for different rtg values. The mean Hamming distance of the machine sequences varies between 104.31 and 109.62 for the different rtg values. Thus, averaged over 100 instaces, only about 5 task are scheduled differently between the rtg that results in the highest mean Hamming distance and the rtg that results in the lowest mean Hamming distance. All the boxplots show a similar amount of outliers within the factor variations. The same properties hold for both of the other distances measures where for all rtg values a low variation between the respective mean values and the overall plot structure is visible. For the action sequences, the mean Hamming distance varies between 116.31 and 118.3. The mean start time distance varies between 0.70 and 0.73. Additionally, no property of the boxplots of the three distance measures shows any anomalie for the rtg factors 1.6 and 1.75 where the model performs the worst or the best respectively. This underlines that the variations in performance are probably due to the underlying variance of the general performance and not caused by the variation of the rtg

Overall the results indicate no meaningful influence of the rtg on the behavior and on the overall performance of the NLS-DT.

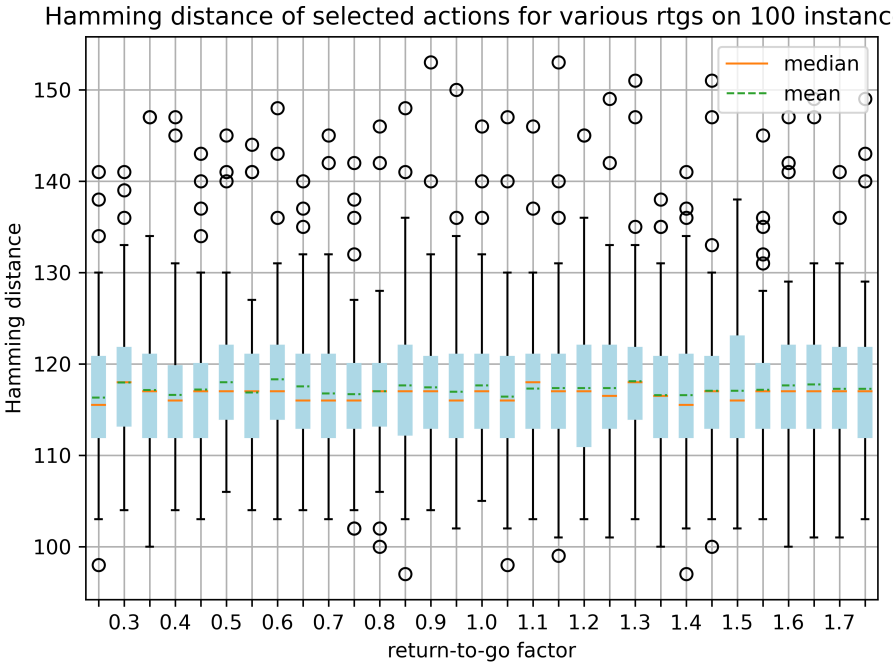

Figure 5.17: Boxplots of Hamming distance between selected actions of NLS and NLS-DT for various rtgs on 100 instances.

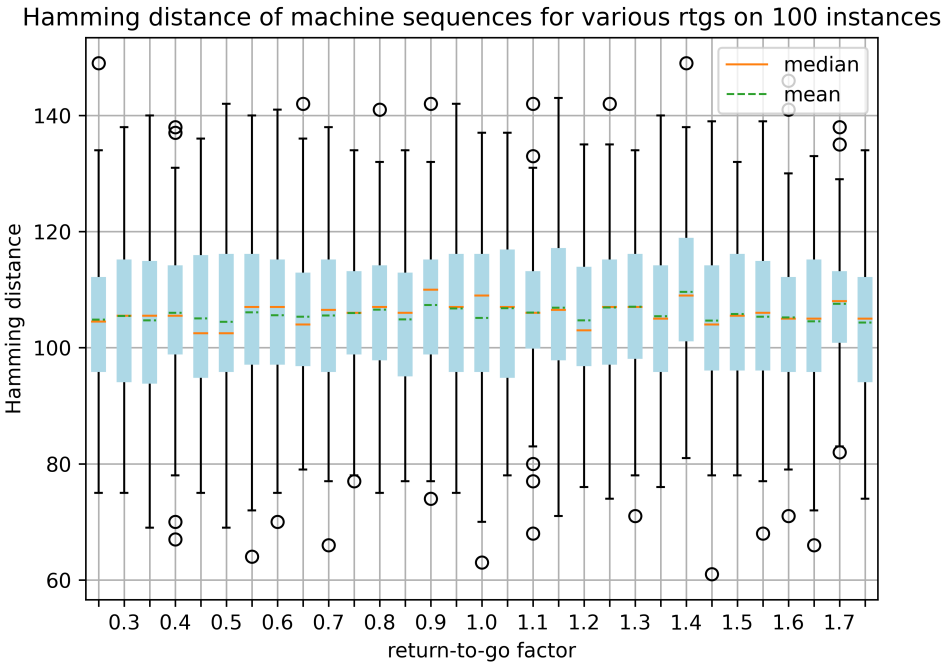

Figure 5.18: Boxplots of Hamming distance between machine sequences of final schedules of NLS and NLS-DT for various rtgs on 100 instances.

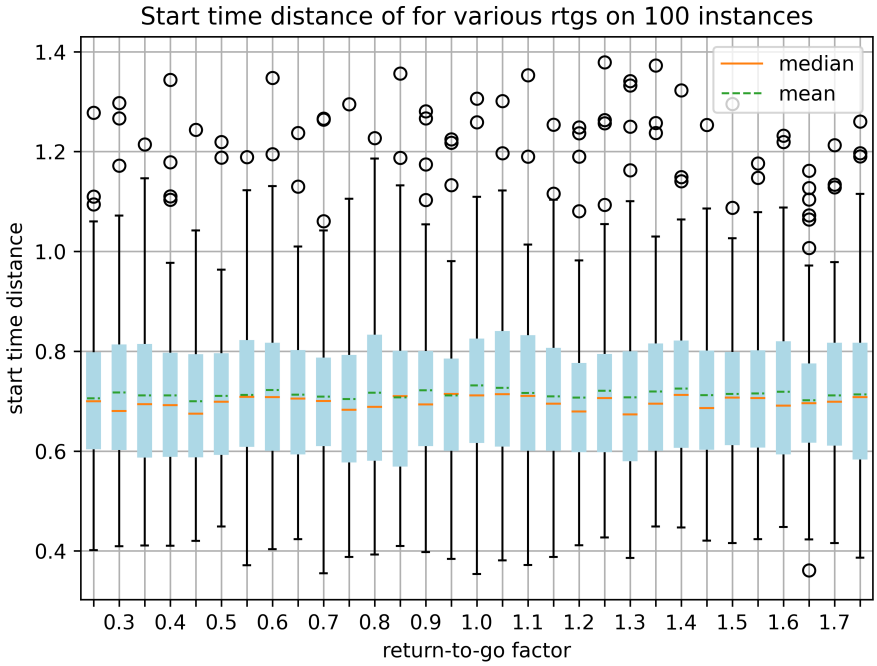

Figure 5.19: Boxplots of start time distances between final schedules of NLS and NLS-DT for various rtgs on100 instances.

6. Conclusion and Outlook

This thesis investigated the usefulness of the decision transformer (DT) architecture, a novel Reinforcement Learning (RL) approach, to solve the Job Shop Scheduling Problem (JSSP). In order to achieve this, two DT models were trained on two distinct origin-agents and then analyzed with respect to their performance and learned behavior. The first DT, called Dispatching Decision Transformer (D-DT), is trained on trajectories of an origin-agent that creates schedules by predicting the next task that is to be dispatched. The second DT, named NeuroLS Decision Transformer (NLS-DT), is trained on trajectories of an origin-agent that predicts actions to control a local search for finding schedules. The D-DT was trained on a dataset containing trajectories of a PPO agent solving 5,000 JSSP instances of size 6x6. The NLS-DT was trained on trajectories created from the original NeuroLS model for multiple problem sizes separately (15x15, 20x20, 30x20, 50x20 and 100x20).

To answer the first research question, which addresses whether the DT models are able to outperform their respective orign-agent, experiments were conducted to assess if the trained DT models achieve a better makespan than their origin-agents. For these experiments, the models were applied to solve JSSP instances of a specifically created test datasets and for the NLS-DT also on the established Taillard benchmark. The D-DT achieved a mean makespan between 65.12 and 82.92 on the created test dataset, while its origin agent achieves a mean makespan of 65.2. When trained on a stochastically selecting origin-agent, the D-DT outperformed the origin-agent by a larger magnitude, achieving a mean makespan of 77.05 compared to 87.375. Although this shows that the D-DT is at least in some scenarios able to outperform the origin-agent, in general it is not able to consistently outperform it. The NLS-DT was able to outperform its origin-agent in four out of the five problem sizes when applied to the Taillard instances. The most notable improvement was achieved for the 15x15 problem by lowering the optimality gap of the origin-agent by 2.79%. In case of the 30x20 instances of the Taillard benchmark the NLS-DT was not able to outperform the origin-agent. On the test dataset consisting of 100 instances for each size the NLS-DT was able to outperform the origin-agent on the 15x15, 30x20 and 100x20 problem sizes. On the 20x20 and 50x20 problems the NLS-DT performed equally to the origin-model.

The second research question, addressing whether the DT models learn their own strategies, was examined by analyzing how the chosen actions and the created schedules differ between the DT models and their respective origin-agents. For this, the Hamming distance of the chosen actions and the start time distance of the tasks of 100 instances were calculated. In case of the NLS-DT additionally the Hamming distance between the order of tasks on each machine was calculated. The analysis for the D-DT has shown that on average it chooses 16.74 of 36 actions differently

than the origin-agent. For the NLS-DT the difference measures were analyzed exclusively for 15x15 problem size. The analysis showed that the NLS-DT develops its own strategy, resulting in schedules that differ from those of the origin-model. On average 108 of 225 tasks per instance and 117 of 200 actions a chosen differently by the NLS-DT. In addition to the distance measures, the frequency of each action in the action space of the NLS-DT was compared to that of the origin-agent. This comparison has additionally proven that the behavior of the NLS-DT differs from that of the origin-agent

In order to answer the third research question, which deals with the influence of different rtg values on the behavior of the DTs, the models were run on their respective test dataset with rtg values between 0.25 and 1.75 times the calculated lower-bound of the instance to be solved. The experiments with the D-DT showed low responsiveness for the context-length 30 where lower rtg-values correlate with lower makespans and lower distance in both distance measures. For the NLS-DT the rtg showed no statistically significant difference on a 95% confidence interval between the difference of mean makespans of the worst and the best performing rtg. This showed that there is no statistically significant influence of the rtg on the achieved mean makespan of the NLS-DT. Furthermore, no effect of the rtg on the distance measures, and consequently on the behavior of the NLS-DT, was observed.

Regarding the D-DT, this study indicates that the DT architecture may not be appropriate for further enhancing models that create schedules by dispatching. A reason for this might be that there is no meaningful information from the sequence, potentially because much of the information that previous observations provide, like previously scheduled tasks and their finishing times, is already present in the observation of a single state.

Overall, the DT is able to reach state-of-the-art results in the JSSP. However, the improvements to their origin-agents are of low magnitude and in general the presented results do not justify the additional expense of training the DT for solving the JSSP. Nevertheless, for the 15x15 problem size the NLS-DT has shown strong improvement compared to its origin-agent and is thereby also better than other known RL-agents for that problem size. Since currently all NLS-DT models are learned with the hyperparameters tuned for the 15x15 problem and the NLS-DT achieved the highest performance improvement for that problem size, further research could be directed into a more exhaustive hyperparameter search particularly tailored to larger problem sizes.

Furthermore, as the NLS-DT ignores the rtgs and therefore seems to overfit on other inputs, further analysis could be directed into investigating how the rtg can be more effectively incorporated in the prediction process. Additionally, investigating the transformers attention mechanism through analysis of attention maps could provide deeper insights into the models behavior and potentially reveal areas for further refinement.

References

[1] H. Y. Fuchigami and S. Rangel, "A survey of case studies in production scheduling: Analysis and perspectives," *Journal of Computational Science*, vol. 25, pp. 425–436, 2018.

[2] P. Sharma and A. Jain, "A review on job shop scheduling with setup times," *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, vol. 230, no. 3, pp. 517–533, 2016.

[3] G. Da Col and E. C. Teppan, "Industrial-size job shop scheduling with constraint programming," *Operations Research Perspectives*, vol. 9, p. 100249, 2022.

[4] J. Siedentopf, *Job-Shop-Scheduling: Planung durch probabilistische lokale Suchverfahren*. Information - Organisation - Produktion, Wiesbaden: Deutscher Universitätsverlag, gabler edition wissenschaft ed., 2002.

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[6] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[7] N. Parmar, A. Vaswani, J. Uszkoreit, Łukasz Kaiser, N. Shazeer, A. Ku, and D. Tran, "Image transformer," 2018.

[8] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch, "Decision transformer: Reinforcement learning via sequence modeling," 2021.

[9] J. K. Falkner, D. Thyssens, A. Bdeir, and L. Schmidt-Thieme, "Learning to control local search for combinatorial optimization," vol. 13717, no. 2, pp. 361–376, 2023.

[10] J. Błażewicz, *Scheduling Computer and Manufacturing Processes*. Berlin and Heidelberg: Springer, second edition ed., 2001.

[11] T. Gonzalez, "Unit execution time shop problems," *Mathematics of Operations Research*, vol. 7, no. 1, pp. 57–66, 1982.

[12] S. Dauzère-Péres, *An Integrated Approach in Production Planning and Scheduling*, vol. 411 of *Lecture Notes in Economics and Mathematical Systems*. Berlin and Heidelberg: Springer, 1994.

[13] M. Kammer, M. van den Akker, and H. Hoogeveen, "Identifying and exploiting commonalities for the job-shop scheduling problem," *Computers & Operations Research*, vol. 38, no. 11, pp. 1556–1561, 2011.

[14] E. Nowicki and C. Smutnicki, "A fast taboo search algorithm for the job shop problem," *Management Science*, vol. 42, no. 6, pp. 797–813, 1996.

[15] J. Kuhpfahl and C. Bierwirth, "A study on local search neighborhoods for the job shop scheduling problem with total weighted tardiness objective," *Computers & Operations Research*, vol. 66, pp. 44–57, 2016.

[16] O. Holthaus and C. Rajendran, "Efficient dispatching rules for scheduling in a job shop," *International Journal of Production Economics*, vol. 48, pp. 87–105, Jan. 1997.

[17] E. Aarts and J. Karel Lenstra, *Local search in combinatorial optimization*. Princeton: Princeton University Press, 2003.

[18] P. van Laarhoven, *Theoretical and computational aspects of simulated annealing*. Phd thesis 4 research not tu/e / graduation not tu/e), Erasmus University Rotterdam, Netherlands, 1988. Proefschrift.

[19] M. Dell'Amico and M. Trubian, "Applying tabu search to the job-shop scheduling problem," *Annals of Operations Research*, vol. 41, p. 231–252, Sept. 1993.

[20] M. L. Pinedo, *Scheduling*. Boston, MA: Springer US, 2012.

[21] L. Perron and V. Furnon, "Or-tools," 2015.

[22] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím, "Ibm ilog cp optimizer for scheduling," *Constraints*, vol. 23, no. 2, pp. 210–250, 2018.

[23] V. Samsonov, K. Ben Hicham, and T. Meisen, "Reinforcement learning in manufacturing control: Baselines, challenges and ways forward," *Engineering Applications of Artificial Intelligence*, vol. 112, p. 104868, 2022.

[24] E. Taillard, "Benchmarks for basic scheduling problems," *European Journal of Operational Research*, vol. 64, no. 2, pp. 278–285, 1993.

[25] R. S. Sutton and A. Barto, *Reinforcement learning: An introduction*. A Bradford book, Cambridge, Massaschusetts and London: The MIT Press, 1998.

[26] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[27] S. Levine, A. Kumar, G. Tucker, and J. Fu, "Offline reinforcement learning: Tutorial, review, and perspectives on open problems," 2020.

[28] Q. Fournier, G. M. Caron, and D. Aloise, "A practical survey on faster and lighter transformers," *ACM Comput. Surv.*, vol. 55, jul 2023.

[29] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[30] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[31] C. Chen, Y.-F. Wu, J. Yoon, and S. Ahn, "Transdreamer: Reinforcement learning with transformer world models," 2022.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[33] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[34] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and leman go neural: Higher-order graph neural networks," 2018.

[35] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and C. Xu, "Learning to dispatch for job shop scheduling via deep reinforcement learning."

[36] J. Park, S. Bakhtiyar, and J. Park, "Schedulenet: Learn to solve multi-agent scheduling problems with reinforcement learning," *CoRR*, vol. abs/2106.03051, 2021.

[37] Z. Iklassov, D. Medvedev, R. Solozabal, and M. Takac, "Learning to generalize dispatching rules on the job shop scheduling," 2022.

[38] G. Bonetta, D. Zago, R. Cancelliere, and A. Grosso, "Job shop scheduling via deep reinforcement learning: a sequence to sequence approach," 2023.

[39] L. Zhao, W. Shen, C. Zhang, and K. Peng, "An end-to-end deep reinforcement learning approach for job shop scheduling," in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 841–846, IEEE, 5/4/2022 - 5/6/2022.

[40] Z. Zhang, H. Mei, and Y. Xu, "Continuous-time decision transformer for healthcare applications," in *Proceedings of The 26th International Conference on Artificial Intelligence and Statistics* (F. Ruiz, J. Dy, and J.-W. van de Meent, eds.), vol. 206 of *Proceedings of Machine Learning Research*, pp. 6245–6262, PMLR, 25–27 Apr 2023.

[41] Y. Lai, J. Liu, Z. Tang, B. Wang, J. Hao, and P. Luo, "Chipformer: Transferable chip placement via offline decision transformer," 2023.

[42] W. Ding, N. Majcherczyk, M. Deshpande, X. Qi, D. Zhao, R. Madhivanan, and A. Sen, "Learning to view: Decision transformers for active object detection," *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 7140–7146, 2023.

[43] S. Wang, X. Chen, D. Jannach, and L. Yao, "Causal decision transformer for recommender systems via offline reinforcement learning," *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2023.

[44] Y.-H. Wu, X. Wang, and M. Hamaya, "Elastic decision transformer," *ArXiv*, vol. abs/2307.02484, 2023.

[45] S. Hu, L. Shen, Y. Zhang, and D. Tao, "Graph decision transformer," 2023.

[46] C. A. Hepburn and G. Montana, "Model-based trajectory stitching for improved offline reinforcement learning," *ArXiv*, vol. abs/2211.11603, 2022.

[47] M. Janner, Q. Li, and S. Levine, "Offline reinforcement learning as one big sequence modeling problem," 2021.

[48] A. Kaparthy, "mingpt," 2020. https://github.com/karpathy/minGPT [Online; accessed 10-December-2023].

[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

[50] C. W. de Puiseau, J. Peters, C. Dörpelkus, H. Tercan, and T. Meisen, "schlably: A python framework for deep reinforcement learning based scheduling experiments," 2023.

[51] PyTorch, "Distributeddataparallel." https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html [Online; accessed 10-December-2023].

[52] T. van Ekeris, R. Meyes, and T. Meisen, *Discovering Heuristics And Metaheuristics For Job Shop Scheduling From Scratch Via Deep Reinforcement Learning*. Hannover : publish-Ing, 2021.

[53] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and C. Xu, "Learning to search for job shop scheduling via deep reinforcement learning," 2022.

[54] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[55] C.-Y. R. Huang, C.-Y. Lai, and K.-T. T. Cheng, "Chapter 4 - fundamentals of algorithms," in *Electronic Design Automation* (L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, eds.), pp. 173–234, Boston: Morgan Kaufmann, 2009.

[56] J.-B. Du Prel, G. Hommel, B. Röhrig, and M. Blettner, "Confidence interval or p-value?: part 4 of a series on evaluation of scientific publications," *Deutsches Arzteblatt international*, vol. 106, no. 19, pp. 335–339, 2009.

[57] Humboldt-Universität zu Berlin, "Konfidenzintervall für die Differenz zweier Erwartungswerte," 2018. [Online; accessed 10-December-2023].

[58] "Job shop instances and solutions." http://jobshop.jjvh.nl/.

A. Appendix

Source Code

The source code of the developed Decision Transformer model and its experiments is available on the appended DVD-Rom and in the following GitHub repository: https://github.com/fwUniGit/Decision-Transformers-For-JSSP

Mean Upper Bound of Taillard Intances

Problem size	Upper bound $\overline{C}_{max}$
15x15	1228.9
20x15	1364.8
20x20	1617.3
30x15	1790.2
30x20	1948.5
50x15	2773.8
50x20	2843.9
100x20	5365.7

Table A.1: Mean upper bound of Taillard intances. Source: [58]

Achieved Makespans of NeuroLS Decision Transformer on each Taillard Instance

Instance	NLS-DT$_{100}$	NLS-DT
15x15		
ta01	1335	1337.00
ta02	1303	1323.00
ta03	1305	1333.00
ta04	1289	1342.00
ta05	1317	1290.00
ta06	1304	1320.00
ta07	1276	1289.00
ta08	1273	1327.00
ta09	1448	1437.00
ta10	1380	1349.00
20x20		
ta21	1818	1818.00
ta22	1782	1773.00
ta23	1721	1738.00
ta24	1806	1778.00
ta25	1759	1759.00
ta26	1843	1860.00
ta27	1948	1974.00
ta28	1760	1781.00
ta29	1814	1790.00
ta30	1772	1793.00

Table A.2: Achieved makespan of NLS-DT for each taillard instance (15x15, 20x20)

Instance	NLS-DT$_{100}$	NLS-DT
30x20		
ta41	2365	2366.00
ta42	2250	2241.00
ta43	2209	2214.00
ta44	2258	2284.00
ta45	2260	2241.00
ta46	2353	2320.00
ta47	2178	2222.00
ta48	2205	2202.00
ta49	2326	2300.00
ta50	2237	2242.00
50x20		
ta61	3144	3154.00
ta62	3290	3269.00
ta63	2991	2991.00
ta64	2998	2989.00
ta65	3096	3106.00
ta66	3158	3100.00
ta67	3136	3130.00
ta68	3021	3032.00
ta69	3407	3407.00
100x20		
ta70	3281	3275.00
ta71	5793	5810.00
ta72	5426	5439.00
ta73	6055	6123.00
ta74	5502	5484.00
ta75	5903	5903.00
ta76	5701	5728.00
ta77	5593	5593.00
ta78	5726	5724.00
ta79	5531	5606.00
ta80	5411	5406.00

Table A.3: Achieved makespan of NLS-DT for each taillard instance (30x20, 50x20, 100x20)

Name, Vorname: ……

E r k l ä r u n g

Hiermit erkläre ich, dass ich die von mir eingereichte Abschlussarbeit (Master-Thesis) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Stellen der Abschlussarbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen wurden, in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

…… ……
Datum Unterschrift

E r k l ä r u n g

Hiermit erkläre ich mich damit einverstanden, dass meine Abschlussarbeit (Master-Thesis) wissenschaftlich interessierten Personen oder Institutionen zur Einsichtnahme zur Verfügung gestellt werden kann.

Korrektur- oder Bewertungshinweise in meiner Arbeit dürfen nicht zitiert werden.

…… ……
Datum Unterschrift